МИНОБРНАУКИ РОССИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ЧЕРЕПОВЕЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт информационных технологий

Кафедра математики и информатики

УЧЕБНО-МЕТОДИЧЕСКИЕ МАТЕРИАЛЫ ПО ДИСЦИПЛИНЕ

«ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ»

Направление подготовки (специальность):
01.03.02 Прикладная математика и информатика

Образовательная программа:
ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ

Очная форма обучения

Составители:

Лавров В.В., старший
преподаватель кафедры МиИ

г. Череповец - 2022

Перечень основной и дополнительной учебной литературы, необходимой для освоения дисциплины (модуля)

Основная литература:

1. Барков, И. А. Объектно-ориентированное программирование : учебник / И. А. Барков. — Санкт-Петербург : Лань, 2022. — 700 с. — ISBN 978-5-8114-3586-9. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/206699
2. Скворцова, Л. А. Объектно-ориентированное программирование на языке C++ : учебное пособие / Л. А. Скворцова. — Москва : РТУ МИРЭА, 2020. — 246 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/163862
3. Зайцев, М. Г. Объектно-ориентированный анализ и программирование : учебное пособие / М. Г. Зайцев. — Новосибирск : НГТУ, 2017. — 84 с. — ISBN 978-5-7782-3308-9. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/118271

Дополнительная литература:

1. Хорев П.Б. Объектно-ориентированное программирование : учебное пособие для вузов. - 4-е изд. - Москва : ИЦ "Академия", 2012. - 447 с.
2. Иванова Г.С. Объектно-ориентированное программирование : учебник для вузов / Иванова Г.С., Ничушкина Т.Н. ; под ред. Г.С.Ивановой. - Москва : Изд-во МГТУ им. Н.Э.Баумана, 2014. - 455 с.
3. Павловская, Т.А. C++. Объектно-ориентированное программирование : практикум : учебное пособие для вузов / Павловская Т.А., Щупак Ю.А. - СПб. : Питер, 2006. - 265 с.
4. Лафоре, Р. Объектно-ориентированное программирование в C++ / Лафоре Р. - 4-е изд. - СПб. : Питер, 2007. - 928 с.

Перечень ресурсов информационно-телекоммуникационной сети «Интернет», необходимых для освоения дисциплины (модуля), включая перечень информационных справочных систем (при необходимости)

1 Интерактивная доска.
2 http://www.ois.org.ua/spravka/mat/index.htm - электронная библиотека по математике.
3 http://eqworld.ipmnet.ru/ru/library.htm- учебно-образовательная физико-математическая библиотека.
4 http://www.exponenta.ru/- образовательный математический сайт.

Учебно-методические указания и рекомендации к изучению тем лекционных и практических занятий, самостоятельной работе студентов

Лекции

№ п/п	Тема лекции	Количество часов
1	Базовые понятия языка С++	2
2	Операторы языка С++	2
3	Адреса, указатели, ссылки, массивы	4
4	Многофайловая организация программы	2
5	Визуальная разработка приложений	4
6	Структуры данных. Стек. Очередь. Дек	4
7	Бинарный поиск	2
8	Введение в динамическое программирование: одномер- ная и двумерная динамика	4
9	Назначение и состав базовых решений на основе системы«1С:Предприятие (учебная версия)»	2
10	Цели, задачи, типы, подходы в разработке прикладных решений.	2
11	Проектирование прикладных решений на базе «1С:Предприятие (учебная версия)» с использованием встроенного языка программирования.	2
	Итого	30

Лабораторные работы

№ п/п	Тема лабораторной работы	Количество часов
1-3	Программирование на языке С++	8
4-6	Разработка приложений с графическим интерфейсом	12
7-10	Структуры данных и алгоритмы обработки данных	14
11-14	Прикладные решения на базе системы «1С:Предприятие (учебная версия)».	14
	Итого	48

Лекция 1

Базовые понятия языка С++

Структура языка С++.

Обобщенная структура языка С++ дана на рис. 1.

Рис. 1. Обобщенная структура языка С++

В левой части рисунка представлены средства языка, предназначенные для определения данных, объектов обработки программы. Типы данных определяют свойства данных, их внутреннее представление, возможные операции, которые можно производить с этими данными.

В следующей части рисунка представлены средства языка - операторы, предназначенные:

во-первых, для обработки данных путем, например, получения новых значений объектов программы в операторе присваивания;

во-вторых, для организации процесса обработки данных, например, организации повторяющейся обработки или организация разветвления процесса обработки.

Далее на рисунке представлены модули – относительно самостоятельные фрагменты программы для функционально законченной обработки данных, оформленные в виде функций.

В следующей части рисунка представлены средства и механизмы объектно-ориентированного программирования. Представлен новый тип данных, объединяющий данные и функции их обрабатывающие в единое целое – объект.

В последней части рисунка описаны средства обобщенного программирования, а именно множество контейнеров – структур данных, в которые можно помещать и извлекать данные любых типов, и набора обобщенных алгоритмов, позволяющих выполнять типовые операции над элементами контейнеров, не зависящие от вида контейнера. Абстракцию данных и алгоритмов обеспечивают шаблоны и итераторы.

Методика создания программ

Процесс разработки программ на С++ предполагает разбиение процесса решения задачи на ряд этапов, выполняющих функционально законченную обработку данных и формирование соответствующих функций. В результате программа представляет собой совокупность функций, одна из которых главная, называемая ***main***. Главная функция может располагаться в любом месте программы, но где бы она не находилась выполнение программы начинается и заканчивается именно в главной функции. Для главной функции можно использовать только имя ***main.***

Определение любой функции, в том числе и главной в С++, состоит из заголовка и тела функции:

<тип возвращаемого функцией результата> <имя> (список параметров)

{ тело функции – последовательность действий функции }

Приведем пример простой программы [3]:

```
#include <iostream.h>
int main ( )
{cout << "Программа стартовала"<<endl;
return 0;
}
```

В результате выполнения программы в консольном окне экрана выведется фраза: ***Программа стартовала.***

В первой строке – команда (директива) препроцессора, обеспечивающая включение в программу средств работы со стандартными потоками ввода/вывода данных. Эти средства подключаются к программе при использовании заголовочного файла с именем ***iostream.*** Стандартным потоком вывода по умолчанию является вывод на экран дисплея. Стандартный поток ввода обеспечивает чтение данных с клавиатуры.

Вторая строка – это заголовок функции ***main.*** В общем случае функция С++ вызывается другой функцией, а заголовок функции описывает интерфейс между ней и той функцией, которая ее вызывает. Слово, стоящее перед именем описывает информацию, которую функция передает в вызывающую функцию. Заголовок главной функции описывает интерфейс между функцией ***main( )*** и операционной системой.

Стандарт языка C++ требует, чтобы определение функции ***main*** начиналось со следующего заголовка: ***int main ( )*** - слово ***int*** указывает, что функция ***main()*** возвращает целое значение.

Возвращаемое функцией ***main()*** значение должно быть равно нулю, если выполнение программы прошло успешно. Круглые скобки после ***main*** требуются в соответствии с синтаксисом заголовка любой функции. В них помещается необязательный для главной функции список параметров. В данном примере список пуст. Некоторые программисты используют следующий также допустимый заголовок: ***void main (),*** означающий, что функция не возвращает результата.

Тело функции – это заключенная в фигурные скобки последовательность описаний, определений и операторов функции. В теле данной программы описаний и определений нет, а есть только два оператора.

Первый из них: ***cout << "Программа стартовала"<<endl;***

где ***cout*** - имя стандартного выходного потока. Данные для вывода передаются потоку с помощью операции <<. То, что нужно вывести, помещается от операции << справа. В данном случае это строка – ***"Программа стартовала",*** заключенная в кавычки последовательность символов. Вслед за строкой помещается еще одна операция вывода ***<<,*** а затем манипулятор ***endl*** (сокращение от "end of line" – "конец строки"). Его роль - очистить буфер выходного потока и поместить в выходной поток символ перехода на новую строку.

Второй оператор в программе ***return 0*** – оператор возврата. Он завершает выполнение программы и передает в точку ее вызова значение выражения, стоящего в операторе. Так как программа "запускается" на исполнение по команде операционной системы, то возврат будет выполнен к операционной системе.

Структура программы

Простая программа на C++ состоит из следующих элементов:

1) препроцессорные директивы, например:
   # include <имя файла>// - включение текстов стандартных файлов
   #define … // - замены в тексте
2) объявление глобальных объектов программы (типов, переменных, констант);
3) объявление одной главной функции ***main ();***
4) объявление ряда неглавных функций;
5) комментарии.

Этапы создания исполняемого кода программы

Рассмотрим технологию подготовки программ.

Классическая схема подготовки исполняемой программы [2] приведена на рис. 2.

Подготовка программы начинается с редактирования файла, содержащего текст программы, который имеет расширение ".cpp".

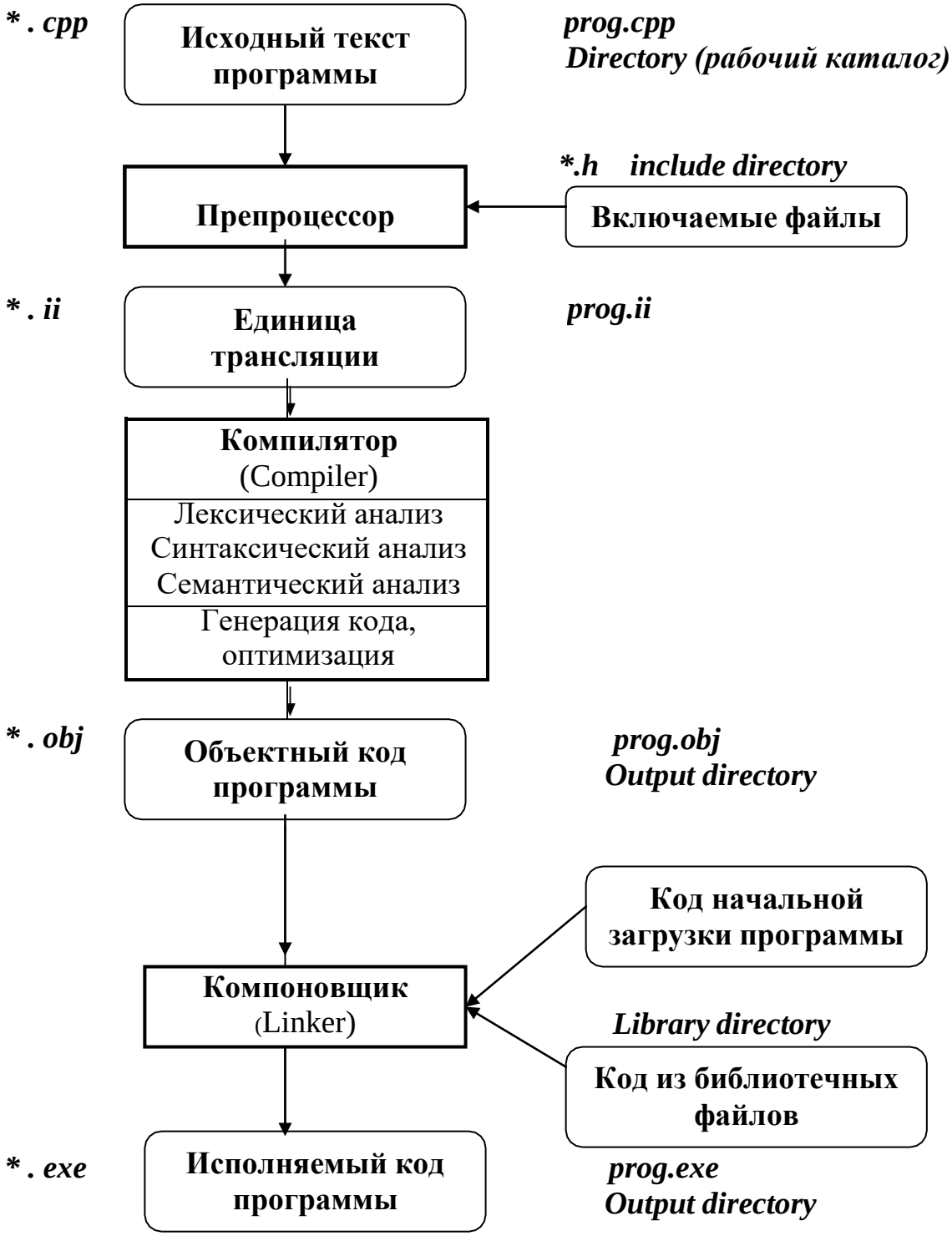


Рис. 2. Схема подготовки исполняемой программы

Перед шагом компиляции показан этап препроцессорной обработки текста программы. В нашем примере препроцессор обрабатывает директиву ***#include <iostream.h>*** и подключает к исходному тексту программы средства ввода/вывода. Результат препроцессорной подготовки при включении специальной опции компилятора помещают в файл с расширением ***".ii".***

Препроцессор сформирует полный текст программы – единицу трансляции (translation unit).

Затем выполняется компиляция программы, которая включает в себя несколько фаз: лексический, синтаксический, семантический анализ, генерация кода и его оптимизация. В результате компиляции получается объектный модуль - некий "полуфабрикат" готовой программы. Файл объектного модуля имеет стандартное расширение ".obj".

Компоновка (сборка) программы заключается в объединении одного или нескольких объектных модулей программы и объектных модулей, взятых из библиотечных файлов и содержащих стандартные функции. В результате получается исполняемая программа в виде отдельного файла (загрузочный модуль и программный файл) со стандартным расширением -".exe", который затем загружается в память и выполняется.

1.1. Лексические основы языка

Алфавит — это тот набор знаков (символов), который допустим в данном языке.

В алфавит языка C++ входят 96 символов.

- Из них 91 изображаемых символов:
  - прописные латинские буквы A..Z;
  - строчные латинские буквы a..z;
  - арабские цифры 0..9;
  - символ подчеркивания _ (рассматривается как буква).

Эти символы используются для образования ключевых слов и имён языка. В языке C++ прописные и строчные буквы различаются.

- В алфавит входят также 28 специальных символов:

, . ; : ? ' ! | / \ ~ * ( ) { } < > [ ] # % & ^ – = " +

- А также 5 – неизображаемых символов:
  - обобщенные пробельные символы (пробел, горизонтальная и вертикальная табуляция, перевод страницы, начало новой строки).

Комментарии

Комментарий — это последовательность любых знаков (символов) компьютера, которая используется в тексте программы для её пояснения. Обычно в тексте программы делают вводный комментарий к программе в целом (её назначение, автор, дата создание и т.д.), а далее дают комментарии к отдельным фрагментам текста программы, смысл которых не является очевидным. Компилятор языка программирования игнорирует комментарии, они нужны только для человека. В языке C++ имеется два вида комментариев: однострочные и многострочные.

Однострочный комментарий начинается с символов // (две косые черты). Всё, что записано после этих символов и до конца строки, считается комментарием. Например:

//Это текст однострочного комментария

Многострочный комментарий начинается парой символов **/*** (косая черта и звёздочка) и заканчивается символами ***/** (звёздочка и косая черта). Текст такого комментария может занимать одну или несколько строк. Всё, что находится между знаками **/*** и ***/**, считается комментарием. Например:

/ *Это текст большого*
многострочного комментария *******/*

В комментариях символы - это не только литеры из алфавита языка C++, но и любые возможные символы, включая русские буквы.

Лексемы языка

Из изображаемых символов алфавита формируются лексемы (tokens) языка. Лексемы - последовательности символов исходного кода программы, имеющие определенное смысловое значение.

В языке C++ имеются следующие категории лексем [1]:

- идентификаторы (identifier);
- ключевые (зарезервированные) слова (keyword);
- знаки операций (operator);
- константы – литералы (literal);
- разделители (знаки пунктуации – punctuator).

Рассмотрим эти лексические элементы языка подробнее.

Идентификаторы

Идентификатор — это имя программного объекта. В идентификаторе могут использоваться латинские буквы, цифры и знак подчеркивания. Прописные и строчные буквы различаются, например, sysop и SYSOP — два различных имени. Первым символом идентификатора может быть буква или знак подчеркивания, но не цифра. Пробелы внутри имен не допускаются. Длина идентификатора по Стандарту не ограничена, но некоторые компиляторы и компоновщики налагают на нее ограничения. Например, компиляторы фирмы Borland различают не более 32-х первых символов. При выборе идентификатора необходимо иметь в виду следующее:

- идентификатор не должен совпадать с ключевыми словами (см. следующий раздел) и с именами используемых стандартных объектов языка;
- не рекомендуется начинать идентификаторы с одного или двух символов подчеркивания, поскольку они могут совпасть с именами системных функций или переменных.

Ключевые (служебные) слова

Ключевые слова — это идентификаторы, зарезервированные в языке для специального использования. Эти идентификаторы имеют специальное значение для компилятора. Их можно использовать только в том смысле, в котором они определены. Список ключевых слов C++ [2] приведен в табл. 1.1.

Константы – литералы и перечисления

В языке C++ существует несколько видов констант: константы – литералы (неименованные константы), именованные константы, константы

перечислений и препроцессорные константы. Рассмотрим константы литералы – лексемы языка.

Таблица 1.1

Список ключевых слов C++

asm	else	new	this
auto	enum	operator	throw
bool	explicit	private	true
break	export	protected	try
case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_cast	typename
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	

Константы – литералы и перечисления

В языке C++ существует несколько видов констант: константы – литералы (неименованные константы), именованные константы, константы перечислений и препроцессорные константы. Рассмотрим константы литералы – лексемы языка.

Константа – литерал – это лексема, представляющая изображение фиксированного числового, символьного или строкового значения. Константы – литералы делятся на пять групп [1]:

целые (integer literal);

вещественные (floating literal – с плавающей точкой);

логические (boolean literal - булевские);

символьные (character literal - литерные);

строковые (string literal - строки).

Компилятор, выделив константу в качестве лексемы, "по внешнему виду" относит ее к определенной группе, а внутри группы по форме записи и по числовому значению – к тому или иному типу данных.

Целые константы могут быть *десятичными, восьмеричными и шестнадцатеричными*.

Десятичная целая константа определена как последовательность десятичных цифр, начинающаяся не с нуля, если это не число нуль, например: 16, 897216, 0, 21. *Отрицательные константы* - это константы без знака, к которым применена операция изменения знака.

Восьмеричные целые константы начинаются всегда с нуля, например 016 имеет десятичное значение 14.

Шестнадцатеричная константа - это последовательность шестнадцатеричных цифр, которой предшествует 0x. В набор шестнадцатеричных цифр кроме десятичных входят латинские буквы от ***a*** (или ***A***) до ***f*** (или ***F***). Таким образом, 0x16 имеет десятичное значение 22, а 0xF - десятичное значение 15.

Диапазон допустимых целых положительных значений - от **0** до **4294967295**. Константы, превышающие указанное максимальное значение, вызывают ошибку на этапе компиляции. Абсолютные значения отрицательных констант не должны превышать **2147483648**.

В зависимости от значения целой константы, компилятор по-разному представляет её в памяти ЭВМ. О форме представления данных в памяти ЭВМ говорят, используя понятие *тип данных.*

Соответствие между значением целых констант и автоматически выбираемыми для них типами данных отражено в табл. 1.2 для компиляторов семейства IBM PC/XT/AT (16-разрядных компиляторов) [3].

Таблица 1.2

Целые константы и выбираемые для них типы
Диапазоны значений констант

Десятичные	**Восьмеричные**	**Шестнадцатеричные**	**Тип данных**
от 0 до 32767	**от 00 до 077777**	**от 0x0000 до 0x7FFF**	**int**
	от 0100000 до 0177777	**от 0x8000 до 0xFFFF**	**unsigned int**
от 32768 до 2147483647	**от 0200000 до 017777777777**	**от 0x10000 до 0x7FFFFFFF**	**long**
от 2147483647 до 4294967295	**от 020000000000 до 037777777777**	**от 0x80000000 до 0xFFFFFFFF**	**unsigned long**
> 4294967295	**> 037777777777**	**> 0xFFFFFFFF**	**ошибка**

В заголовке <limit.h> определены предельные значения целых величин различных типов. Младший тип целых констант – ***int*** имеет диапазон допустимых значений от – **32767** до **32767 .**

Для 32-разрядных компиляторов диапазон значений типа ***int*** обычно равен **-2147483647** и **2147483647.** Для типа ***unsigned int*** ("беззнаковый целый") минимальное значение равно 0, а максимальное - **4294967295.** Предельные значения целых констант даны в заголовке <climit.h>. И в реализациях 32-

разрядных компиляторов, в которых не различаются типы ***int*** - "целое" и ***long*** - "длинное целое" при использовании целых чисел, превышающих значение **4294967295**, возникает ошибка. Если программиста по каким-либо причинам не устаревает тот тип, который компилятор приписывает константе, то он может явным образом изменить тип. Можно явно указать тип, используя суффиксы ***L, l (long)*** и ***U, u (unsigned)***. Например, константа ***64L*** будет иметь тип ***long***, хотя значению 64 должен быть приписан тип ***int***, как это видно из табл. 1.2. Для одной константы можно использовать два суффикса в произвольном порядке. Например, константы ***0x22ul, 0x33Lu*** будут иметь тип ***unsigned long***.

Перечислимые константы. Стандарт относит к целочисленным константам и перечислимые. Перечислимые константы (или константы перечисления) определяются с помощью служебного слова ***enum***. Это целочисленные константы, которым даются уникальные имена по следующему правилу:

enum { список идентификаторов констант = значения констант }

Имена констант уникальны, но значения могут повторяться. Кроме того, значения в определении можно опускать. Константы автоматически получат значения, которые будут начинаться от нуля, а затем увеличиваться на 1 для следующей константы. Возможны определения, в которых присутствуют определения констант со значением и без значения. В этом случае константы без значения автоматически получают значения. Правило о последовательном увеличении на 1 значения констант действует и в этом случае. Например, определение:

enum { a, b =0, c, d, e=2, f };

вводит следующие константы:

a = =0, b = =0, c= =1, d = =2, e = =2, f = =3.

Вещественные константы

Для представления вещественных чисел используются константы, представляемые в памяти компьютера в форме с плавающей точкой. Даже не отличаясь от целой константы по значению, вещественные константы имеют другую форму внутреннего представления в ЭВМ. При операциях с такими константами требуется использование арифметики с плавающей точкой. Компилятор распознает вещественную константу по внешним признакам.

Вещественная константа может включать следующие шесть частей: целая часть (десятичная целая константа); десятичная точка; дробная часть (десятичная целая константа); признак экспоненты "**e**" или "**E**"; показатель десятичной степени (десятичная целая константа, возможно, со знаком); суффикс ***F*** (или ***f***), либо ***L*** (или ***l***).

При записи констант с плавающей точкой могут опускаться целая или дробная часть (но не одновременно); десятичная точка или символ экспоненты с показателем степени (но не одновременно). Примеры констант с плавающей точкой:

125. .0 .17 3.141F 1.2e-5 .314159E25 2.77 2E+6L

Вещественная константа в экспоненциальном формате представляется в виде мантиссы и порядка. Мантисса записывается слева от знака экспоненты (E или e), порядок — справа от знака. Значение константы определяется как произведение мантиссы и возведенного в указанную в порядке степень числа 10. Обратите внимание, что пробелы внутри числа не допускаются, а для отделения целой части от дробной используется не запятая, а точка.

При отсутствии суффиксов вещественное число имеет форму внутреннего представления, которой соответствует тип данных ***double***. Добавив суффикс ***F*** или ***f***, константе придают тип ***float*** и соответственно тип ***long double,*** если в ее конце суффиксы ***L*** или ***l***. В табл. 1.3 приведены диапазоны возможных значений и длины внутреннего представления (размеры в битах) данных вещественного типа в конкретной реализации C++ (компилятор DJGPP).

Таблица 1.3

Данные вещественного типа

Тип данных	Размер в битах	Диапазон значений
float	32	от 3.4E-38 до 3.4E+38
double	64	от 1.7E-308 до 1.7E+308
long double	96	от 3.4E-4932 до 1.1E+4932

Булевские (логические) константы

Это два литерала типа ***bool: true*** (ИСТИНА) и ***false*** (ЛОЖЬ). Тип ***bool*** и литералы ***true*** и ***false*** были добавлены в последних версиях Стандарта языка. Наряду с логическими литералами продолжают действовать правила, унаследованные из ранних версий языка, в соответствии с которыми значению ЛОЖЬ соответствует число 0, а любое отличное от нуля значение воспринимается в логическом выражении как ИСТИНА.

Символьные (литерные) константы

Символьные константы – это один или два символа, заключенные в апострофы. Символьные константы, состоящие из одного символа, занимают в памяти один байт и имеют стандартный тип ***char***. Примеры: **'z', '*', '\012' , '\0',** '**\n**' – односимвольные константы. Двухсимвольные константы занимают два байта и имеют тип ***int,*** при этом первый символ размещается в байте с меньшим адресом (о типах данных рассказывается в следующем разделе) '**ab**', '**\x07\x07',** '**\n\t**' – двухсимвольные константы.

Внутри апострофов может быть любой символ, имеющий изображение. Однако в ПК есть символы, не имеющие графического изображения. Это, как правило, управляющие символы, например символ перехода на новую строчку. Для изображения таких символов используется комбинация из нескольких символов, начинающаяся с обратной косой черты.

Последовательности символов, начинающиеся с обратной косой черты, называются управляющими эскейп - последовательностями (escape-sequence).

Эскейп - последовательности используются:

- для записи символов, не имеющих графического изображения (например, \a — звуковой сигнал);
- для записи символов**:** символа апострофа ('), обратной косой черты **(** \ ), знака вопроса (**?**) и кавычки (**"**);
- для записи любого символа с помощью его шестнадцатеричного или восьмеричного кода, например, \073, \0xF5. Числовое значение кода должно находиться в диапазоне от 0 до 255.

В табл. 1.4 приведены допустимые значения эскейп-последовательностей. Управляющая последовательность интерпретируется как одиночный символ. В таблице **000** – строка от одной до трех восьмеричных цифр, **hh** – строка из одной или двух шестнадцатеричных цифр. Последовательность **'\0'** обозначает пустую литеру.

Таблица 1.4

Допустимые ESC–последовательности в языке C++

Изображение	Внутренний код	Обозначаемый символ	Действие или смысл
'\a'	**0x07**	**bel (**audible bell)	звуковой сигнал
'\b'	**0x08**	**bs** (backspace)	возврат на шаг(забой)
'\f '	**0x0C**	**ff** (form feed)	перевод страницы
'\n'	**0x0A**	**lf** (line feed)	перевод строки (LF)
'\r'	**0x0D**	**cr** (carriage return)	возврат каретки(CR)
'\t	**0x09**	**ht** (horizontal tab)	горизонтальная табуляция
'\v'	**0x0B**	**vt** (vertical tab)	вертикальная табуляция
'\\'	**0x5C**	\ **(**backslash**)**	обратная черта
'\''	**0x27**	' (single quote)	апостроф
'\"'	**0x22**	'' (double quote)	кавычка
'\?'	**0x3F**	**?** (question mart)	вопр. знак
'\000'	**000**	octal number	Восьмеричный код символа
'\0xhh'	**hh**	hex number	Шестнадцатеричный код

Для использования внутренних кодов символов нужна таблица, в которой каждому символу компьютера соответствует числовое значение его кода в десятичном, восьмеричном и шестнадцатеричном представлении. На IBM-совместимом ПЭВМ применяется таблица кодов ASCII [2].

Работу с символьными константами иллюстрирует программа:

```
#include<iostream.h>
void main()
{char c; int i;
c = 'ab'; cout << c << '\t';
```

i = 'ab'; cout << i <<'\t';
i = c; cout << i << '\t';
i = 'ba';
c = i; cout << c << '\t';}

В результате программы на экран будет выведено:

a 25185 97 b

здесь **25185** – двухбайтовое целое число, а **97** – код символа ***'a'*.**

Если выводить на экран односимвольную константу, то будет выведено изображение символа, но если эту же константу поместить, например, в арифметическое выражение, то значением константы будет ее десятичный внутренний код.

Для 32-разрядного компилятора допустимы константы – несколько символов, заключенных в апострофы, которые называются мультисимвольными (multicharacter literal) и имеют тип ***int.***

Строковые константы

Строка или строковая константа - это последовательность символов, заключенная в кавычки.

Внутреннее представление строки в памяти таково: все символы размещаются подряд, и каждый символ занимает 1 байт, в котором размещается внутренний код символа. А в конце строки компилятор помещает еще один символ, называемый байтовым нулем '\0'. Этот символ как любой другой занимает в памяти 1 байт, 8 двоичных разрядов, в которых находятся нули.

Среди символов строковой константы могут быть эскейп-последовательности, например:

*''**Монография** \'' **Турбо – Паскаль**\''**.** ''*

Строки, записанные в программе подряд или через пробельные символы, при компиляции конкатенируются (склеиваются). Таким образом, в тексте программы последовательность из строк:

''Миру - '' ''мир!''

эквивалентна одной строке: *''**Миру – мир!**''*

Длинную строковую константу можно размещать на нескольких строках в программе, используя еще и символ переноса строк - '\'

В строке может быть один символ, например, **''A'',** которая в отличие от символьной константы **'A'** занимает в памяти 2 байта. Строковая константа может быть пустой **''',** при этом ее длина равна 1 байту. Символьная константа не может быть пустой, запись **''** - не допустима.

Кроме непосредственного использования строк в выражениях, строку можно поместить в символьный массив, например, при его инициализации и обращаться к ней по имени массива (раздел 2).

Знаки операций

Знаки операций – это один из элементов выражений. Выражения есть правило получения значения. Результат операции - это всегда значение.

Знак операции — это один или более символов, определяющих действие над операндами. Внутри знака операции пробелы не допускаются. Операции делятся на унарные, бинарные, тернарные по количеству участвующих в них операндов. Один и тот же знак может интерпретироваться по-разному в зависимости от контекста.

В табл. 1.5 представлены операции, приоритеты (ранги) и ассоциативность операций [3].

Таблица 1.5

Операции языка C++ и их приоритеты

Ранг	**Операции**	**Ассоциативность**
1	**:: . -> ( ) [ ]**	→
2	**! ~ + - ++ -- & *** (тип) sizeof new deletetип ( ) typeid dynamic_cast static_cast reinterpret_cast const_cast	←
3	**.* ->***	→
4	*** / %** (мультипликативные)	→
5	**+ –** (аддитивные)	→
6	**<< >>** (сдвиги)	→
7	**< <= >= >** (сравнения)	→
8	**== !=** (сравнения)	→
9	**&** (поразрядная конъюнкция)	→
10	**^** (поразрядное исключающее ИЛИ)	→
11	**|** (поразрядная дизъюнкция)	→
12	**&&** (логическая конъюнкция)	→
13	**||** (логическая дизъюнкция)	→
14	**? :** (условная операция)	←
15	**= *= /= %= += -= &= ^= |= <<= >>=** операция присваивания	←
16	throw	
17	, (операция запятая)	→

Кроме стандартных режимов использования операций язык C++ допускает расширение (перегрузку) их действия, дает возможность распространения действия на объекты классов. Примером такой перегрузки являются операции поразрядных сдвигов >> и <<. Когда слева от них в выражениях находятся входные и выходные потоки, они трактуются как операции извлечения данных из потока >> и вывода данных в поток <<.

В табл. 1.6 дано краткое описание стандартных операций языка C++ [5].

Таблица 1.6

Сводка стандартных операций C++

Унарные операции	**Форма**
1. **&** - операция получения адреса некоторого объекта программы	& lvalue (lvalue – имя объекта программы)
2. ***** - разыменование, доступ по адресу к значению объекта	* a (a – указатель на объект)
3**. –** - унарный минус	**–** выражение
4. + - унарный плюс	**+** выражение
5**. !** – логическое отрицание (НЕ) **false –** если операнд истинный и **true –** если операнд ложный	!выражение !1 ==0, !(-5)==0(ложь) !0 == 1 (истина)
6. ++ - инкремент, увеличение значения операнда на 1 − префиксная форма - до использования его значения; − постфиксная форма – после использования его значения;	++ lvalue lvalue ++
7. -- - декремент, уменьшение значения операнда на 1 − префиксная форма; − постфиксная форма	-- lvalue lvalue--
8. **sizeof** – размер в байтах внутреннего представления объекта	sizeof выражение **sizeof (тип)**
9. **new** – динамическое выделение памяти	new имя_ипа new имя_ипа инициализатор
10. **delete** – освобождение динамически выделенной памяти	delete указатель delete [] указатель
11. **()** - явное преобразование типа	(тип) выражение
12. **()** - функциональная форма преобразования типа	тип (выражение) тип имеет простое имя
13. **typeid** – операция определения типа операнда	typeid (выражение) typeid(имя_типа)
14. **dynamic_cast** - операция приведения типа с проверкой допустимости при выполнении программы	dynamic_cast <целевой тип> (выражение)
15. **static_cast –** операция приведения типов с проверкой допустимости приведения во время компиляции.	static_cast <целевой тип> (выражение)

Продолжение табл. 1.6

16. **reinterpret_cast –** операция приведения типов без проверки допустимости приведения.	reinterpret_cast <целевой тип> (выражение)
17. **const_cast –** операция приведения типов, которая аннулирует действие модификатора const	const_cast <целевой тип> (выражение)
18**. :: -** операция доступа из тела функции к внешнему объекту	:: имя_объекта
Бинарные операции	**Форма**
Аддитивные:	
19. + - сложение арифметических операндов, или сложение указателя с целочисленным операндом	выражение + выражение
20. – - вычитание арифметических операндов или указателей	выражение – выражение
Мультипликативные	
21. * -умножение арифметических операндов	выражение * выражение
22. / -деление операндов арифметических типов. При целых операндах дробная часть результата отбрасывается	выражение / выражение 20/6 равно 3, -20/6 равно -3, 20/(-6) равно -3
23. % -получение остатка от целочисленного деления	выражение % выражение 13%4 равно 1, (-13)%4 равно -1 13%(-4) равно 1,(-13)%(-4) равно -1
Операции сдвига	
24. << - сдвиг влево битового представления левого целочисленного операнда на количество разрядов, равное значению правого операнда	lvalue << выражение
25. >> - сдвиг вправо битового представления левого целочисленного операнда на количество разрядов, равное значению правого операнда	lvalue >> выражение
Поразрядные операции	
26. & поразрядная конъюнкция (И) битовых представлений целочисленных операндов	выражение& выражение

Продолжение табл. 1.6

27. | - поразрядная дизъюнкция (ИЛИ) битовых представлений целочисленных операндов	выражение | выражение
28. ^ - поразрядное исключающее ИЛИ битовых представлений целочисленных операндов	выражение ^ выражение
Операции отношений (сравнения) Результат**: true (1)** , если сравнение истинно и **false (0)–** если ложно	
29. < - меньше	выражение < выражение
30. < = - меньше или равно	выражение <= выражение
31. > - больше	выражение > выражение
32. > = - больше или равно	выражение >= выражение
33. = = - равно	выражение = = выражение
34. ! = - не равно	выражение != выражение
Логические бинарные операции Результат**: true (1)** - истина и **false (0) –** ложь	
35. && - конъюнкция (логическое И) скалярных операндов и отношений	выражение **&&** выражение
36. || - дизъюнкция (логическое ИЛИ) скалярных операндов и отношений	выражение **||** выражение
Операции присваивания	
37. = простое присваивание левому операнду значения выражения - операнда из правой части	lvalue = выражение
38 операция = - составное присваивание левому операнду результат операции между левым и правым операндом; операциb *** / % + - << >> & | ^**	lvalue операция = выражение эквивалентно lvalue = lvalue операция (выражение)
Операции доступа к компонентам структурированного объекта	
39. **.** (точка) – прямой доступ к компоненту	имя_объекта.имя_компонента

Продолжение табл. 1.6

40. –> косвенный доступ к компоненту структурированного объекта, адресуемого указателем	указатель_на_объект–>имя_ компонента
Операции доступа к адресуемым компонентам класса	
41. **.*-** прямое обращение к компоненту класса по имени объекта и указателю на компонент	имя_объекта **.*** указатель_ на_компонент
42. **–>*** косвенное обращение к компоненту класса через указатель на объект и указатель на компонент	указатель_на_объект **–>*** указатель_на_компонент
43. **::** - бинарная операция расширения области видимости	имя_класса :: имя_компонента имя_пространства_имен :: имя
44. ( ) – операция круглые скобки – операция вызова функции	имя_функции (список_аргументов)
45. [ ] - операция квадратные скобки - индексация элементов массивов	имя_массива [индекс**]**
46. **? :** - условная (тернарная) операция;	выражение_1 **?** выражение_2 **:** выражение_3 если выражение_1 истинно, то значением операции является значение выражения_2, если выражение_1 ложно, то значением операции является значение выражения_3
47. **,** - операция запятая – это несколько выражений, разделенных запятыми, вычисляются последовательно слева направо; результатом операции является результат самого правого выражения	(список_выражений)

Операция и выражение присваивания

Операция присваивания обозначается символом '='. Простейший вид операции присвоения: ***l = v*** .

Здесь ***l*** - выражение, которое может принимать значение, ***v*** - произвольное выражение.

Операция присвоения выполняется справа налево, т.е. сначала вычисляется значение выражения ***v***, а затем это значение присваивается левому операнду ***l***. Левый операнд в операции присваивания должен быть так называемым **адресным выражением**, которое иначе называют ***lvalue***.

Примером адресного, или именующего выражения, является имя переменной. Адресным выражением никогда не являются константы. Не является ***lvalue*** и простое выражение, например, выражение ***a+b.*** Адресное выражение это объект, представляющий некоторый именованный участок памяти, в который можно поместить новое значение.

В языке С++ операция присваивания образует выражение присваивания, т.е. ***a = b*** означает не только засылку в ***a*** значения ***b***, но и то, что ***a = b*** является выражением, значением которого является левый операнд после присвоения. Отсюда следует, что возможна, например, такая запись:

a = b = c = d = e + 2;

Если тип правого операнда не совпадает с типом левого, то значение справа преобразуется к типу левого операнда (если это возможно). При этом может произойти потеря значения, например:

int i; char ch; i=3.14; ch=777;

Здесь ***i*** получает значение 3, а значение 777 слишком велико, чтобы быть представленным как ***char***, поэтому значение ***ch*** будет зависеть от способа, которым конкретная реализация производит преобразование из большего в меньший целый тип.

Существует так называемая комбинированная операция присваивания вида: ***a on =b,*** здесь ***on*** - знак одной из бинарных операций:

+ - * / % >> << & | ^ && ||.

Присваивание ***a on = b*** эквивалентно ***a = a on b***.

Разделители

Разделители или знаки пунктуаций входят в состав лексем.

- *Квадратные скобки* ***[]*** – ограничивают индексы массивов и номера индексированных элементов:

  int A[5] ={ …}; A[3]=5;

- *Круглые скобки* ***( ):***

  −выделяют условное выражение в условном операторе*:* ***if (x<0) x = -x;***

  −обязательный элемент в определении, описании и вызове функций:

  float F(float x, int n) //определение функции
  {тело функции}
  float F(float , int); // описании функции
  F(3.14, 10); // вызов функции

  −обязательны в определении указателя на функцию и в вызове функции через указатель:

  int (*pointer) ( ); //определение указателя
  (* pointer) ( ); //вызов функции через указатель на функцию

  −применяются для изменения приоритета операций в выражениях

  −элемент оператора цикла:

  for(int i =0, j = 3; i < j ; i+ =2, j++)
  {тело цикла};

  −используются при преобразовании типа:

(имя типа) операнд ; имя типа(операнд);

- в макроопределениях, обрабатываемых препроцессором:
  #define имя(список параметров) (строка замещения)

- *Фигурные скобки* ***{ }:***
  - Обозначают начало и конец составного оператора или блока.
    Пример составного оператора в условном операторе:
    if (x<y) {x++; y--};
    Пример блока, являющегося телом функции:
    float s (float a, float b)
    {return a+b;}
    После закрывающей скобки '**}**' не ставится точка с запятой '**;**'.
  - Используются для выделения списка компонент структур, объединений и классов:
    struct st { char*b; int c;};
    union un {char s [2]; unsigned int i; };
    class m {int b; public : int c, d; m(int);};
    Точка с запятой '**;**' обязательна после определений каждого типа.
  - Используются для ограничения списков инициализации массивов, структур, объединений, объектов классов при их определении:
    int k[ ] = { 1,2,3,4,5,6,7};
    struct tt {int ii; char cc;} ss ={777 , '\n'};

- *Запятая* ***','***
  - разделяет элементы списков формальных и фактических параметров функций;
  - разделяет элементы списков инициализации структурированных объектов;
  - разделитель в заголовке цикла ***for*:**
    for(int x=p, y=q, i=2; i < 100; z=x+y, x=y, y=z, i++)
    Запятую - разделитель следует отделять от запятой- операции с помощью круглых скобок:
    int i=1; int m [ ] = { i, (i=2 , i * i) ,i };

- *Точка с запятой* ***';'*** – завершает каждый оператор и пустой в том числе.

- *Двоеточие* ***':'***
  - служит для отделения метки от оператора: ***метка: оператор;***
  - при описании производного класса:
    class x : A, B {список компонентов};

- *Многоточие* ***'…'*** – используется для обозначения переменного числа параметров у функции

- *Звездочка* ***'*'*** используется как разделитель в определении указателя:
  int* r;

- *Знак* ***'='*** при определении объектов программы с инициализацией отделяет имя объекта от инициализирующего значения.

- *Символ '#'* используется для обозначения директив препроцессора.
- *Символ '&'* играет роль разделителя при определении ссылок:
  int d; int &c = d;

1.2. Представление данных

Данные - это формализованное представление информации. В компьютерной программе данные – это значение объектов программы. Данные С++ могут быть в виде констант и переменных.

Данные, которые не изменяются в программе, называются константами. Данные, зафиксированные в программе и изменяемые в ней в процессе обработки, являются переменными.

Переменные, перед тем как их использовать в программе, должны быть объявлены. Объявление переменной состоит в первую очередь из указания ее типа.

Понятие типа данных.

Тип данных характеризует:

- внутреннее представление данных в памяти компьютера;
- набор допустимых операций (действий);
- множество допустимых значений.

Классификация данных языка С++. Основные и производные типы

Все типы данных можно подразделить на простые (основные) — они предопределены стандартом языка, и сложные (или составные) — задаются пользователем. Данные простого типа нельзя разложить на более простые составляющие без потери сущности данного. Простые типы данных создают основу для построения более сложных типов: массивов, структур, классов. Простые типы в языке С++ — это целые, вещественные типы, символьный и логический тип и тип **void**.

Для определения и описания переменных основных типов используются следующие ключевые слова:

- **char** (символьный);
- **short** (короткий целый);
- **int** (целый);
- **long** (длинный целый);
- **float** (вещественный);
- **double** (вещественный с удвоенной точностью);
- **void** (отсутствие значения).

В табл. 1.7 приведены основные типы данных с диапазоном значений и назначением типов для компиляторов семейства IBM PC/XT/AT ([3]).

В таблице базовыми типами являются: ***char, int, float, double, void.*** Остальные получаются из них с использованием **модификаторов** типа: ***unsigned*** (беззнаковый), ***signed*** (знаковый), ***long*** (длинный), ***shot*** (короткий).

Целые типы

Целый тип данных предназначен для представления в памяти компьютера обычных целых чисел. Основным и наиболее употребительным целым типом является тип ***int***. Гораздо реже используют его разновидности: ***short*** (короткое целое) и ***long*** (длинное целое).

Таблица 1.7

ОСНОВНЫЕ ТИПЫ ДАННЫХ

Тип	**Размер (биты)**	**Диапазон значений**	**Назначение типа**
unsigned char	8	0…255	Небольшие целые числа и коды символов.
char	8	-128…127	Очень малые целые числа и ASCII-коды.
enum	16	-32768…32767	Упорядоченные наборы целых значений.
unsigned int	16	0…65535	Большие целые и счётчики циклов.
shot int	16	-32768…32767	Небольшие целые. Управление циклами.
int	16	-32768…32767	Небольшие целые. Управление циклами.
unsigned long	32	$0\ldots 2^{32}-1$	Астрономические расстояния.
long	32	$-2^{31}\ldots 2^{31}-1$	Большие числа, популяции.
float	32	3.4E-38 …3.4E+38	Научные расчёты (7 значащих цифр)
double	64	1.7E-308 … 1.7E+308	Научные расчёты (15 значащих цифр)
long double	80	3.4E-4932 …1.7E+4932	Финансовые расчёты (19 значащих цифр)
void			

По умолчанию все целые типы являются *знаковыми*, то есть старший бит в таких числах определяет знак числа: 0 — число положительное, 1 — число отрицательное. Для представления отрицательного числа используется дополнительный код.

Кроме знаковых чисел на C++ можно использовать *беззнаковые*. В этом случае все разряды участвуют в формировании целого числа. При описании беззнаковых целых переменных добавляется слово ***unsigned*** (без знака).

Для 32-разрядных компиляторов, которые не делают различия между целыми типами ***int*** **и** ***long*** целые типы представлены для сравнения в табл.1.8.

Таблица 1.8

Целые типы для 32-разрядных компиляторов

Тип данных	***Размер, байт***	***Диапазон значений***	
char	**1**	**-128 ... 127**	**(-2^7 – 2^7-1)**
short	**2**	**-32768 ... 32767**	**(-2^{15} – 2^{15}-1)**
int	**4**	**-2147483648 ... 2147483647**	**(-2^{31} – 2^{31}-1)**
long	**4**	**-2147483648 ... 2147483647**	**(-2^{31} – 2^{31}-1)**
unsigned char	**1**	**0 ... 255**	**(0 – 2^8-1)**
unsigned short	**2**	**0 ... 65535**	**(0 – 2^{16}-1)**
unsigned int	**4**	**0 ... 4294967295**	**(0 – 2^{32}-1)**
unsigned long	**4**	**0 ... 4294967295**	**(0 – 2^{32}-1)**

Символьный тип

В стандарте C++ для представления символьной информации есть два типа данных, пригодных для этой цели, — это типы ***char*** и ***wchar_t.***

Тип ***char*** используется для представления символов в соответствии с системой кодировки *ASCII* (*American Standard Code for Information Interchange — Американский стандартный код обмена информации*) [2,3]. Это семибитовый код, его достаточно для кодировки 128 различных символов с кодами от 0 до 127. Символы с кодами от 128 до 255 используются для кодирования национальных алфавитов, символов псевдографики и др.

Тип ***wchar_t*** предназначен для работы с набором символов, для кодировки которых недостаточно 1 байта, например, Unicode. Размер типа ***wchar_t*** обычно равен 2 байтам. Если в программе необходимо использовать строковые константы типа ***wchar_t***, то их записывают с префиксом ***L***, например, ***L"Слово".***

Таким образом, значениями символьных данных являются целые числа – значения их внутреннего кода. Однако в операторах ввода/вывода фигурируют сами символы, что иллюстрирует следующая программа:

```
#include <iostream.h>
char  c,b ;
void main()
{ c='*';       b= 55;
cout<<c<<'\t'<<b<<'\t';
cout<< (int)c;}
```

Результатом будет:

```
*       7       42
```

Вывелись символы *'**'* и символ с кодом 55 – символ '7'. И код символа *'**'*, как результат приведения типа символа к целому типу.

Логический тип

Логический (булевый) тип обозначается словом ***bool***. Данные булевого типа могут принимать только два значения: ***true*** и ***false*** и занимают в памяти 1 байт. Значение **false** обычно равно числу 0, значение ***true*** — числу 1.

Вещественные типы

Особенностью вещественных (действительных) чисел является то, что в памяти компьютера они практически всегда хранятся приближенно.

Имеется три вещественных типа данных: ***float, double*** и ***long double***. Основным считается тип ***double.*** Так, все математические функции по умолчанию работают именно с типом ***double***. В табл. 1.9 приведены основные характеристики вещественных типов [2].

Таблица 1.9

Основные характеристики вещественных типов

Тип данных	***Размер, байт***	***Диапазон абсолютных величин***	***Точность, количество десятичных цифр***
float	4	от 3.4E—38 до 3.4E+38	7 — 8
double	8	от 1.7E—308 до 1.7E+308	15 — 16
long double	10 (12)	от 3.4E-4932 до 1.1E+4932	19 — 20

Рекомендуется везде использовать тип ***double***. Работа с ним всегда ведётся быстрее, меньше вероятность заметной потери точности при большом количестве вычислений.

Тип void

Тип ***void*** — самый необычный тип данных языка C++. Нельзя определить переменную этого типа. Тем не менее, это очень полезный тип данных. Он используется:

- для определения функций, которые не возвращают результата своей работы;
- для указания того, что список параметров функции пуст;
- а так же этот тип является базовым для работы с указателями. Указатель на тип ***void - у***казатель ни на что, но который может быть приведен к любому типу указателей. Так всё программирование с использованием Win32 API построено на применении указателей на тип **void [7]**.

Новое обозначение типа

Используя спецификатор ***typedef*** можно в программе вводить удобное обозначение для сложных обозначений типов по следующему правилу:

typedef имя типа новое имя типа.

В следующем примере:

typedef unsigned char cod; cod symb;

объявлена переменная ***symb*** типа ***unsigned char.***

С помощью операций *, &, [ ], ( ) и механизмов определения структурированных типов можно создавать производные типы.

Представление некоторых форматов производных типов:

1. type имя [ ] – массив элементов типа type, например: ***long m [5];***

2. ***type1 имя (type2)*** - функция с аргументом ***type2*** и результатом типа ***type1***:

3. ***type * имя*** - указатель на объект типа ***type***, например: ***char*ptr***;

4. type* имя [ ] –массив указателей на объекты типа ***type***: ***int* arr[10];***

5. type (*имя) [ ] –указатель на массив объектов типа ***type***: ***int (*ptr) [10];***

6. ***type1* имя (type2)*** – функция, с аргументом типа ***type2*** и возвращающая указатель на объект типа ***type1***;

7. ***type1 (*имя) (type2)*** - указатель на функцию с аргументом типа **type2**, возвращающую значение типа ***type1***;

8. ***type1* (*имя) (type2)*** – указатель на функцию с аргументом ***type2***, возвращающую указатель на объект типа ***type1***;

9. ***type & имя = имя_объекта типа_type*** – определение ссылки;

10. ***type (&имя) (type2)*** – ссылка на функцию с аргументом ***type2***, возвращающую результат типа ***type1***;

11. ***struct имя{ type1 имя1; type2 имя2; };-*** объявление структуры;

12. ***union имя {type1 имя1; type2 имя2;};*** - объявление объединения;

13. ***class имя { type1 имя1; type2 имя2 (type3);};*** - определение класса.

Таким образом, типы можно разделить на ***скаляры, агрегаты и функции.***

Скаляры – это арифметические типы, перечисляемые типы, указатели и ссылки. ***Агрегаты*** (структурированные типы) – массивы, структуры, объединения и классы. ***Функции*** подробно рассмотрим в разделе 3.

Объявление переменных и констант в программе

Форма объявления переменных заданного типа:

имя_типа список имен переменных;

Например: **float x, X, cc2, pot_b;**

Переменные можно инициализировать, то есть задавать им начальные значения при их определении. Форма определения начальных значений проиллюстрирована на примере:

unsigned int year = 1999;

unsigned int year (1999);

Последнее определение с инициализацией разрешено только в функциях.

Допустимы следующие формы объявления именованных констант:

1) константы в объявлении перечисляемого типа;
2) с помощью спецификатора ***const*** :

const имя_типа имя_константы = значение

пример: ***const long M = 99999999***;

3) определение константы в препроцессорной директиве ***define***

пример: ***#define имя_константы значение_константы***

1.3. Объекты программы и их атрибуты

Переменная - это именованная область памяти. Имя переменной – это ссылка на некоторую область памяти. Переменная - частный случай *леводопустимого выражения***.**

Понятия леводопустимых и праводопустимых выражений

Леводопустимое выражение – это конструкция для обращения к некоторому участку памяти, куда можно поместить значение (***lvalue, l-значение*****).**

Понятие *леводопустимого выражения* включает все возможные формы обращения к некоторому участку памяти с целью изменения его содержимого. Название сформировалось по причине, что *леводопустимые выражения* располагаются слева в операторе присваивания.

Примеры *леводопустимых выражений*:

1) имена скалярных переменных;
2) имена элементов массивов;
3) имена указателей;
4) ссылки на ***lvalue*** (синонимы ***lvalue***);
5) имена элементов структурированных данных:

   имя структуры . имя элемента;

   указатель на структуру -> имя элемента элемента;

6) выражения с операцией '*****' - разыменования указателя:

int i, *p =&i; *p=7;

здесь объявляется переменная и указатель на эту переменную, а затем с помощью операции разыменования указателя осуществляется доступ к ячейке памяти переменной.

7) вызовы функций, возвращающих ссылки на объекты программы.

Выражения, которые могут располагаться в правой части оператора присваивания, называются *праводопустимыми* выражениями.

Примеры *праводопустимых выражений*:

1) любое арифметическое, логическое выражение;
2) имя константы;
3) имя функции (указатель константа);
4) имя массива (указатель константа);
5) вызов функции, не возвращающей ссылки.

В дальнейшем рассмотрении, в качестве ***lvalue*** будет рассматриваться переменная как объект программы.

Кроме типов для переменных явно или по умолчанию определяются:

- класс памяти (задает размещение объекта);
- область действия, связанного с объектом идентификатора (имени);
- видимость объекта;
- продолжительность существования объекта;
- тип компоновки (связывания).

Все перечисленные атрибуты (свойства) взаимосвязаны и должны быть либо явно указаны, либо они выбираются по контексту неявно при определении переменной. Рассмотрим их подробнее.

Тип как уже указывалось выше, определяет размер памяти выделяемого для значения объекта, правила интерпретации двоичных кодов значений объектов.

Класс памяти определяет размещение объекта в памяти и продолжительность его существования**.**

Явно задать *класс памяти* можно с помощью спецификаторов: ***auto, register, static, extern.***

Ниже указаны спецификаторы *класса памяти* и соответствующее им место размещения объекта:

- ***auto*** **-** автоматически выделяемая, локальная оперативная память - ***сегмент стека*** (временная память). Спецификатор ***auto*** может быть задан только при определении переменной блока, например в теле функции. Локальная продолжительность существования. Этим объектам память выделяется при входе в блок и освобождается при выходе из него. Вне блока переменные класса ***auto*** не существуют.
- ***register*** **-** автоматически выделяемая по возможности регистровая память (***регистры процессора***). Спецификатор ***register*** аналогичен **auto,** но для размещения переменной используется не оперативная память, а регистры процессора. Если регистры заняты другими переменными, переменные класса ***register*** обрабатываются как объекты класса ***auto***.
- ***static*** - статическая продолжительность существования, память выделяется в ***сегменте данных,*** объект существует до конца программы. Внутренний тип компоновки – объект будет существовать в пределах того файла с исходным текстом программы, где он определен. Этот класс памяти может быть приписан как переменным, так и функциям.
- ***extern*** - внешняя, глобальная память, выделяется в ***сегменте данных***. Объект класса ***extern*** обладает статической продолжительностью существования и внешним типом компоновки. Он - глобален, то есть, доступен во всех файлах программы. Этот класс может быть приписан как переменным, так и функциям.

Класс памяти и соответственно размещение переменной (в стеке, в регистре, в динамически распределенной памяти, в сегменте данных) зависит как от синтаксиса определения, так и от размещения определения в программе.

Область действия (ОД) идентификатора (имени) - это часть программы, в которой можно обращаться к данному имени (сфера действия имени в программе).

Рассмотрим все случаи:

1) имя определено в блоке (локальные переменные)**: ОД** - от точки определения до конца блока;

2) формальные параметры в определении функции (формальные параметры – это те же локальные переменные функции): **ОД** параметров – блок тела функции;
3) метки операторов: **ОД** меток – блок тела функции;
4) глобальные объекты: **ОД** глобальных объектов - вся программа от точки их определения или от точки их описания;
5) формальные параметры в прототипе функции: **ОД** параметров - прототип функции.

Область видимости (ОВ)

Понятие области видимости понадобилось в связи с возможностью повторных определений идентификатора внутри вложенных блоков или функций. В этом случае разрывается связь имени с переменной, то есть с участком памяти данной переменной, она становится "невидимой" внутри вложенного блока, хотя сфера действия имени сохраняется.

ОВ – это часть программы, в которой обращение к имени переменной позволяет обратиться к участку памяти, связанному с данной переменной.

ОВ может быть только меньшей или равной ОД. Следующая программа проиллюстрирует данную ситуацию [3]**:**

```
#include<iostream.h>
int k=0;
void main()
{int k=1;
        {cout << k;
        char k = 'A';       cout << k;
        cout << ::k;        cout << k ;}
cout << k;}
```

Результат выполнения программы: **1A0A1**

Объявлена глобальная переменная ***int k***. Затем в главной функции определяется локальная переменная с тем же именем и того же типа. Глобальная переменная становится "невидимой" в теле функции. Обращаться к "невидимой" внутри функции переменной можно, используя операцию доступа к внешнему объекту ***::k***.

В главной функции определен внутренний блок, в котором определена локальная переменная внутреннего блока с тем же именем ***char k.*** От точки определения символьной переменной до конца внутреннего блока локальная целочисленная переменная ***int k*** становится недоступной. После выхода из блока видимость (доступность) данной переменной восстанавливается.

Продолжительность существования

Продолжительность существования - это период, в течение которого идентификаторам в программе соответствуют конкретные объекты в памяти.

Определены три вида продолжительности: ***статическая, локальная и динамическая.***

Объектам со ***статической продолжительностью существования*** память выделяется в начале выполнения программы и сохраняется до конца программы.

Статическую продолжительность имеют все функции и файлы.

Все глобальные переменные (т.е. объявленные вне всех функций) обладают статической продолжительностью существования.

Локальные переменные (объявленные в функциях), со спецификатором ***static*** также имеют статическую продолжительность**.** Статическая переменная, локализованная в функции, не теряет своего значения при отработке функции. Инициализируется статическая переменная только при первом вызове функции, при втором вызове и последующих вызовах – значением этой переменной будет то, что сохранилось в памяти при предыдущем вызове.

Глобальные и статические переменные по умолчанию инициализируются нулями.

Если в функции имеется описание переменной со спецификатором ***extern***, это означает, что эта переменная глобальная и ее определение дано в другом месте вне функции. Такая переменная имеет статическую продолжительность существования.

Локальной продолжительностью существования обладают автоматические (локальные) переменные, объявленные в блоке. Такие переменные создаются при каждом входе в блок, где они определены и уничтожаются при выходе. Локальные переменные должны инициализироваться только явно, иначе их начальные значения не предсказуемы. Область действия локального объекта – блок. Спецификатор класса ***auto*** всегда избыточен, так как этот класс по умолчанию приписывается всем объектам, определенным в блоке.

Объекты с ***динамической продолжительность существования*** создаются и уничтожаются с помощью операторов в процессе выполнения программы по желанию программиста. Память таким переменным выделяется в области динамически распределяемой памяти, называемой **кучей.**

Для создания объекта используются операция ***new*** или функции *malloc()* и *calloc(),* а для уничтожения - операция ***delete*** или функция *free ().*

Операция: ***new имя_типа*** или ***new имя_типа инициализатор*** выделяет и делает доступным участок памяти для объекта данного типа и в выделенный участок заносит значение инициализатора, что не обязательно.
Операция возвращает адрес первого байта выделенного участка, или нулевое значение в случае неудачи. Если надо определить динамическую переменную типа ***int,*** следует объявить указатель на ***int*** и ему присвоить результат операции **new,** например: ***int*r = new (15);***

В дальнейшем доступ к выделенному участку памяти обеспечивается выражением ****r.***

Продолжительность существования выделенного участка – от точки создания до конца программы или до явного освобождения памяти операцией ***delete r;***

Тип компоновки

Если программа состоит из нескольких файлов, каждому имени, используемому в нескольких файлах, может соответствовать:

1) один объект, общий для всех файлов, или

2) один и более объектов в каждом файле.

Файлы программы могут транслироваться отдельно, и в этом случае возникает проблема установления связи между одним и тем же идентификатором и единственным объектом, которому он соответствует. Таким объектам компоновщик обеспечивает ***внешнее связывание*** при объединении отдельных модулей программы (первый случай).

Для объектов, локализованных в файлах, используется ***внутреннее связывание*** (второй случай)**;**

Тип компоновки компилятор устанавливает по контексту.

Определения и описания объектов программы.

Все описанные выше атрибуты (тип, класс памяти, ОД и так далее) приписываются объекту при его определении (объявлении) или при его описании, а также контекстом определения и описания.

В чем разница между определением или описанием.

При ***определении (definition)*** или ***объявлении*** объекта ему дается имя и устанавливаются атрибуты объекта, в соответствии с которыми выделяется нужный объем памяти и определяется формат внутреннего представления объекта. При ***определении*** происходит связывание имени объекта с участком памяти. ***Определение*** выполняет инициализацию объекта. ***Определение*** объекта может быть только одно в программе.

Описание или ***декларация (declaration)*** дает знать компилятору, что объект определен и напоминает свойства объекта (в основном компилятор интересуют типы).

Обычно, ***описание*** - это представление в конкретной функции уже объявленного где–то объекта. ***Описаний*** объекта может быть несколько в программе.

Нередко описание и определение по внешнему виду совпадают. Не совпадают они в следующих случаях:

1)описание – прототип функции;

2)описание содержит спецификатор ***extern;***

3)описывается класс или структурный тип;

4)описывается статический компонент класса;

5)описывается имя типа с помощью **typedef.**

Определения (объявление) переменных заданного типа имеет следующий формат:

s m тип имя1 иниц.1, имя2 иниц.2, … ;

где ***s*** **-** спецификатор класса памяти **auto , static , extern , register,**
m **–**модификатор ***const*** или ***volatile***:
const **-** указывает на постоянство объекта;
valatile **–** указывает на изменчивость объекта без непосредственного обращения к нему.
Синтаксис ***инициализации*** переменной, определяющий на этапе компиляции начальное значение переменной**:**
***имя = инициализирующее выражение*;**
либо:
имя (инициализирующее выражение) **–** применяется только в функциях.

1.4. Выражения и преобразования типов

Выражение – это правило получения нового значения. Выражения формируются из операндов, операций и круглых скобок. Порядок применения операций к операндам определяется рангами операций и их ассоциативностью. Круглые скобки, как и в математических выражениях, позволяют изменить порядок выполнения операций.

При выполнении операций нужно учитывать особенности представления в программе данных разных типов, являющихся операндами выражений. В связи с этим рассмотрим преобразование и приведение типов, допустимые в C++.

Явное приведение типа

Синтаксис функционального приведения типа:

type(выражение)

Примеры: ***int('*'), float (5/3), char(65).***

Функциональное преобразование типа не может применяться для типов, не имеющих простого имени. Например, конструкции:

unsigned long (15/7) или ***char*(0777)***

вызовут ошибку при компиляции.

Кроме функционального преобразования типа может использовать каноническую операцию приведения к требуемому типу. Для ее представления обозначение типа заключается в круглые скобки. Такая операция может применяться и для типов, имеющих сложное обозначение. Допустима записи: ***(unsigned long)15/7*** или ***(char*) 0777***

Другую возможность явного преобразования типов со сложным обозначением дает введение собственных обозначений с помощью ***typedef***:

typedef unsigned long ul; ul(15/7);
typedef char* pchar; pchar(0777);

В последних версиях языка C++ [5] введены еще четыре операции явного преобразования типа, имеющих следующий формат:

название_ cast <целевой тип> операнд

dynamic_cast - операция приведения типа с проверкой допустимости на этапе выполнения программы;

static_cast – операция приведения типов с проверкой допустимости приведения во время компиляции;
reinterpret_cast – операция приведения типов без проверки допустимости приведения;
const_cast – операция приведения типов, которая аннулирует действие модификатора ***const.***

Приведенным выше примерам соответствуют следующие выражения:
static_cast <unsigned long>(15/2); и ***static_ cast <char*>(0777);***

При преобразовании типов существуют некоторые ограничения. Но прежде чем остановиться на них, рассмотрим стандартные преобразования типов, выполняемые по умолчанию.

Стандартные преобразования типов

При вычислении выражений операции требуют, чтобы операнды имели соответствующий тип, а если требования не выполнены, производится неявное приведение типа.

Неявное приведение типов происходит при инициализации, когда тип инициализирующего выражения приводится к типу определяемого объекта. То же относится ко всем формам операции присваивания, происходит неявное преобразование типа выражения к типу левого объекта. Реально типы могут преобразовываться один в другой многими способами. Рассмотрит на данном этапе лишь некоторые из них. Преобразования типов, выполняемые неявно:

- преобразования в логические значения: в значения типа ***bool*** преобразуются обобщенные целые числа (включая и символы), вещественные числа и указатели; ненулевые значения преобразуются в ***true***, а нулевые - в ***false***;
- преобразование указателей: любой указатель может быть неявно преобразован в ***void****; значение **0** преобразуется в любой указательный тип; неконстантный указатель преобразуется в константный указатель того же типа.
- преобразование операндов в арифметических выражениях.

Последнее преобразование рассмотрим подробнее. При преобразовании нужно различать преобразования, изменяющие внутреннее представление данных и преобразования, изменяющие только интерпретацию внутреннего представления. Например, при преобразовании ***unsigned int*** в ***int*** изменяется только интерпретация внутреннего представления. При преобразовании ***double*** в ***int*** изменяется как длина участка памяти для внутреннего представления, так и способ кодировки. При таком преобразовании возможен выход за диапазон допустимых значений типа ***int*** , потеря значимости, точности и так далее. Именно поэтому в программах рекомендуется с осторожностью применять преобразования типов.

В арифметических выражениях происходит автоматическое преобразование типа операндов к общему типу – наиболее высокому типу согласно иерархии типов. Преобразование происходит по схеме, изображенной на рис.3 [4]:

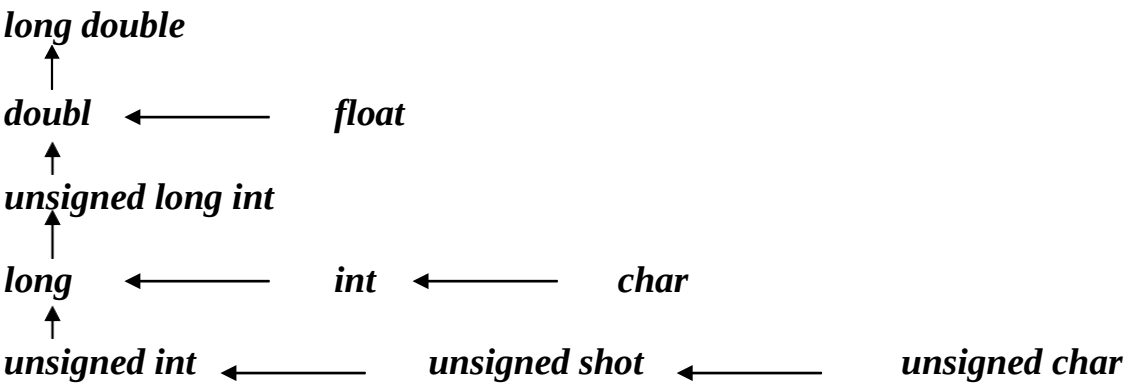


Рис.3. Схема последовательных преобразований типов операндов

Преобразования, гарантирующие сохранение значимости.

Используя в программе арифметические выражения, следует учитывать, что некоторые из них приводят к потерям информации и изменению числового значения. На рис.4 в соответствии со стандартом языка C++ [2] представлены стрелочками преобразования, гарантирующие сохранение точности и неизменность числового значения.

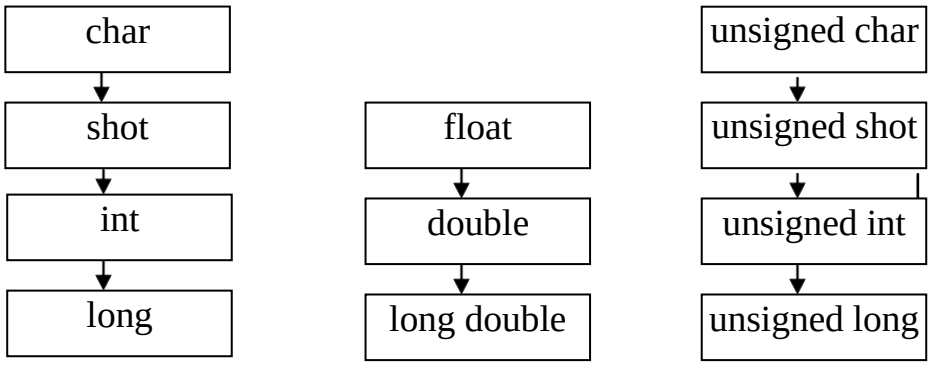


Рис.4. Последовательности преобразований типов, гарантирующие сохранение значимости

Лекция 2

1.5. Операторы языка C++

Операторы - это конструкции языка, определяющие действия и логику выполнения действий в программе.

По характеру действий различают два типа операторов: операторы преобразования данных и операторы организации обработки данных.

Первая группа – это последовательно выполняемые операторы. Рассмотрим их.

Последовательно выполняемые операторы - это:

- операторы присваивания;
- операторы ввода и вывода данных;
- операторы вызова функций;
- операторы-выражения.

Все они являются типичными операторами преобразования данных, и именно эти операторы используются при программировании линейных процессов вычислений.

Каждый оператор языка С++ заканчивается разделителем ***';'***. Любое выражение, после которого стоит ***';'***, воспринимается компилятором как оператор (не считая выражений, входящих в заголовок оператора ***for***).

Оператор присваивания является также оператором выражения, так как оператор формируется из выражения присваивания, в конце которого поставлена "точкой с запятой".

Оператор простого присваивания: ***lvalue = выражение ;***

Оператор составного присваивания: ***lvalue on= выражение ;***

Примеры: ***x*= i;*** ***i=x-4*i;***

Оператор – выражение – вызов функции, не возвращающей никакого значения.

Форма оператора: ***имя функции (список фактических параметров);***

Пример фрагмента программы:

```
void main ()
{ void cd (char);          // прототип функции
  cd ('*'); }          // оператор – выражение, вызов функции
```

Операторы – выражение – инкремент или декремент l-значения:

Формы операторов, с учетом префиксных и постфиксных форм операций инкремента и декремента:

++(lvalue); ***(lvalue)++;***

- -(lvalue); ***(lvalue)- -;***

Пример: ***i++;*** - возрастание значения переменной ***i*** на единицу.

Операторы ввода - вывода данных также являются операторами – выражениями.

Операторы используют непосредственно входные и выходные потоки из библиотеки потоковых классов, описания которых находятся в заголовочном файле **iostream.h.**

Препроцессорная директива: ***#include <iostream.h>***

подключает к программе библиотеку ввода/вывода, построенную на основе механизма классов. Ввод/вывод верхнего уровня в С++ построен на основе потоков, с которыми программа при исполнении обменивается данными.

Поток следует понимать как последовательность обмениваемых байтов между оперативной памятью и внешними устройствами (файл на диске, принтер, клавиатура, дисплей, стример и т.п.), или между различными участками оперативной памяти.

cin – имя стандартного входного потока по умолчанию, связанного с клавиатурой; ***cout*** – имя стандартного выходного потока по умолчанию, связанного с экраном дисплея;

>> - операция извлечение данных из потока или операция ввода;

<< - операция вставки данных в поток или операция вывода;

Операции возвращают ссылки на соответствующие потоки, то есть фактически сами потоки.

Операции извлечения данных из потока и вставки данных в поток являются основой для операторов ввода-вывода данных.

Форма оператора ввода (ввод данных с внешнего устройства в ОП):

cin >> ***lvalue*****;**

lvalue –это именованный участок оперативной памяти, значение которого можно изменять, частный случай – имя переменной. В последнем случае формат оператора ввода таков: ***cin >> имя переменной ;***

Из потока ***cin*** извлекается значение и помещается в оперативную память, выделенную под переменную.

Внутри ЭВМ данные хранятся в виде двоичных кодов, которые регламентированы для каждого типа данных. При вводе выполняется преобразование символов визуального представления данных из потока в двоичные коды внутреннего представления данных, при этом происходит автоматическое распознавание типов вводимых данных.

Форма оператора вывода (вывод данных из ОП на внешнее устройство):

cout**<<** ***выражение*** **;**

Из оперативной памяти извлекается значение выражения и помещается в выходной поток ***cout***. При этом происходит преобразование двоичных кодов типизированного значения выражения в последовательность символов, изображающих значение на внешнем устройстве. Интерпретация выводимого значения производится автоматически.

К этой группе операторов отнесем ***пустой оператор, составной и блок***.

Пустой оператор представляется символом "точкой с запятой", перед которым нет никакого выражения. Пустой оператор не предусматривает никаких действий. Используется там, где синтаксис языка требует присутствия оператора, а по смыслу программы никакие действия не должны выполняться.

Составной оператор - это последовательность операторов, заключенная в фигурные скобки.

Блок - определения, описания и последовательность операторов, заключенные в фигурные скобки.

Синтаксически и блок, и составной оператор являются единичными операторами. Внутри них после каждого оператора ставятся "точки с запятой", но после их закрывающей фигурной скобки символ '***;***' не ставится.

Описания и определения объектов программы, после которых стоит '***;***' также считаются операторами.

Перед любым оператором может стоять ***метка*** (включая и пустой оператор, описания и определения). *Метки* – это идентификаторы, локализованные в теле функции.

Иногда удобно поставить метку перед пустым оператором в конце функции.

К операторам управления работой программы относятся операторы выбора, циклы, операторы передачи управления.

Операторы выбора **-** реализуют в программе алгоритмические схемы ветвления. И в первую очередь к ним относится условный оператор.

Условный оператор**,** реализующий алгоритмическую схему развилка, имеет следующую форму:

if (выражение) оператор 1 else оператор 2;

в качестве ***выражения*** может быть любое скалярное выражение, которое автоматически приводится к логическому типу; каждый **из** ***операторов (1 или 2)*** – это по синтаксису один оператор простой или составной.

Если ***выражение*** истинно (то есть его значение не равно нулю), то выполняется ***оператор 1***(прямая ветвь алгоритма*),* в противном случае выражение – ложно и выполняется ***оператор 2*** (альтернативная ветвь).

Операторами 1,2 не могут быть описания и определения, но могут быть блоки с описаниями и определениями.

Перед ***else*** ставится ***';'***, если ***оператор 1*** – простой оператор.

Пример условного оператора, в котором в прямой и альтернативной ветвях находятся составной оператор и блок:

if(x>0) {x = -x; q (x);} else {int i=3; q(i*x);}

Условный оператор может иметь сокращенную форму:

if (выражение) оператор

Пример: ***if (a>b) a = -a;***

В случае ложности ***выражения*** никаких действий не выполняется.

Вложенные условные операторы

Если ***оператор 1*** или ***оператор 2*** в полной форме или ***оператор*** в сокращенной форме являются также условными операторами, то эти операторы называются вложенными условными операторами, которые также имеют прямую и альтернативную ветвь. При определении, к какому условному оператору какая относится альтернативная ветвь, существует правило: рассматриваются слева направо каждый ***else.*** Очередная альтернативная ветвь ***else*** принадлежит к ближайшему к ней, свободному (не связанному с другим ***else)*** оператору ***if.***

Рассмотрим пример:

if (x = =1) if (y = =1) cout << "x =1 и y =1"; else cout << "x != 1";

Условный оператор составлен неправильно. Действительно, при значениях ***x=1 и y!=1***, будет выведена неправильная фраза ***"x!=1"***

Ниже представлены два варианта правильно составленных операторов:

if (x = = 1) { if (y = = 1) cout << "x =1 и y =1";}else cout << "x != 1";

или

if (x= =1) if (y= =1) cout<<" x =1 и y =1 ";else ; else **cout** ***<< "x != 1";***

Оператор switch, реализующий алгоритмическую схему мультиветвление имеет две формы – форма переключателя и форма выбора варианта.

Форма переключателя:

switch (переключающее выражение)
{ case константное варажение1: операторы1;
…
case константное выражение N: операторы N;
default: операторы;
};

здесь ***переключающее выражение*** может быть любого перечисляемого типа: целочисленного или символьного; ***константные выражения*** должны быть того же типа (или приводящимися к нему), что и переключающее выражение и различны по значению; для каждой ветви алгоритма возможно использовать несколько *константных выражений*, например: ***case 1 : case 5: операторы;***

Сначала вычисляется переключающее выражение. Полученное значение сравнивается со значениями константных выражений. Если совпадает значение, то выполняются операторы данного варианта и операторы всех последующих вариантов, включая и вариант с меткой ***default***. Если значение не совпало ни с одним значением константных выражений, выполняется вариант с меткой ***default.*** Вариант ***default*** может располагаться в любой части внутри фигурных скобок, а может просто отсутствовать и тогда при отсутствии совпадения не выполняется ничего.

Форма альтернативного выбора:

switch (выражение)
{ case константное варажение1 : операторы 1; break;
…
case константное выражение N: операторы N; break;
default: операторы; break;
};

Если значение ***выражения совпадет с одним из константных значений***, то выполняются операторы только данного варианта, после чего управление программой передается оператору, следующему за оператором ***switch.***

Обработка любого варианта может включать кроме операторов еще и описания и определения объектов. В этом случае все это нужно заключить в фигурные скобки, тем самым превратить в блок.

Операторы цикла. Циклы реализуют алгоритмическую схему повторения обработки данных. В С++ определены три разных цикла:

- *цикл с предусловием:*

while (выражение-условие) тело_цикла

- *цикл с постусловием:*

do тело_цикла

while (выражение-условие);

- *цикл с параметром:*

for (инициализация_цикла; выражение-условие; выражения коррекции) тело_ цикла

Тело_цикла - это отдельный оператор, который всегда завершается точкой с запятой, либо составной оператор, либо блок. Только описание или определение не может быть телом цикла. Операторы цикла задают многократное выполнение операторов тела цикла.

Выражение-условие – во всех операторах скалярное (чаще всего логическое или арифметическое) определяет условие продолжение повторения обработки, если оно истинно (отлично от нуля). Прекращение выполнения цикла происходит в случаях:

- ложное (нулевое) значение *выражения-условия;*
- выполнение в теле цикла оператора передачи управления за пределы тела цикла.

Последнюю возможность рассмотрим позже.

Оператор ***while*** - ***оператор цикла с предусловием***. При входе в цикл вычисляется условие, если оно отлично от нуля, выполняется *тело_цикла.* Затем вычисление *выражения-условия* и выполнение *тела_цикла* повторяются, пока условие не станет ложным.

Следующая функция подсчитывает длины строки, заданной с помощью адресующего ее указателя – параметра функции [6]:

```
int lt (char*stroka)
{ int ln =0;
while (*stroka++) ln++ ;
return ln;}
```

Ниже приведены некоторые выражения условия:

while (a<b), while (point != NULL) (point-указатель),
while(point), while (point ! = 0).

Если в теле цикла не изменяется выражение- условие, могут возникнуть бесконечные циклы или "зацикливание".

Пример бесконечного цикла, с пустым оператором в теле: ***while (1);*** Такой цикл может быть закончен за счет событий, явно не предусмотренных в программе. Например, событие – указание операционной системе "снять задачу".

Оператор ***do*** является ***оператором цикла с постусловием***. При входе в цикл выполняется *тело_цикла*. Затем проверяется *выражение-условие* и, если его значение истинно, вновь выполняется тело цикла и так далее. К выражению-условию требования те же: оно должно изменяться при итерациях за счет операторов тела циклов.

Цикл ***for*** – ***цикл с параметром***. Заголовок цикла после зарезервированного слова ***for*** в круглых скобках включает три части: *инициализация_цикла, выражение-условие и список выражений для коррекции*,

которые разделяются двумя знаками '***;***'. Каждая из этих частей может отсутствовать, но даже если все они отсутствуют, два разделителя - '***;***' должны присутствовать в заголовке.

Инициализация_цикла – это выражение, или определения объектов одного типа. Вычисляется один раз при входе в цикл. Заканчивается точкой с запятой. Как правило, определяются и инициализируются параметры цикла.

Выражение-условие - логическое выражение, проверяется на каждой итерации цикла. Если его нет, полагают, что оно истинно. В этом случае сохраняется следующая за условием точка с запятой.

Выражения коррекции – список выражений, разделенных запятыми, которые вычисляются на каждой итерации после выполнения *тела_цикла* до следующей проверки условия.

Тело_цикла – может быть отдельным простым оператором, составным оператором или блоком. При входе в цикл один раз вычисляются выражения из секции *инициализация_цикла*. Вычисляется значение *выражения-условия,* если оно истинно, выполняются операторы *тела_цикла* и вычисляются *выражения коррекции.* Далее цепочка повторяется, пока *выражение-условие* не станет ложным.

for – цикл с заданным числом повторений, для подсчета числа итераций используется управляющая переменная – параметр цикла.

Примеры показывают, что управляющая переменная может изменяться на любую требуемую величину:

for(c =0 ; c<=100 ; c +=10)…; ***for(b =100; b >= -100; b -= 25)…;***

for(let =’A ; let <= ‘Z’; let ++)…; ***for(pr =0.0 ; pr <= 100.0 ; pr +=0.5)…;***

Область действия имени объекта, объявленного в заголовке цикла, определяется от точки объявления объекта до конца блока, в котором цикл присутствует.

Вложенные циклы

Разрешено вложение любых циклов в любые циклы.

Действует следующее правило: для каждой итерации внешнего цикла выполняются все итерации внутреннего цикла.

Операторы передачи управления

К операторам передачи управления относятся оператор безусловного перехода ***goto***, оператор возврата из функции ***return***, оператор выхода из цикла или переключателя ***break*** и оператор перехода к следующей итерации цикла ***continue***.

Оператор безусловного перехода имеет вид:

goto идентификатор метки;

Передача управления разрешена на любой идентификатор, помеченный меткой. Имя метки действует и уникально в теле функции, где расположен оператор. Существует запрет: нельзя передавать управление через определение,

содержащее инициализацию объекта, но можно обходить вложенные блоки, содержащие определение с инициализацией.

При использовании оператора ***goto*** рекомендуется:

1) не входить внутрь блока извне;
2) не входить внутрь условного оператора;
3) не входить извне внутрь переключателя;
4) не передавать управление внутрь цикла;

Есть случаи, когда использование оператора ***goto*** обеспечивает наиболее простое решение:

1) выход из нескольких вложенных циклов или переключателей;
2) к одному участку программы перейти из разных мест функции;

Оператор возврата из функции имеет вид***:***

return выражение; или ***return;***

Возвращает в точку вызова функции значение ***выражения***, которое, если присутствует, может быть только скалярным. Если ***выражение*** в операторе ***return***. отсутствует, то возвращаемое функцией значение имеет тип ***void.***

Оператор выхода из цикла или переключателя break прекращает выполнение операторов цикла или переключателя и осуществляет передачу управления к следующему за циклом или переключателем оператору.

При многократном вложении циклов и переключателей оператор ***break*** не может передать управление из внутреннего уровня на самый внешний. ***break*** позволяет выйти из внутреннего цикла в ближайший внешний цикл.

Оператор перехода к следующей итерации continue используется только в циклах. Завершает текущую итерацию цикла и начинается проверка условия продолжения цикла. Его действия эквивалентны оператору передаче управления в самом конце тела цикла.

Лекция 3

Адреса, указатели, ссылки, массивы

2.1. Указатели и адреса объектов

Понятие переменной определялось как имя ячейки памяти, в которой хранится значение указанного типа. Каждая ячейка памяти имеет свой уникальный **адрес**. Чтобы получить адрес переменной (простой или структурированной), используется унарная операция **&,** которая применима к объектам, размещенным в памяти и имеющим имена**.**

Указатель – это объект (переменная), значениями которого являются адреса участков памяти, выделенных для объектов конкретных типов.

Организация памяти в процессорах семейства 80x86

Основная память ПЭВМ – это память с непосредственным (прямым) доступом к участку памяти с любым адресом, не зависимо от того, к какому участку выполнялось предыдущее обращение.

Наименьшим адресуемым участком памяти является байт, содержащий 8 бит (двоичных разрядов). Оперативная память представляет собой ***последовательность пронумерованных байтов***, физические адреса (номера байтов), которых начинаются от 0 и возрастают.

Участки памяти, кратные 16 байт называются ***параграфами***, которые пронумерованы от нуля до 65535 . Всего 65536 параграфов, это соответствует объему оперативной памяти = 1 Мбайт.

Физический адрес параграфа (номер байта, с которого начинается параграф) равен произведению номера параграфа на 16. Начало любого параграфа может быть принято за начало ***сегмента памяти***. Длина сегмента памяти не может превышать 64Кбайт

Процессоры семейства 80x86 используют сегментированную организацию памяти.

В полном сегментированном адресе любого объекта два 16-разрядных числа: ***0xHHHH : 0xHHHH***, где ***H*** **–**любая шестнадцатеричная цифра. Первое число – это номер параграфа, с которого начинается сегмент. Это число называется ***сегментной частью адреса*.** Второе число – определяет смещение от начала сегмента интересующего нас первого байта объекта и называется ***смещением*** или ***относительной частью адреса*.**

Обе части адреса – это четырехразрядные шестнадцатеричные числа, или 16-разрядные двоичные числа. Они могут принимать значения от 0 до 65535 (64Кбайт)

По полному сегментному адресу формируется 20-разрядные физические адреса: сегментная часть* 16 (номер первого байта сегмента) + относительная часть (смещение в сегменте) = ***0xHHHH * 0x10 + 0xHHHH = =0xHHHH0 + 0xHHHH = 0xHHHHH.***

Для работы с сегментированными адресами в процессорах семейства 80x86 имеются регистры сегментов:

- ***CS*** *(Code Segment)*-регистр сегмента кода программы, используется для формирования адресов выполняемых команд;

- ***DS*** *(Data Segment)*-регистр сегмента данных, используется для формирования адресов данных;
- ***SS*** *(Stack Segment)*- регистр сегмента стека, для формирования адресов данных из стека;
- ***ES*** *(Extra Segment)* - регистр сегмента расширения дополнительного сегмента данных.

Схема возможного размещения сегментов в памяти

Процессор может одновременно адресовать 4 различных сегмента памяти, каждый из которых может быть размером до 64 Кбайт. Они могут пересекаться и даже могут быть размещены все в одном участке размером в 64Кбайт. Схематично расположение сегментов представлено на рис.5.

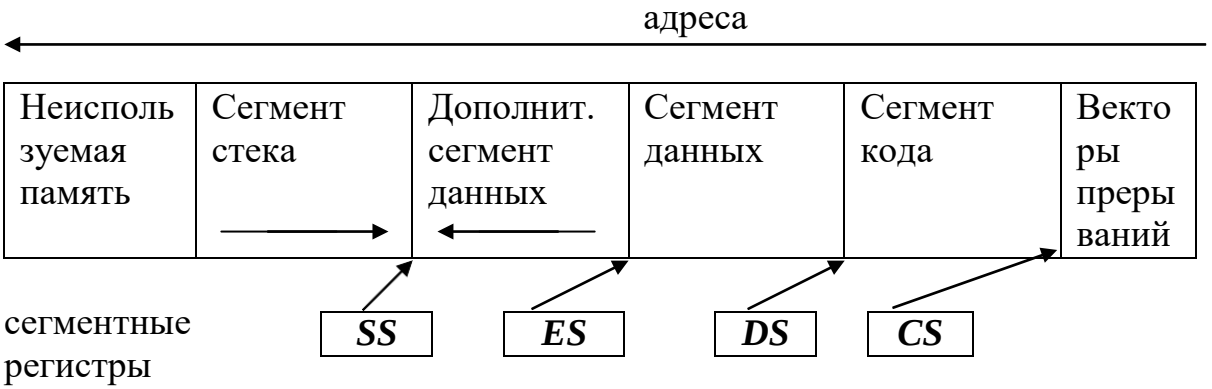


Рис. 5. Возможное размещение сегментов в памяти

При сегментной организации адресуется 2^{20} **(~** 1 млн.**)** байтов. Но для формирования адреса использовались два 16-разрядных двоичных числа. С помощью этих чисел можно было бы адресовать на прямую без разбиения на сегменты $2^{16} * 2^{16} = 2^{32}$ (~ 4 млрд.) байтов.

Все возможности не используются из-за допустимости пересечения сегментов. Например, комбинации чисел **0x0000:0x0413, 0x0001:0x0403, 0x0021:0x0203** адресуют один и тот же физический адрес **0x413 (0x00000+0x0413==0x00010+0x0403==0x00210+0x0203==0x00413).**

Значениями указателей в общем случае **(*far*)** является число типа ***long,*** старшие байты которого состоят из сегментной части адреса, а младшие – из смещения ***0xHHHHHHHH.***

Указатели в компиляторах TC++ и BC++ делятся на 4 группы, для описания которых введены модификаторы (служебные слова), позволяющие выбирать внутреннее представление указателей:

- **near** – ближние;
- **far** – дальние;
- **huge** – нормализованные;
- **_cs, _ds, _es, _ss** – сегментные.

Ближние указатели имеют длину 2 байта и представляют смещение в конкретном сегменте. Адресуют только 64Кбайта памяти.

Дальние **указатели** занимают 4 байта, содержат и номер сегмента и смещение. Адресуют только 1Мбайт памяти. Разные сочетания сегментной части и смещения могут адресовать один и тот же физический адрес, один и тот же байт могут адресовать несколько указателей.

Нормализованные указатели имеют длину 4 байта и воспринимаются,

как одно 32-разрядное значение, но позволяют однозначно адресовать только 1Мбайт памяти. Любому физическому адресу представляется единственное сочетание сегмента и смещения. Дается номер только того сегмента, для которого смещение не больше 15 (от 0 до 15). Например, для физического адреса **0x413** полный адрес будет **0x0041:0x0003** и соответственно значение нормализованного указателя однозначно равно **0x00410003.**

Сегментные указатели – это ближние указатели, имеют длину 2 байта и хранят значение смещения в известных сегментах (данных, стеке, и так далее). Для доступа к этим сегментам введены регистровые переменные (_CS, _DS, _ES, _SS), в которых хранится сегментная часть адреса.

А сегментные указатели задают смещение в известных сегментах. Например, определим указатель**: nt _ss * pss;** значение указателя **pss** позволяет задавать смещение в сегменте стека.

2.2. Объявление указателей

В простейшем случае объявление указателя имеет вид:

type * имя указателя;

В определении указателя обязательно должен быть указан тип данных, на который "указывает" указатель, так как указатель несет информацию об адресе участка памяти и о размерах этого участка памяти.

type – тип объекта, на который "указывает" указатель;

***type ** – тип указателя, если рассматривать указатель как переменную.

Например, в объявлении:

int*A, *B, *C, D;

объявлено три указателя ***A , B , C*** на данные типа ***int*** и переменная ***D***- типа ***int.***

Инициализация указателей

Инициализация указателей имеет две формы:

type * имя указателя = инициирующее выражение ;

type * имя указателя (инициирующее выражение);

В качестве ***инициирующего выражения*** может быть:

1) явно заданный адрес участка памяти: ***type * имя = (type *) 0x158e0ffc;***

2) выражение, возвращающее адрес объекта с помощью операции ***'&'***:

type count= ; type_1* iptr = (type_1*) & count;

3) указатель, имеющий значение:

char ch = '+', *R = & ch; char *ptr = R;

4) инициирующее выражение равно пустому указателю: ***NULL*** – специальное обозначение указателя, ни на что не указывающего. ***char * ch (0)*** эквивалентно ***char * ch (NULL).***

Доступ к значению объекта, адресуемому указателем, обеспечивает операция разыменования, выражение: ****имя указателя;*** позволяет получить значение самого объекта:

char cc = 'a', *pc = &cc; cout << *pc;

и изменить это значение: ****pc = '+'; cin >> *pc;***

Можно сказать, что ****указатель*** **–** обладает правами переменной.

Если определен указатель без инициализации**: *char *p,*** использовать выражение ****p*** в операциях присваивания или в операторе ввода не правомерно. Указатель ***p*** не обладает значением - адресом участка памяти, куда можно было

занести значение. Указателю можно присвоить адрес участка памяти объекта типа ***char*:**

1) ***p = new char ;*** //динамическое выделение, **delete (p) -** освобождение памяти
2) p = (char *) 0x157e0ffc; //значение адреса преобразуется к указателю ***char****
3) p =(char*) malloc (sizeof (char));
// динамическое выделение памяти **free (p)** – освобождение памяти
4) p присвоить адрес переменной типа ***char***: ***char c; p = &c;***
5) p присвоить значение другого указателя на данные типа ***char****:

char s , *ptr=&s; p=ptr;

Теперь допустимы операции ****p ='+' ; cin >> *p;***

Указатели константы и на константы

Указатели бывают константами (то есть значение указателя нельзя поменять в программе) и указателями на константы (то есть указатель, адресующий неизменный участок памяти).

В общем случае определение константного указателя на константу имеет вид: ***type const * const имя указателя;***

в этом определении ***type const –*** тип константы, на которую "указывает" указатель. А в последующей части после разделителя ***'*' - const имя указателя*** определяется имя константного указателя**.** В этом случае нельзя изменить значения указателя и нельзя изменить значение участка памяти, на который "указывает" указатель.

Определение константного указателя имеет вид:

type* const имя указателя;

Например, в определении: ***char * const K = (char *) 1047;***
K – это указатель, значение которого невозможно изменить. Возможно менять значения по адресу ***K,*** то есть допустимы операции: ****K = '*'*** или ***cin >> *K;***

Определение указателя на константу имеет вид:

type const * имя указателя;

Например, после определений:
const float A = 56,1 float B = 1.1; float const *pA = &A ;
не допустимы операции изменения значения ****pA***, но допустимо разорвать связь указателя ***pA*** с константой ***A,*** присвоив указателю адрес другой переменной: ***pA = &B;*** и тогда допустимы, например**: **pA=0.1*** или ***cin >> *pA;***

Еще раз отметим свойства операции взятия адреса - ***'&'*** столь важной для инициализации указателей**.** Операция ***'&'*** применима только к объектам, имеющим ***имена*** и ***размещенным в ОП.***

2.3. Типы указателей и операции над ними

Типы указателей могут быть стандартными (указатели на объекты основных типов) и производными (указатели на массивы, указатели на указатели, указатели на функции, указатели на константы, на структуры, на объединения, на объекты классов и на данные типов определенных пользователем с помощью спецификатора ***typedef).***

Указатели на основные типы данных **-** на арифметические данные и на символьные.

Пример работы с указателем на данные арифметического типа ***int***:

int n = 6, *pn = &n;

cout << pn<<'\t' << *pn; //будет выведено***: 0x1E10FFC 6***

//***адрес значение***

Пример работы с указателем на данные символьного типа ***char***:

char c = 65, *pc= &c;

cout << pc << '\t' << *p << '\t' << &c; //будет выведено ***A A A,***

то есть при выводе указателя, разыменованного указателя и адреса будет выводиться сам символ с кодом ***65*** – это прописная латинская буква ***A***

Если мы хотим вывести значение адреса символьной переменной и ее внутренний код надо поступить так:

cout << (void*)pc <<'\t'<< (int)*pc //результат: ***0x1e76a0c2 65***

явно привести указатель к типу ***void***, а разыменованный указатель к типу ***int.***

Операция ***- & имя*** дает однозначный результат – адрес данного объекта.

Результат операции разыменования ****имя_указателя –*** неоднозначен и зависит не только от значения указателя, но и от его типа, который указывает размер участка памяти, который будет доступен:

В следующем примере один и тот же адрес переменной рассматривается как значение указателей разных типов:

{long L = 0x12345678L;

char*ch=(char*)&L; int*I =(int*)&L; unsigned long*UN=(unsigned long*)&L;

cout << hex; //манипулятор вывода в шестнадцатеричном формате

cout << (int)*ch << '\t' << *I << '\t' << *UN << endl;

cout << (void*)ch << '\t' << I << '\t' << UN << endl; …}

Результат фрагмента программы:

0x78 0x5678 0x12345678

0x18efoffc 0x18efoffc 0x18efoffc

В первой строке выведены значения ячеек памяти в шестнадцатеричном формате, во второй строке значения указателей (адресов каждой ячейки).

Значения самих указателей совпали, а значения объектов зависят от размера участка памяти, на который "указывает" соответствующий указатель (1 байт, 2 байта или 4 байта).

Рассмотрим указатель типа ***void*.*** Такой указатель может иметь значение – адрес какого-либо объекта программы, однако он не содержит информации о размере этого объекта в памяти, и, следовательно, посредством такого указателя нельзя работать с объектом. Это можно исправить, используя явное преобразование типа указателя.

Указатель типа ***void**** приводится к любому указателю явным преобразованием типа, например:

void* p; int* ip, i= 10; p = &i ; ip = (int*)p;

Любой указатель приводится к типу ***void**** по умолчанию:

int i, j, *b = &i; void* a = &j;

a = b; **–** допустимо, так как ***b*** автоматически приводится к типу ***void*,***

b = a; –недопустимо, надо явно приводить ***a*** к типу ***int*: b= (int*) a;***

Операции над указателями

1) разыменование **(*);**
2) преобразование типа явное (только каноническая форма);
3) присваивание;
4) взятие адреса (**&);**
5) аддитивные операции (сложение и вычитание);
6) инкремент – автоувеличение **(++);**
7) декремент – автоуменьшение **(--);**
8) операции отношения.

Первые три операции уже рассмотрены выше. Рассмотрим остальные.

Понятие указателя на указатель

Указатель как переменная, обладает адресуемой ячейкой в памяти (размером в общем случае – 4 байта) и, следовательно, к имени указателя применима операция взятия адреса. Адрес указателя можно присвоить указателю на указатель. Рассмотрим пример, в котором объявляется указатель на указатель.

int i =3, *a =&i, **b=&a; ***//int**b;*** указатель на указатель
cout << b << '\t' << *b << '\t' << **b;
0x1204fff0 0x1204fff4 3
//(&a) (a, &i) (*a, i)

Сначала будет выведено значение указателя ***b***, равное адресу указателя ***a***. Затем выведется значение ****b*** – результат разыменования указателя ***b***, которое равно значению указателя ***a*** – адресу переменной ***i***. И последним выведется значение *****b*** - результат разыменования указателя ****b,*** равное значению результата разыменования указателя ***a,*** равное значению самой переменной ***i.***

Аддитивные операции

I. Вычитание

Вычитание указателей на объекты одного и того же типа

Разность однотипных указателей – это «расстояние» между двумя участками памяти, адресуемыми указателями, выраженное в единицах кратных длине объекта того типа, который адресуется указателями:

type*p1, *p2… //далее указатели получают значения адресов объектов
p1 – p2 = ((long) p1 – (long) p2) / sizeof(type)

//здесь ***(long) p –*** значение указателя

Рассмотрим фрагмент программы:

```
…{int a = 1, b = 4, *aa = &a, *bb = &b;
cout << aa << '\t' << bb << endl;
cout << (aa – bb) << '\t' << ( (long) aa – (long) bb );}
```

Результат выполнения:

0x18e80ffe 0x18e80ffc **-** адреса переменной ***a*** и переменной ***b***
1 2 **-** значение разности указателей (1) и разность значений указателей (2).

Программа иллюстрирует, что:

1) разница двух указателей на соседние объекты одного типа равна ***1***

2) разница значений указателей на соседние объекты типа ***int*** равна ***2***

3) адрес первого объявленного объекта больше чем следующего**,** так как стек заполняется от больших адресов к малым адресам.

Вычитание из указателя целое число:

При вычитании из указателя целого числа: ***указатель – K;*** формируется значение указателя меньшее на величину ***K*sizeof(type),*** где ***K*** – вычитаемое число, ***type*** - тип объекта, к которому относится указатель.

II. Сложение указателя с целым значением

При сложении указателя с целым числом: ***указатель + K;*** формируется значение указателя большее на величину ***K* sizeof(type).*** Пример:

```
…{int a=0, b = 1, c = 3, d = 5, *p = &d;
cout << *p << '\t' << p << '\t' << *(p + 1) << '\t' << (p + 1) << '\t';
cout << *(p + 2) << '\t' << (p + 2) << '\t'<< *(p + 3) << '\t' << (p + 3);}…
```

Результат:

```
5       0x15e80ff4  3       0x15e80ff6  1       0x15e80ff8  0       0x15e80ffa
```

III. Инкремент ++ (декремент --) – увеличивает (уменьшает) значение указателя на ***sizeof(type).*** Указатель смещается к соседнему объекту с большим (меньшим) адресом.

Таким образом, обладая правами объекта (как именованного участка памяти), указатель имеет адрес, длину внутреннего представления и значение:

1) значения указателя – это адреса объектов, на которые "указывает" указатель;
2) адрес указателя получают операцией: ***&имя указателя***;
3) длина внутреннего представления

sizeof (void*) == sizeof (char*) == sizeof (любой указатель) == 4

4) раз указатель это объект, то можно определять указатель на указатель и так сколько нужно раз: ***int i = 10, * p1 = &i, **p2 = &p1, ***p3 = &p2;***

type* – тип указателя на указатель на переменную типа ***type;***

type** – тип указателя на указатель, который указывает на указатель на переменную типа ***type;***

При этом: ****p3 = = p2,*** ****p2 == p1,*** ****p1 == i,***

то есть к значению переменной можно добраться путем последовательных разыменований указателя ***p3***: ****(*(*p3)),*** что эквивалентно – ****** p3.***

2.4. Ссылки

Ссылка – это другое имя уже существующего объекта (именованного участка памяти).

При определении ссылки ей не выделяется память, ссылка дает новое имя уже существующему участку памяти.

При определении ссылок инициализирующее выражение обязательно!

Инициализирующим выражением может быть только ***l-value***, то есть объект, имеющий место в памяти и имя.

Объявление ссылки имеет вид:
type &имя_ссылки (инициирующее выражение)
type &имя_ссылки = инициирующее выражение

Пример:
int J = 5, &I = J; //можно было отдельно определить **int &I = J**
I++; ***//I = = 6 и J = = 6***
I+ = 10; ***//I == 16 и J == 16***

Все действия, которые происходят со *ссылкой,* происходят и с *переменной,* инициализирующей ссылку, и наоборот все, что происходит с *переменной*, происходит и со *ссылкой*.

При работе со ссылками действуют следующие правила:
1) ссылка не может иметь тип ***void*** **(*void & a=b* –** не допустимо);
2) ссылку нельзя создавать динамически;
3) нет ссылок на др. ссылки (***int& &a = r*** **–**не допустимо)
4) нет указателей на ссылки (***int & *p= &r*** **–**не допустимо)
5) нет массивов ссылок

long a = 1, b = 2, c = 3; long& MR[ ] = {a, b, c} - не допустимо

5) ссылка навсегда связана с объектом инициализации;
6) объект может несколько ссылок

*Инициализация ссылок не обязательна****:***
1) в описаниях внешних объектов ***extern float & b;***
2) в описания ссылок на компоненты классов;
3) в спецификациях формальных параметров;
4) в описании типа, возвращаемого функцией;

Результатом применения операции ***sizeof (имя ссылки*****)** является размер в байтах участка памяти, выделенного для инициализатора ссылки. Операция **&** возвратит адрес инициализатора ссылки.

2.5. Массивы

Массив – это совокупность данных одного типа, рассматриваемых как единое целое. Все элементы массива пронумерованы. Массив в целом характеризуется ***именем***. Обращение к элементам массива выполняется по их номерам (***индексам*****),** которые всегда начинаются с **0**.

Массивы могут состоять из числовых данных, символов, строк, указателей и так далее. Символьные массивы, как правило, представляют в программе текстовые строки.

Если для обращения к какому-то элементу массива достаточно одного индекса, массив называется *одномерным*.

Если данные удобно представлять в форме таблицы, тогда для обращения к конкретному элементу надо задать два индекса: номер строки и номер столбца. Такие массивы называются *двумерными(или матрицами)*.

Массивы с числом индексов больше 1 называются *многомерными*.

Форма объявления одномерного массива (вектора):

type имя массива [K];

K – константное выражение, определяет *размер* массива (количество элементов в массиве); ***type –*** тип элементов массива.

Форма объявления многомерного массива:

type имя массива [K 1] [K2] …[K N];

type – тип элементов массива;

N -размерность массива- количество индексов, необходимое для обозначения конкретного элемента;

K1…KN - количество элементов в массиве по ***1…N***- му измерению, так в двумерном массиве ***K1*** – количество строк, а ***K2*** – количество столбцов;

Значения индексов по ***i***-му измерению могут изменяться от ***0*** до ***Ki – 1;***

K1*K2*…*KN – размер массива (количество элементов массива).

Например, ***float Z[13][6];*** определяет двумерный массив, первый индекс которого изменяется от 0 до 12, а второй индекс - от 0 до 5.

Внутреннее представление массивов в оперативной памяти

При определении массива для его элементов выделяется участок памяти, размеры которого определяются количеством элементов массива и их типом: ***sizeof (type)* количество элементов массива,*** где ***sizeof(type) –*** количество байтов, выделяемое для одного элемента массива данного типа.

Операция ***sizeof*** имеет две формы: ***sizeof(тип)*** и ***sizeof(объект).*** Учитывая это, а также то, что имя массива – это имя структурированной переменной, размер участка памяти, выделенного для всего массива, можно определить также из следующего выражения: ***sizeof (имя массива).***

В оперативной памяти все элементы массива располагаются подряд. Адреса элементов одномерных массивов увеличиваются от первого элемента к последнему. В многомерных массивах элементы следуют так, что при переходе от младших адресов к старшим наиболее быстро меняется крайний правый индекс массива. Так, при размещении двумерного массива в памяти сначала располагаются элементы первой строки, затем второй, третьей и т. д.

Например, элементы массива **int T [3][3]** будут располагаться так:

первая строка			**вторая строка**			**третья строка**		
T[0][0]	**T[0][1]**	**T[0][2]**	**T[1][0]**	**T[1][1]**	**T[1][2]**	**T[2][0]**	**T[2][1]**	**T[2][2]**

Возрастание адресов ⟶

Инициализация числовых и символьных массивов.

Инициализация - это задание начальных значений объектов программы при их определении, проводится на этапе компиляции.

Инициализация одномерных массивов возможна двумя способами: либо с указанием размера массива в квадратных скобках, либо без явного указания:

int test [4] = { 10, 20 , 30 ,40 };

char ch [ ] = { ‘a’ , ‘b’ , ‘c’ , ‘d’ };

Список инициализации помещается в фигурные скобки при определении массива.

В первом случае инициализация могла быть неполной, если бы в фигурных скобках находилось меньше значений, чем четыре.

Во втором случае инициализация должна быть полной, и количество элементов массива компилятор определяет по числу значений в списке инициализации.

Рассмотрим инициализацию многомерных числовых массивов. Массив любой мерности в С++ рассматривается как одномерный. Например, массив:
int B [3] [2] [4] – трактуется как одномерный массив из трех элементов - двумерных массивов (матриц) ***int [2] [4]***, с именами ***B[0], B[1], B[2],*** каждый из этих массивов трактуется как одномерный массив, состоящий из двух одномерных массивов (векторов) ***int [4]***.

Располагаются элементы массива в ОП естественно одномерно: сначала располагаются все элементы первой матрицы, как было описано выше, затем второй и третьей. При определении, например, трехмерного массива инициализация его элементов может проводиться либо с учетом внутренней структуры массива, отделяя двумерные и одномерные массивы фигурными скобками, либо списком значений в соответствии с расположением элементов в оперативной памяти:

```
int B [ 3 ] [ 2 ] [ 4] = { { {1, 2, 3, 4}, {5, 6, 7, 8} },
                           { {9, 10, 11, 12}, {13, 14, 15, 16} },
                           { {17, 18, 19, 20}, {21, 22, 23, 24} } };
```

эквивалентно:

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 21, 23, 24 };
```

Для инициализации символьных массивов чаще всего используются строки. Инициация символьного массива может быть выполнена значением строковой константы, например, следующим образом:

```
char stroka [10] = "строка";
```

При такой инициализации компилятор запишет в память символы строковой константы и добавит в конце двоичный ноль '\0', при этом памяти будет выделено с запасом (10 байт).

Можно объявлять символьный массив без указания его длины:

```
char stroka1 [ ] = "строка";
```

Компилятор, выделяя память для массива, определит его длину, которая в данном случае будет равна 7 байтов (6 байтов под собственно строку и один байт под завершающий нуль).

Инициализация двумерных символьных массивов проводится следующим образом:

```
char name [5] [18] = {"Иванов", "Петров", "Розенбаум", "Цой", "Григорьев"};
```

При объявлении символьного массива будет выделено памяти по 18 байтов на каждую строку, в которые будут записаны символы строки и в конце каждой строки добавлен байтовый ноль, всего 90 байтов. При таком объявлении в массиве остается много пустого места.

При определении многомерных массивов с инициализаций (в том числе и двумерных) значение первого индекса в квадратных скобках можно опускать.

Количество элементов массива по первому измерению компилятор определит из списка инициализации. Например, при определении:

char sh [ ] [40] = {''====================================='',
''| F | F | F | F | F |'',
''====================================''};

компилятор определит три строки массива по 40 элементов каждая, причем **sh[0], sh[1], sh[2] –** адреса этих строк массива в оперативной памяти. В каждую из строк будет записана строковая константа из списка инициализации.

Форма обращения к элементам массивов

С помощью операции [] (квадратные скобки) обеспечивается доступ к произвольному элементу массива.

Обращение к элементу одномерного массива имеет вид:

имя массива [индекс],

где ***индекс –*** это не номер элемента, а его ***смещение*** относительно первого элемента с индексом **0.** Пример:

int A[10]; A[5] = 0; //A[5] – обращение к шестому элементу массива.

Для обращения к элементам многомерного массива также используется имя массива, после которого в квадратных скобках стоит столько индексов, сколько измерений в массиве. Пример обращения к элементу двумерного массива: ***имя массива [i][j],*** это обращение к элементу ***i –й*** строки и ***j-го*** столбца двумерного массива.

2.6. Массивы и указатели

Имя массива воспринимается двояко на этапе определения массива и на этапе его использования.

С одной стороны ***имя массива*** следует рассматривать как имя структурированной переменной. И применение таких операций как ***sizeof*** и ***&*** к имени массива дают в качестве результата соответственно размер внутреннего представления в байтах всего массива и адрес первого элемента массива:

sizeof(имя массива) – размер массива;

&имя массива - адрес массива в целом, равный адресу первого элемента**.**

С другой стороны ***имя массива*** - это ***константный указатель,***

значением которого является адрес первого элемента массива и значение данного указателя нельзя изменять.

Рассмотрим одномерный массив. В соответствии с вышесказанным соблюдается равенство:

имя_массива = = &имя_массива = = &имя_массива[0],

то есть, имя массива отожествляется с адресом массива в целом и с адресом его первого элемента.

Данные соотношения позволяют сформулировать еще один способ обращения к элементам массива.

В соответствии с операцией сложения указателя с целым числом, если к имени массива (константному указателю) прибавить целое число ***i*:**

имя_массива +i ,

то на машинном уровне сформируется следующее значение адреса:

имя_массива + i * sizeof(тип элемента),

которое равно адресу ***i - го*** элемента массива:

имя_массива + i = = &имя_массива[i].

Операция разыменования адреса объекта предоставляет доступ к самому объекту. Применяя операцию разыменования для левой и правой части представленного выше уравнения, получаем результат:

****(имя_массива + i) = = имя_массива [i],***

из которого следует, что обращаться к ***i-му*** элементу массива можно любым из этих эквивалентных способов**.**

Многомерные массивы рассмотрим на примере объявления двумерного массива: ***type T [m][n]; m, n*** – целочисленные константы, ***type*** – тип элемента массива. В массиве ***m*** строк по ***n*** элементов в строке **(*n* столбцов).** Имя двумерного массива ***T*** также отожествляется с его адресом, а также с адресом его первого элемента ***T[0][0].***

Каждая строка двумерного массива - это одномерный массив с именем ***T[i],*** где ***i*** – индекс строки, и имя этого массива является адресом первого элемента строки и адресом строки в целом.

Адреса строк массива равны: ***T[0], T[1],…,T[m-1]***, и, как показано выше, эквивалентны следующим выражениям: ***T, *(T+1),…,*(T+m-1).***

Выражения: ***T[i] = = *(T+i) = =&T[i][0]*** – представляют адрес ***i*** –строки массива, и, следовательно, обращаться к элементу ***i –й*** строки ***j-го*** столбца массива можно следующими способами:

T[i][j] = = *(*(T+i)+j)==*(T[i]+j)==(*(T+i))[j]

Указатели и строки

Как и массивы типа ***char,*** указатели ***char**** могут инициализироваться строковыми константами:

char * имя_1 = "строка";

char * имя_2 = {"строка"};

char * имя_3("строка");

В операторе вывода по значению указателя выведется строка до байтового нуля:

cout << имя указателя на строку;

Если надо вывести адрес строковой константы, при выводе надо воспользоваться приведением типа указателя к ***void****:

cout << (void*) имя указателя на строку;

Если надо представлять в программе группу текстовых строк, целесообразно объявить массив указателей типа ***char**** по количеству строк:

char* name [5]= {"Иванов", "Петров", "Розен", "Цой", "Григорьев"};

Адреса отдельных строк равны значениям указателей из массива: ***name[0], name[1], name[2], name[3], name[4].***

По адресам выводим строки:

for (int i=0; i < 5; i++) cout << name[i] << endl;

Благодаря манипулятору ***endl*** фамилии выведутся в столбик.

Ввод/вывод элементов массивов

Ввод/вывод числовых массивов

Ввод/вывод значений арифметических массивов производится поэлементно.

Для одномерного массива следует организовать цикл (повторение обработки данных), в каждой итерации которого, изменять индекс элемента и производить ввод (вывод) значения соответствующего элемента. Пример иллюстрирует ввод элементов одномерного массива:

```
int test [100] ;
for (int i = 0; i < 100; i++) cin >> test [i];…
```

При вводе/выводе элементов двумерного массива для обращения к элементам необходимо устанавливать номера строк и столбцов элементов. При этом целесообразно учитывать, как элементы массива располагаются в оперативной памяти. Внешний цикл следует организовать по номерам строк. В теле этого цикла для каждого номера строки организовать внутренний цикл, в котором перебирать номера столбцов. Следуя такому алгоритму, обращение к элементам массива будет происходить в той последовательности, в которой они располагаются в оперативной памяти. Пример программы иллюстрирует вышесказанное:

```
#include < iostream.h>
#include< stdlib.h>
void main()
{ int T[3][4];
for( int i =0 ; i < 3; i++) { cout << '\n';
for( int j=0 ; j < 4; j++) { T[i][j] = rand( );  cout << T[i][j] << ''   '';} }
```

При вводе/выводе элементов многомерных массивов должно быть организовано столько циклов перебора индексов сколь мерный массив. Причем, самый внешний цикл – цикл перебора самого левого индекса, а в самом вложенном цикле варьируется самый правый индекс.

Ввод/вывод символьных массивов

Ввод и вывод символьных массивов можно производить поэлементно, как и числовых массивов, то есть рассматривать символьный массив как набор отдельных символов.

Синтаксис языка С++ допускает также обращение к символьному массиву целиком по его имени, а именно по адресу этого массива в

оперативной памяти. При этом также допускается обращение к отдельным элементам – символам по их индексу в массиве.

Объявим некоторый символьный массив:

char text [80];

Следующие операторы позволяют произвести ввод символьных строк:

1) ***cin>>text;*** **-** символы извлекаются из стандартного входного потока ***cin,*** и заносятся в оперативную память по адресу ***text,*** ввод начинается от первого символа, отличного от пробела до первого обобщенного пробела. В конце строки в память помещается двоичный ноль.

2) ***cin.getline(text, n);*** **-** извлекаются из стандартного входного потока ***cin*** любые символы, включая и пробелы, и заносятся в оперативную память по адресу ***text.*** Ввод происходит до наступления одного из событий: прочитан ***n-1*** символ или ранее появился символ перехода на новую строку ***'\n'***, (при вводе с клавиатуры это означает, что была нажата клавиша ***Enter*)**. В последнем случае из потока символ ***'\n'*** извлекается, но в память не помещается, а помещается в память символ конца строки ***'\0'***.

3) ***gets(text);*** - читается строка из стандартного потока (по умолчанию связанного с клавиатуры) и помещается по адресу ***text***. Вводятся все символы до символа перехода на новую строку ***'\n' (Enter)***, который в память не записывается, а в конце строки помещает двоичный ноль ***'\0'.***

4) ***scanf("%s", text);*** – из стандартного потока читается строка символов до очередного пробела и вводит в массив ***text***. В конце помещается байтовый ноль. Если строка формата имеет вид ***"%ns"***, то считывается ***n*** непробельных символов.

5) ***scanf("%nc", text);*** – из стандартного потока вводятся **n** любых символов, включая и пробелы, и символ конца строки.

Если стандартный входной поток связан с клавиатурой, все приведенные выше операторы, в основе которых лежат вызовы функций, останавливают программу до ввода строки символов.

Вывод строки позволяют произвести следующие операторы:

1) ***cout << text;*** **-** выводит всю строку до байтового нуля в стандартный выходной поток ***cout,*** по умолчанию связанный с экраном дисплея.

2) ***puts(text);*** **-** выводит строку в стандартный поток и добавляет в завершении символ **'\n'** – перехода на новую строку.

3) ***printf("%s" ,text);*** **-** выводит в стандартный выходной поток всю строку;

printf("%ws" ,text); **-** выводит всю строку в поле ***w,*** где ***w*** – целое число, количество текстовых позиций на экране для вывода символов. Если **w** больше числа символов в строке, то слева (по умолчанию) или справа (формат ***"%-ws"*)** от строки выводятся пробелы. Если ***w*** меньше, чем количество выводимых символов, то выводится вся строка.

printf("%w.ns" ,text); **-** выводит **n** символов строки в поле ***w;***

printf("%.ns" ,text); -выводит **n** символов строки в поле ***w = n;***

2.7. Создание динамических массивов

При определении массива ему выделяется память. При определении внешнего или статического массива память выделяется в ***сегменте данных,*** элементы массива по умолчанию инициализируются нулевыми значениями. При определении автоматического массива память выделяется в ***сегменте стека.***

В обоих случаях происходит ***статическое выделение памяти*** под массив либо в сегменте статических данных, либо в стеке. Это означает, что либо до конца программы, либо до конца блока массивам будут соответствовать участки памяти.

Можно выделять память под массив ***динамически*** в программе, с помощью соответствующих операторов и также программно освобождать ее по желанию программиста. Рассмотрим этот случай.

Объявим указатели (переменные) на объект типа ***type.*** Указатели можно связать с массивами - статическим (1) и динамическими (2, 3):

1) type * имя1 = имя уже определенного массива типа type;
2) type* имя2 = new type [количество элементов массива];
3) type * имя3 = (type *) malloc (количество элементов * sizeof(type));

Проиллюстрируем это на примерах:

int n [5] = { 1, 2, 3 , 4 ,5}; //определен статический массив
int*pn = n; //присоединение указателя к массиву статической памяти
//значение ***pn*** - адрес первого элемента массива
float*p1 = new float [10]; //динамически выделено **40 байт** памяти
//значение ***p1*** - адрес первого байта этого участка
double*p2 = (double*) malloc (4 * sizeof (double));
//динамически выделено **32** байта памяти, значение ***p2-*** адрес первого байта

После того как указатель связан с массивом доступ к элементам массива осуществляется следующими способами:

1) имя указателя [индекс элемента]
2) *(имя указателя + индекс элемента)
3) ***имя указателя*** ++(операция допустима, т.к. указатель – переменная)

Пример вывода элементов арифметического массива (производится поэлементно):

```
int A [10] = {7.3, 5.5, 1, … };
for (int i=0, *pA=A; i<10 ; i++)
cout<<*pA++<<'' '';      //что эквивалентно cout << pA[i], или cout << *(pA+i)
```

Массивы указателей

Как любые переменные указатели могут быть объединены в массивы.

Примеры массивов указателей статической памяти:

char*A [6] – одномерный массив указателей из 6 элементов – указателей на объекты типа ***char***, элементы массива **A** имеют тип ***char*.**

Выражение: ***(long)(A+1)- (long) A = 4 (байта)*** **–** дает внутренний размер указателя.

По общему правилу можно определять одномерные динамические массивы указателей, учитывая, что тип элемента массива равен ***type****:

1) type* * имя1 = new type* [количество указателей в массиве] ;

2) type* * имя2 = (type* *) malloc (количество элементов * sizeof(type*));

Создание двумерного динамического массива с помощью динамического массива указателей.

Двумерный массив составляется из одномерных массивов - строк массива, представляющих собой одномерные массивы, которые можно представить с помощью указателей.

Сколько строк в двумерном массиве (матрице) – столько надо указателей для их представления. Таким образом, надо объявить массив указателей на строки матрицы.

Массив указателей на строки матрицы тоже является одномерным массивом, который тоже можно определить динамически.

Элемент массива указателей имеет тип ***type*,*** где ***type*** **-** тип элементов исходной матрицы, поэтому массив указателей можно представить с помощью указателя на указатель ***type**.***

Пусть надо выделить память на динамическую матрицу элементов типа ***int***, размером: ***m***- строк по ***n*** элементов в строке (***m***-строк и ***n*** – столбцов).

```
int m, n;
cin >> m >> n;              //размеры матрицы m x n введем с клавиатуры
int ** W;                   //определим указатель на указатель
                            //для представления динамического массива указателей
W = new int*[m];            // выделяем память для массива указателей из m
//элементов типа int*, W[i] – указатель на i-ю строку динамической матрицы
//В цикле выделяем память на одномерные массивы из n элементов
for (int i =0; i< m; i++) W[i] = new int [n];
```

Память на динамическую матрицу (***m x n)*** выделена, к элементам которой можно обращаться с помощью стандартных выражений:

W[i] [j] **или** ****(*(W+i)+j)*;**

После работы можно освободить память, причем сначала надо освободить память, выделенную на элементы строк матрицы, а затем освободить память, выделенную на массив указателей на строки матрицы:

```
for (int i =0; i< m; i++) delete W[i];
delete[ ] W;
```

Указатель на массив. Многомерные массивы динамической памяти.

Указатель на массив – это переменная, значением которой является адрес массива. Этому указателю доступен участок памяти, выделенный под массив. При прибавлении к такому указателю 1 получается значение адреса, большее на размер массива.

Определение указателя на массив рассмотрим на примере определения указателя ***ptr*** на массив из 6 элементов типа ***int:***

int (*ptr) [6];

Выражение**:**

(long)(ptr+1) - (long) ptr == 12 (байтов)

возвращает размер массива в байтах, на который "указывает" указатель ***ptr.***

Многомерный динамический массив можно определить следующим образом: ***new тип массива,*** где тип массива – это тип элементов массива и нужное количество квадратных скобок, с константными выражениями, определяющие размеры массива.

При таком определении существуют следующие правила:

1) при выделении памяти для динамического массива его размеры должны быть полностью определены.
2) нельзя явно инициализировать массив при динамическом выделении памяти.

Определим трехмерный динамический массив типа ***int [3][2][4].*** Этот массив состоит из трех подмассивов - матриц ***2x4.*** Определим указатель на подмассив – матрицу, типа ***int*** размером ***2x4***: ***int (*p) [2] [4];***

С помощью операции **new** выделяется участок памяти для трех матриц, причем операция возвращает адрес первой матрицы массива***,*** который и присваивается указателю ***p***:

p = new int [3] [2] [4] – выделена память для массива ***3x2x4*** типа ***int***

Чтобы освободить память, выделенную на массив, используем оператор: ***delete [ ] p;***

Формы обращения к элементам массива:

p[i][j][k], либо ****(*(*(ptr1+i)+j)+k).***

В операторе:

cout << (long) (p+1) – (long) p; **//16**

- размер участка памяти, на который "указывает" указатель ***p.***

Определение типа массива имеет следующий вид:

typedef type имя_типа_массива [k1] [k2] ..[kn];

Пример: ***typedef float array [3][5][2];***

array – имя типа массива **3x5x2** с элементами типа ***float***

array Mas; **-** определен массив ***Mas*** типа ***array.***

Определение типа указателя на массив имеет следующий вид:

typedef type (*имя_типа_указателя_на_массив) [k1] [k2] ..[kn];

Пример: ***typedef float (*tpm) [5][2];***

tpm – имя типа указателя на массив **5x2** с элементами типа ***float***

tpm pm ; - объявлен указатель такого типа

Раздел 3. Функции

3.1. Определение, описание и вызовы функций

Программа на языке С++ представляет собой совокупность произвольного количества функций, одна (и единственная) из которых -

главная функция с именем ***main***. Выполнение программы начинается и заканчивается выполнением функции ***main.*** Выполнение неглавных функций инициируется в главной функции непосредственно или в других функциях, которые сами инициируются в главной.

Функции – это относительно самостоятельные фрагменты программы, оформленные особым образом и снабженные именем.

Каждая функция существует в программе в единственном экземпляре, в то время как обращаться к ней можно многократно из разных точек программы.

Упоминание имени функции в тексте программы называется ***вызовом функции***. При вызове функции активизируется последовательность образующих ее операторов, а с помощью передаваемых функции параметров осуществляется обмен данными между функцией и вызывающей ее программой.

По умолчанию все функции внешние (класс памяти ***extern***), доступны во всех файлах программы. При определении функции допускается класс памяти ***static,*** если надо, чтобы функция использовалась только в данном файле программы.

Определение функции

Определение функции – это программный текст функции. Определение функции может располагаться в любой части программы, кроме как внутри других функций. В Языке С++ нет вложенных функций.

Определение состоит из заголовка и тела функции:

<тип > <имя функции > (<список формальных параметров>)
тело функции

1) ***тип –*** тип возвращаемого функцией значения, с помощью оператора ***return,*** если функция не возвращает никакого значения, на место типа следует поместить слово ***void***;
2) ***имя функции –*** идентификатор, уникальный в программе;
3) ***список формальных параметров (сигнатура параметров*) –** заключенный в круглые скобки список спецификаций отдельных формальных параметров, перечисляемых через запятую:
   <тип параметра> <имя параметра>,
   <тип параметра> <имя параметра> = <умалчиваемое значение>,
   если параметры отсутствуют, в заголовке после имени функции должны стоять, либо пустые скобки **(),** либо скобки **– (*void*);**
   для формального параметра может быть задано, а может и отсутствовать умалчиваемое значение – начальное значение параметра;
4) ***тело функции –*** это блок или составной оператор, т.е. последовательность определений, описаний и операторов, заключенная в фигурные скобки.

Вызов функции

Вызов функции передает ей управление, а также фактические параметры при наличии в определении функции формальных параметров.

Форма вызова функции:

имя функции (список фактических параметров);

Список фактических параметров может быть пустым, если функция без параметров: ***имя функции ();***

Фактические параметры должны соответствовать формальным параметрам по количеству, типу, и по расположению параметров.

Если функция возвращает результат, то ее вызов представляет собой выражение с операцией "круглые скобки". Операндами служат имя функции и список фактических параметров. Значением выражения является возвращаемое функцией значение.

Если функция не возвращает результата (тип – ***void),*** вызов функции представляет собой оператор.

При вызове функции происходит передача фактических параметров из вызывающей программы в функцию, и именно эти параметры обрабатываются в теле функции вместо соответствующих формальных параметров.

После завершения выполнения всех операторов функция возвращает управление программой в точку вызова.

Описание функции (прототип)

При вызове функции формальные параметры заменяются фактическими, причем соблюдается строгое соответствие параметров по типам. В связи с этой особенностью языка С++ проверка соответствия типов формальных и фактических параметров выполняется на этапе компиляции.

Строгое согласование по типам между параметрами требует, чтобы в модуле программы до первого обращения к функции было помещено либо ее определение, либо ее описание (прототип), содержащее сведения о типе результата и о типах всех параметров.

Прототип (описание) функции может внешне почти полностью совпадать с заголовком определения функции:

<тип функции> <имя функции>
(<спецификация формальных параметров>);

Отличие описания от заголовка определения функции состоит в следующем:

- наличие '**;**' в конце описания – это основное отличие и
- необязательность имен параметров, достаточно через запятые перечислить типы параметров.

Переменные, доступные функции

1) локальные переменные:

- объявлены в теле функции, доступны только в теле функции;
- при определении переменной ей выделяется память в *сегменте стека*, при завершении выполнения функции память освобождается;

2) формальные параметры:

- объявлены в заголовке функции и доступны только функции;
- формальные параметры за исключением параметров – ссылок являются локальными переменными, память им выделяется в *стеке*;

- параметр – ссылка доступен только функции, но он не является переменной, на него не выделяется память, это некоторая абстракция для обозначения внешней по отношению к функции переменной;

3) глобальные переменные:

- переменные объявлены в программе как внешние, т.е. вне всех функций, включая и главную функцию ***main;***
- чтобы глобальная переменная была доступна функции, функция не должна содержать локальных переменных и формальных параметров с тем же именем; локальное имя "закрывает" глобальное и делает его не доступным;

Оператор return

Оператор ***return*** **-** оператор возврата ***управления программой и значения*** в точку вызова функции. С помощью этого оператора функция может вернуть одно скалярное значение любого типа. Форма оператора:

return (выражение);

- выражение определяет значение, возвращаемое функцией; выражение вычисляется, результат преобразуется к типу возвращаемого значения и передается в точку вызова функции;
- если функция не возвращает результата, оператор может, либо отсутствовать, либо присутствовать с пустым выражением: ***return;***
- функция может иметь несколько операторов ***return*** с различными выражениями, если алгоритм функции предусматривает разветвление.

Функция завершается, как только встречается оператор ***return***. Если функция не возвращает результата, и не имеет оператора ***return,*** она завершается по окончанию тела функции.

Формальные и фактические параметры функции

Посредством формальных параметров осуществляется обмен данными между функцией и вызывающей ее программой. В функцию данные передаются для обработки. Функция, обработав эти данные, может вернуть в вызывающую функцию результат обработки.

В определении функции фигурируют формальные параметры, которые показывают, какие данные следует передавать в функцию при ее вызове, и как они будут обрабатываться операторами функции.

Список формальных параметров (список аргументов) функции указывает, с какими фактическими параметрами следует вызывать функцию.

Фактические параметры передаются в функцию при ее вызове, заменяя формальные параметры. Фактические параметры, по количеству, по типу (он должен быть идентичным или совместимым), по расположению должны соответствовать формальным параметрам.

Умалчиваемые значения параметров

Формальный параметр может содержать умалчиваемое значение. В этом случае при вызове функции соответствующий фактический параметр может быть опущен и умалчиваемое значение используется в качестве фактического параметра. При задании умалчиваемых значений должно соблюдаться правило.

Если параметр имеет умалчиваемое значение, то все параметры справа от него также должны иметь умалчиваемые значения.

Передача фактических параметров

В С++ передача параметров может осуществляться тремя способами:

− по значению, когда в функцию передается числовое значение параметра;

− по адресу, когда в функцию передается не значение параметра, а его адрес, что особенно удобно для передачи в качестве параметров массивов;

− по ссылке, когда в функцию передается не числовое значение параметра, а сам параметр и тем самым обеспечивается доступ из тела функции к объекту, являющемуся фактическим параметром.

Передача параметров по значению

Формальным параметром может быть только имя скалярной переменной стандартного типа или имя структуры, определенной пользователем.

При вызове функции формальному параметру выделяется память в стеке, в соответствии с его типом. Фактическим параметром является – выражение, *значение* которого копируется в стек, в область ОП, выделенную под формальный параметр. Фактическим параметром может быть просто неименованная константа нужного типа, или имя некоторой переменной, значение которой будет использовано как фактический параметр.

Все изменения, происходящие с формальным параметром в теле функции, не передаются переменной, *значение* которой являлось фактическим параметром функции.

Передача параметров по адресу - по указателю

Формальным параметром является указатель *type*p* на переменную типа *type*. При вызове функции формальному параметру-указателю выделяется память в стеке - 2 байта, если указатель ближний и 4 байта, если указатель дальний. Фактическим параметром может быть либо адрес в ОП переменной типа *type***,** либо значение другого указателя, типа *type** из вызывающей программы. В область памяти (в стеке), выделенную для указателя *p* будет копироваться значение некоторого *адреса* из вызывающей функции.

В теле функции, используя операцию разыменования указателя **p***,* можно получить доступ к участку памяти, адрес которого получил *p* при вызове функции, и изменить содержимое этого участка**.**

Если в теле функции изменяется значение **p***,* при вызове функции эти изменения произойдут с тем объектом вызывающей программы, адрес которого использовался в качестве фактического параметра.

Передача параметров по ссылке

В языке С++ ссылка определена как другое имя уже существующей переменной. При определении ссылки оперативная память не выделяется Инициализатор, являющийся обязательным атрибутом определения ссылки, представляет собой имя переменной того же типа, имеющей место в памяти. Ссылка становится синонимом этой переменной.

*type & имя ссылки = имя переменной***;**

Основные достоинства ссылок проявляются при работе с функциями.

Если определить ссылку *type&a* и не инициализировать ее, то это равносильно созданию объекта, который имеет имя, но не связан ни с каким участком памяти. Это является ошибкой.

Однако такое определение допускается в спецификациях формальных параметров функций. Если в качестве формального параметра функции была определена неинициализированная ссылка - некоторая абстрактная переменная, которая имеет имя, но не имеет адреса, в качестве фактического параметра при вызове функции следует использовать имя переменной из вызывающей программы того же типа, что и ссылка. Эта переменная инициализирует ссылку, т. е. связывает ссылку с участком памяти.

Таким образом, ссылка обеспечивает доступ из функции непосредственно к внешнему участку памяти той переменной, которая при вызове функции будет использоваться в качестве фактического параметра.

Все изменения, происходящие в функции со ссылкой, будут происходить непосредственно с переменной, являющейся фактическим параметром.

Это наиболее перспективный метод передачи параметров, так как в этом случае вообще не происходит копирование фактического параметра в стек, будь то значение или адрес, функция непосредственно оперирует с внешними по отношению к ней переменными, используемыми в качестве фактических параметров.

3.2. Классификация формальных параметров

Формальный параметр скаляр, указатель на скаляр, ссылка на скаляр.

В качестве скаляра будут рассматриваться переменные основных типов, элементы массивов или структур.

Если формальными параметрами функции являются:

а) скаляры;

б) указатели на скаляры;

в) ссылки на скаляры;

то фактическими параметрами должны быть:

а) значения скаляров;

б) адреса скаляров;

в) имена скаляров (имена участков памяти).

Формальные параметры – массивы

Массив в качестве фактического параметра может быть передан в функцию только по адресу, то есть с использованием указателя.

I) Формальный параметр- определение массива:

В качестве формальных параметров можно использовать:

1) определение массива с фиксированными границами, например:
   float A[5], int B[3][4][2], char S [25];
2) определение одномерного символьного массива с открытой границей:

char S1[];

При работе со строками, то есть с одномерными массивами данных типа ***char,*** последний элемент которых имеет известное значение - ***'\0',*** нет необходимости передавать размеры массива;

3) определение числового или многомерного символьного массива с открытой левой границей и параметр для передачи размера левой границы:

float A[ ], int B[ ][4][2], char S2 [ ][60], и **int n**

При всех этих определениях в стеке выделяется оперативная память на один указатель для передачи в функцию адреса нулевого элемента массива – фактического параметра.

Массив, адрес которого будет использован при вызове функции, как фактический параметр может быть изменен за счет выполнения операторов функции.

Если формальный параметр - массив с фиксированными границами как в первом случае, фактическим параметром будет имя массива из вызывающей программы с теми же фиксированными границами или же значение указателя на такой фиксированный массив.

При использовании второго определения фактическим параметром будет имя символьного массива, или указателя на первый элемент массива.

При использовании третьего определения фактическими параметрами будут имя массива из вызывающей программы, но уже с произвольным количеством элементов по первому измерению и размер массива по первому измерению.

Проиллюстрируем сказанное на примере определения функции, заполняющей трехмерный массив – параметр последовательными натуральными числами, начиная с единицы и возвращающей по ссылке сумму его элементов:

```
const int n=4, k =5;
float f ( float a [ ] [n][k] , int m, float & s )
{s = 0; int t =1;
for ( int i=0; i< m; i++) for( int j=0; i< n; j++) for ( int l=0; l< k; l++)
  { a[i][j][l]=t++;  s+= a[i][j][l];}}
void main ( )
{ float s, dat [3][4][5];
f(dat, 3, s); //вызов функции заполняющей массив dat
cout << s;}
```

II) Формальный параметр - указатель:

1) В качестве формального параметра определяется указатель на первый элемент массива любой мерности и второй параметр – общее количество элементов в массиве:

type*p, int n.

Фактическими параметрами в этом случае будут – указатель типа ***type*,*** значение которого - адрес первого элемента массива и второй параметр - значение общего количества элементов в массиве. При этом надо помнить, что имя массива любой мерности – это константный указатель, значением которого является адрес первого элемента массива, однако только для одномерного массива имя – есть указатель на элемент массива, для двумерного массива имя массива – это указатель на строку массива и так далее. Чтобы получить указатель на элемент массива для двумерного массива, надо разыменовать имя массива, для трехмерного массива – два раза разыменовать имя массива и так далее. Или решать проблему явным приведением типов указателей.

В качестве примера определим функцию, формирующую упорядоченный массив из элементов двух других массивов. Все три массива представим с помощью указателей на первый элемент массива. Используем еще два параметра для указания размеров исходных массивов. Размер результирующего массива является суммой количества элементов двух массивов.

```
#include<iostream.h>
void f (int*a, int*b, int*c, int n, int m)
{int i, j, p;
  for ( i=0; i < n+m; i++)          //формирование неупорядоченного массива
    if (i < n) c[i] = a[i];else c[i] = b[i-n];
  //упорядочивание массива методом пузырькового всплытия
  for (i = 0; i < n+m-1; i++)
    for (j = n+m - 1; j > i; j -- )
      if (c[j] < c[j-1]){p= c[j]; c[j] = c[j-1]; c[j-1] = p;}
}
int i, a [ ] = {7,9,5,4,0,2,89,33,73,11},
b [ ] = { 23,87,55,45,4,3,0,6,7,3},
n= sizeof( a ) / sizeof ( a[0]),
m= sizeof( b ) / sizeof ( b[0]);
//а)создаваемый массив – статический:
void main ( ){
int c [sizeof( a ) / sizeof ( a[0])+ sizeof( b ) / sizeof (b[0]) ];
// - выделена память на статический массив
f (a, b, c, n, m);
for(i = 0; i < n+m; i++)
cout << c[i] << " ";
}
//б) создается динамический массив:
void main ( )
{int*x = new int [n+m];// - выделена память на динамический массив
f (a , b , x , n , m ) ;
for ( i =0; i < n+m; i++)
cout << x[i] << " ";
```

```
delete [ ] x;
}
```

2) Формальные параметры - указатели на двумерный массив, размеры которого заранее не известны.

Параметрами являются указатель на массив указателей на строки матрицы: ***type**p*** и два значения целого типа: ***int m*** - количество этих строк и ***int n*** - количество элементов в каждой строке. В этом случае динамически выделяется память и на массив указателей на строки матрицы и на числа в этих строках, то есть производится полностью динамическое выделение памяти на двумерный массив.

В качестве примера определим функцию, заполняющую массив – параметр случайными числами:

```
void mas ( int** ptr, int m, int n)
{randomize( );
for (int i =0; i < m; i++)
for ( int j = 0; j < n; j++)
ptr[i][j] = rand( );
}
void main ( )
{float **ptr = new float* [m]; // динамическое выделение памяти
                               // на массив указателей на строки массива
for( i=0; i< m; i++ )
ptr[i] = new float [n]; //динамическое выделение памяти на строки массива
mas (ptr,5,6);        //вызов функции
for (i=0; i< m; i++) delete ptr[i]; //освобождение памяти на строки
delete [ ] ptr;       //освобождение памяти на массив указателей на строки
}
```

3) Формальные параметры указатели - для передачи многомерного массива. Параметрами являются указатель на подмассив и количество подмассивов в многомерном массиве. При этом размеры подмассива должны быть фиксированы.

<type> (* имя_указателя) [N1][N2] [N3]…[NN]; - определение указателя на массив размерностью ***[N1][N2]…[NN]*** с элементами ***type*.**

Для иллюстрации определим функцию, заполняющую натуральными числами трехмерный массив, состоящий из произвольного числа матриц **3x4**:

```
void mas (int (*lp) [3][4], int n)
{ int i, j, k, t=1;
for (i=0; i < n; i++) {cout << "\n\n";
    for (j=0; j < 3; j++) {cout << "\n";
        for (k=0; k< 4; k++) { lp[i][j][k] = t++;cout <<lp [i][j][k] << " "; } } } }
void main ()
{ int i, j, k, n; cin >> n; //вводится количество матриц в трехмерном массиве
int (*tp) [3][4];           //определен указатель на матрицу
```

```
tp = new  int [ n ] [3][4];
/* выделена динамическая память на n матриц, и адрес первой матрицы
присвоен указателю tp */
mas (tp, n);          //далее можно работать с элементами массива tp[i][j][k]
delete [ ] tp;        //освобождение памяти
}
```

3.3. Результат функции, возвращаемый с помощью оператора return

Оператор **return -** оператор возврата ***управления программой и значения*** в точку вызова функции. С помощью этого оператора функция может вернуть одно скалярное значение любого типа. Форма оператора:

return (выражение);

выражение вычисляется, результат преобразуется к типу возвращаемого значения и передается в точку вызова функции.

К скалярам относятся данные простых стандартных типов, указатели и ссылки. Рассмотрим некоторые возможные результаты работы функций, возвращаемые с помощью оператора ***return*:**

1) скалярное значение любого простого типа, например ***return (5);***
2) указатель на скаляр, массив;
3) ссылка - возвращаемый результат функции

Результат указатель на скаляр, массив

Функция возвращает указатель на скаляр **-** возвращает адрес скаляра. Определим функцию поиска числа в произвольном массиве. Функция возвращает адрес числа или NULL, то есть указатель на элемент массива.

```
int * poisk ( int*c, int m, int n )
/* *c – указатель на первый элемент массива, m – количество элементов
массива , n- искомое число*/
{for (int i =0; i < m; i++ ){ if (*c = = n) return ( c );c++;}
return (NULL);}
void main ( )
{ int a[5][4] = {…}, i, j;
int*p = poisk (*a, 20 ,7 );  //*a-адрес первого элемента, 20 - общее количество
                             //элементов, ищем число 7
if (p== NULL) cout << "число отсутствует";
else { cout<<(i= (p - *a ) / 4);  //номер строки
cout<<(j = p - *a – i * 4); };     //номер столбца
}
```

Результат функции – указатель на одномерный массив.

Используя указатель на элемент массива, можно в теле функции выделить память на одномерный динамический массив.

Этот указатель можно возвратить из функции с помощью оператора **return,** и тем самым возвратить "указатель на одномерный массив**".**

Определим функцию, формирующую одномерный динамический массив, заполняя его значениями, вводимыми с клавиатуры. Формальный параметр функции – количество элементов в формируемом массиве. Функция возвращает указатель на первый элемент массива:

```
float* vvod ( int n)
{float*p = new int [n];          // динамическое выделение памяти под массив,
for( int i=0; i < n; i+ ) cin >> p[i];          //заполнение элементов массива
return p;}          //возвращение массива;
void main ( )
{float*dat, n;      cin >> n;     // вводится с клавиатуры количество элементов;
dat = vvod (n);          //указателю присваивается результат вызова функции;
for(int i=0; i < n; i++)    // поэлементный вывод на экран элементов массива
cout << dat[i] << "\t";
delete dat;}          // освобождение памяти;
```

Функция возвращает указатель на указатель на элемент массива

С помощью указателя на указатель на элемент массива в теле функции может быть сформирован двумерный динамический массив. Этот указатель на указатель можно быть возвращен как результат работы функции с помощью оператора ***return,*** и тем самым возвращен "указатель на двумерный массив"**.**

В качестве иллюстрации определим функцию, возвращающую двумерный массив натуральных чисел, размеры массива передаются в функцию посредством параметров:

```
int** mas (int m, int n)
{int k=1;
int ** ptr= new int* [m ];  //выделение памяти на массив из m указателей на int;
for (int i=0; i < m; i++)
ptr [i] = new int [ n];  //выделение памяти, для каждой строки из n элементов
for (int i =0; i < m; i++)  //заполнение массива и вывод его элементов;
{cout<<"\n"; for (int j = 0; j < n; j++) { ptr[ i] [j] = k++ ;cout<<ptr [i][j]<<" ";} }
return ptr;}          // возвращается "указатель на матрицу"
void main ( )
{int n , m ,i ,j;      cin>> n>>m;        // размеры массива вводятся с клавиатуры;
int ** Q = mas (m, n );  //указателю присваивается результат вызова функции;
…//Q[i][j] - обращение к элементам динамического двумерного массива
//освобождение памяти
for (i=0; i < m; i++)      delete Q[i];
delete [ ] Q; }
```

Результат функции - указатель на подмассив. Определим в качестве примера функцию, возвращающей значение указателя на подмассив (**4x5)** элементов типа **int**, которое является адресом первой матрицы трехмерного массива, сформированного в функции.

```
#include <iostream.h>
typedef int (*TPM)[4][5];//TPM - тип указателя на матрицу 4x5
```

```
TPM fun (int n)    //параметр функции – количество подмассивов в массиве
{int i, j, k, t=1;
TPM lp=new int [n][4][5]; // выделяем память на трехмерный массив
for (i=0; i < n; i++){cout << "\n\n";
    for (j=0; j < 4; j++){ cout << "\n";
        for (k=0; k< 5; k++){lp[i][j][k]=t++; cout <<lp [i][j][k] << " ";} } }
return lp;}
void main ()
{ int n, i, j , k; cin>>n;
TPM D=fun(n); //вызов функции
//D[i][j][k] - обращение к элементам трехмерного массива
delete []D; //освобождение памяти
}
```

Ссылка - возвращаемый результат функции

Ссылки не являются настоящими объектами, ссылка связана с участком памяти инициализирующего ее выражения.

Если функция возвращает ссылку, это означает, что функция должна возвращать не значение, а идентификатор для обращения к некоторому участку памяти, в простейшем случае имя переменной. Таким образом, в функции должен быть, например, такой оператор: ***return имя переменной;***

При этом следует помнить, что в операторе ***return*** не должно стоять имя локальной переменной, так как после вызова функции участки памяти, связанные в сегменте стека с локальными переменными становятся недоступными. Таким образом, в операторе должно стоять имя участка памяти из внешней программы.

Вызов такой функции представляет собой частный случай ***l-значения (l-value)***, которое представляет в программе некоторый участок памяти. Этот участок памяти может иметь некоторое значение и это значение можно изменять, например, в операторе присваивания.

В соответствие с этим, вызов такой функции может располагаться как в правой, так и в левой части оператора присваивания.

Следующая функция, возвращающая ссылку на элемент массива с максимальным значением, проиллюстрирует вышесказанное. Массив передается в функцию посредством параметра.

```
int& rmax (int d [ ], int n)
{int imax = 0;
for (int i =1; i< n; i++)
imax = (d[imax] > d[i] ? imax : i);
return (d[imax]);}
void main ()
{ int n =5, a [ ] = { 3, 7 , 21 , 33 , 6};
cout << rmax (n, a )<<endl;
rmax(n,a)=0;
```

```
for ( int i =0 ; i <n ; i++)
cout << a[i] << " " ;}
```

Результат программы:

```
33
3 7 21 0 6
```

Один из вызовов функции **rmax()** находится в левой части оператора присваивания, что позволяет занести в элемент новое значение.

3.4 Указатели на функции

Имя функции является указателем-константой на эту функцию. Значением этого указателя является адрес размещения операторов функции в оперативной памяти. Это значение адреса можно присвоить другому указателю-переменной на функцию с тем же типом результата и с той же сигнатурой параметров. И затем этот новый указатель можно применять для вызова функции.

Введем понятие указателя на функцию. Указатель на функцию – это некоторая переменная, значениями которой являются адреса функций (имена функций), характеристики которых (тип результата и сигнатура параметров) совпадают с характеристиками, указанными в определении указателя.

Определение указателя на функцию:

<тип_рез_ф> (* имя указателя) (спецификация_парам) =
< имя инициирующей функции>;

- при определении достаточно перечислить через запятую типы параметров, имена параметров можно опустить;
- ***тип_рез_ф*** – это тип результата, возвращаемого функцией;
- инициализация указателя не обязательна, но при ее наличии тип результата, и сигнатура параметров инициирующей функции должны быть такими же, как в определении указателя.

Например, определены два указателя:

int* (*fptr) (char*, int); int (*ptr) (char*);

В примере указатели были определены без инициализации, но в дальнейшем этим указателям – переменным можно присвоить значения указателей – констант, а именно идентификаторы (имена) конкретных функций, спецификации которых должны полностью соответствовать спецификациям в определениях указателей.

Как только некоторому указателю присвоено имя функции, вызов этой функции можно производить, как используя имя функции, так и используя имя указателя на функцию.

Эквивалентные вызовы функции с помощью указателя на эту функцию:

имя указателя (список фактических параметров);
(*имя указателя) (список фактических параметров);

Рассмотрим на примере использование указателя на функцию. Определим функцию вычисления длины строки - параметр функции (количества символов в строке до байтового нуля, строка):

```
int len (char* e)
{int m=0;    while (e[m++]);return (m-1);}
void main ()
{int (*ptr) (char*);          //объявлен указатель на функцию без инициализации;
ptr = len;                    //указателю присвоено значение – имя функции len;
char s [ ] = "rtgcerygw";
int n = ptr(s);}
```

Массивы указателей на функции

Указатели на функции могут быть объединены в массивы.

Ниже дано определение массива указателей на функции, возвращающие значение типа ***float*** и имеющие два параметра типа ***char*** и ***int***. В массиве с именем ***ptrArray*** четыре таких указателя:

```
float ( *ptrArray ) ( char, int ) [4];
```

Эквивалентное определение массива:

```
float ( *ptrArray [ 4 ] ) ( char, int );
```

При объявлении массива можно провести инициализацию элементов массива указателей на функции именами соответствующих функций.

Пример определения массива указателей с инициализацией:

```
float v1 ( char s, int n) {…}…
float v4 ( char s, int n) {…}
float (*ptrr [4]) (char, int) = { v1, v2, v3, v4 };
```

В рассматриваемом примере даны определения четырех однотипных функций ***v1,…v4*** и определен массив ***ptrr*** из четырех указателей, которые инициализированы именами функций ***v1,…v4.***

Для того чтобы обратиться, например, к третьей из этих функции, можно использовать такие операторы:

```
float x = (*ptrr [2]) (‘a’ , 5);
float x = ptrr [2] (‘a’ , 5);
```

Для удобства последующих применений целесообразно вводить имя типа указателя на функцию с помощью спецификатора ***typedef:***

```
typedef <тип_функции> (*имя типа_указателя)
                                   (спецификация_параметров);
```

Массивы указателей на функции удобно использовать при разработке программ, управление которыми выполняется с помощью меню, реализующих вызов различных функций обработки данных в интерактивном режиме.

Рассмотрим алгоритм программы простейшего меню:

- варианты обработки данных определяются в виде функций **art1() – act4();**
- объявляется тип **menu** - тип указателя на такие функции;

- объявляется массив **act** из четырех указателей на функции, инициированный именами функций **art1() – act4();**

***Интерактивная часть*:**

- на экран выводятся строки описания вариантов обработки данных и соответствующие вариантам целочисленные номера;
- пользователю предлагается выбрать из меню нужный ему пункт и ввести значение номера, соответствующее требуемому варианту обработки;
- пользователь вводит значение номера с клавиатуры;
- по номеру пункта, как по индексу, из массива указателей выбирается соответствующий элемент, инициированный адресом нужной функции обработки; производится вызов функции.

Использование массива указателей существенно упрощает программу, так как в данном случае отпадает необходимость использовать оператор **switch** – для выбора варианта.

Ниже приведена программа:

```
# include < iostream.h>
# include < stdlib.h>
// определение функций обработки данных:
void act1 ( ) { cout << "чтение файла"; }
void act2 ( ) { cout << "модификация файла"; }
void act3 ( ) { cout << "дополнение файла"; }
void act4( ) { cout << "удаление записей файла"; }
typedef void ( * menu) ();        // дано описание типа указателя на функции;
menu act [4] = { act1, act2 , act3 , act 4};    //определен массив указателей;
void main ()
{int n;
cout << "\n1 - чтение файла";
cout << "\n2 - модификация файла";
cout << "\n3 - дополнение файла";
cout << "\n4 - удаление записей файла";
while (1) { cout << "\n введите номер"; cin >>n;
if ( n >= 1 && i<= 4) act [n-1] ( ); else exit(0);}}
```

Указатель на функцию - параметр функции

Указатели на функции удобно использовать в качестве параметров функций, когда объектами обработки функций должны служить другие функции.

В С++ все функции внешние, любую определенную функцию можно вызывать в теле любой другой функции непосредственно по имени, не передавая имя функции через механизм параметров.

Указатели на функции как параметры функций целесообразно использовать, когда в создаваемой функции должна быть заложена возможность обработки не конкретной, а произвольной функции. В этом случае адрес обрабатываемой функции целесообразно передавать в функцию

посредством параметра. В качестве формального параметра следует объявить указатель на функцию, а при вызове функции передавать в качестве фактического параметра идентификатор (адрес) нужной обрабатываемой функции.

Указатели на функции в качестве формальных параметров можно, например, использовать в функциях:

1) формирования таблиц результатов, получаемых с помощью различных функций (формул);
2) вычисления интегралов с различными подынтегральными функциями;
3) нахождения сумм рядов с различными общими членами и т. д.

Приведем пример: определить и вызвать функцию ***table()*** для построения таблицы значений различных функций. Функция ***table()*** использует в качестве параметров указатели на функции, которые определяют функции для вычислений значений в таблице.

Алгоритм задания:

- определяются три однотипных функции с одним вещественным параметром ***(a(x), b(x), c(x))*** для расчета значений, выводимых в таблицу;
- объявляется тип указателя ***func*** на такие функции;
- определяется массив ***S*** из трех указателей на функции инициированный именами функций ***a, b, c***;
- определяется функция ***table,*** выводящая в виде таблицы значения трех функций передаваемых **в** ***table*** посредством параметров; аргументами функции ***table*** являются:
  во-первых, массив ***ptrA*** указателей на функции с открытыми границами для передачи функций, вычисляющих значения и целочисленный параметр ***n*** для передачи количества указателей в массиве;
  и, во-вторых, параметры для аргумента функций – начальное значение - ***xn***, конечное значение - ***xk*** и шаг изменения аргумента - ***dx;***
- в главной функции производится вызов функции ***table()*** и передаются фактические параметры – инициированный конкретными функциями массив ***S***, количество указателей в массиве - **3** и значения аргумента – начальное, конечное и шаг изменения аргумента.

Алгоритм функции table:

- устанавливается начальное значение аргумента функций ***x=xn;***
- пока аргумент функций не достигнет своего конечного значения ***(x<= xk)*** выполняется повторяющаяся обработка: при каждом значении аргумента выводится строка значений трех функций, вызовы которых производятся с использованием указателей на функции из массива указателей и затем значение аргумента увеличивается на величину ***dx***.

Текст программы:

```
# include <iostream.h>
float a ( float x) { return x*x }
float b ( float x) { return (x*x +100) }
```

```
float c ( float x) { return sqrt ( fabs(x)) +x;}
typedef float (* func) ( float x) ;
func S [3] = {a, b, c}
```
//определение функции таблицы
```
void  table ( func ptrA [ ] , int n, float xn , float xk , float dx )
{float x = xn;
while ( x<= xk )
{cout << "\n";
for (int i=0; i< n; i++)
  { cout. width(10); cout << (* ptrA[i] ) (x);}
x+=dx ;} }
void main
{ table ( S, 3, 0., 2., 0.1 );}
```

Указатель на функцию может быть результатом работы функции, то есть функция может возвращать указатель на функцию с помощью оператора ***return*** и с помощью формального параметра (например, передача параметра по ссылке).

Рассмотрим это на примере:

```
typedef void ( * menu) ( )
menu act [4] = { act1, act2 , act3 , act 4};
menu F1 (int i) { return act [i] }
void F2 (int i , menu & r) { r = act [i] }
void main ( )
{ int i;        menu p1 , p2;
while(1)      // бесконечный цикл;
{ cin >> i;
if ( i>=1 && i<=4 )
{p1 = F1 (i-1);      p1( );          // вызов функции;
F2(i-1, p2);p2 ( );          // вызов функции;
} else exit ( 0); } }
```

Ссылка на функцию

Подобно указателю на функцию определяется и ссылка на функцию:

<тип_функции> (& имя ссылки) (спецификация_параметров)
<инициализирующее_выражение>;

здесь:

- ***<тип_функции>*** - тип возвращаемого функцией значения;
- ***спецификация_параметров*** – определяет сигнатуру функций, ***<инициализирующее_выражение>*** - обязательный элемент и является именем уже известной функции, имеющей тот же тип и ту же сигнатуру параметров, что и ссылка.

Ссылка на функцию является *синонимом (псевдонимом)* имени функции, обладает всеми правами основного имени функции.

3.5. Рекурсивные функции

Рекурсия – это способ организации обработки данных в функции, когда функция обращается сама к себе прямо или косвенно.

Функция называется косвенно рекурсивной, если она содержит обращение к другой функции, которая содержит прямой или косвенный вызов определяемой (первой) функции.

Если в теле функции явно используется вызов этой функции, то имеет место прямая рекурсия.

Рекурсивная форма алгоритма дает более компактный текст программы, но требует дополнительных затрат оперативной памяти для размещения данных и времени для рекурсивных вызовов функции.

Рекурсивный алгоритм позволяет повторять операторы тела функции многократно, каждый раз с новыми параметрами, конкурируя тем самым с итерационными методами (циклами). И также как и в итерационных методах, алгоритм должен включать условие для завершения повторения обработки данных, то есть иметь ветвь решения задачи без рекурсивного вызова функции.

Как было сказано выше, использование рекурсии не дает никакого практического выигрыша в программной реализации, и ее следует избегать, когда есть очевидное итерационное решение.

При выполнении рекурсивной функции происходит многократный ее вызов, при этом

- в стеке сохраняются значения всех локальных переменных и параметров функции для всех предыдущих вызовов, выделяется память для локальных переменных очередного вызова;
- переменным с классом памяти **extern** и **static** память выделяется один раз, которая сохраняется в течение всего времени программы;
- вызов рекурсивной функции происходит до тех пор, пока не будет получено конкретное значение без рекурсивного вызова функции.

Приведем примеры:

1) Определить функцию, возвращающую значение факториала целого числа. Факториал определятся только для положительных чисел формулой: $N!=1\cdot 2\cdot 3\cdot 4\cdot \ldots \cdot N$, и факториал нуля равен 1 (**0! =1).**

Определение факториала можно переписать в виде "математической рекурсии": $N!=N\cdot(N-1)!$**,** то есть факториал вычисляется через факториал. Соответственно определение функции имеет вид:

```
long fact ( int k)
{if ( k <0 ) return 0 ;
if ( k = = 0 ) return 1;            // здесь рекурсия прерывается;
return k * fact ( k- 1);}
```

2) Определить функцию, возвращающую целую степень вещественного числа. Определение **n**-й степени числа **X** можно представить в виде:

$X^n = X * X^{n-1}$ при **n >0,**
$X^n = X^{n+1} / X$ при **n< 0,**
и в соответствии с этим определить рекурсивную функцию:

```
double step ( double X , int n )
{ if (n = = 0) return 1;     // здесь рекурсия прерывается;
if (X = = 0) return 0;
if (n > 0 ) return X * step(X , n-1 );
if (n < 0 ) return step(X , n+1 ) / X;}
```

3) Определить функцию, возвращающую сумму элементов массива. Запишем формулу для суммы **n** элементов в виде суммы последнего элемента и суммы первых **n-1** элементов**:**

$S_n = a[n-1] + S_{n-1}$ **,**

то есть сумма вычисляется через сумму. Соответственное определение рекурсивной функции:

```
int sum (int a[] , int n )
{ if (n = = 1) return ( a[0] );     // здесь рекурсия прерывается;
else return ( a[n-1] + sum (a, n-1) );}
```

3.6. Перегрузка функций. Шаблоны функций

Перегрузка функций

С++ позволяет определить в программе произвольное количество функций с одним именем, при условии, что все они имеют разный состав параметров, или *сигнатуру:*

```
void F (int);
void F (int, int);
void F (char*);
```

При вызове перегруженной функции компилятор анализирует ее сигнатуру и выбирает из списка одноименных функций ту функцию, сигнатура которой соответствует вызываемой.

При этом возвращаемый функцией тип значения не имеет, функции:

```
void F ( int),
int F (int)
```

не являются перегруженными, компилятор их не различает.

Для того чтобы различать одноименные функции компилятор использует кодирование имен функций (создает уточненные имена). Функциям даются имена, в которые входят имя класса, если эта функция компонентная, имя функции и список обозначений типов параметров.

```
class MyClass {…
void Func (int);              //@MyClass@Func$qi
void Func (int, int);         //@MyClass@Func$qii
void Func (char*);            //@ MyClass@Func$qpz
…};
```

Пример перегрузки глобальных функций:

```
#include<iostream.h>
void Print ( int i) { cout << "\n int =  " <<i;}
void Print ( int*pi ) { cout<<"\n pointer = " << pi << ", значение = " << *pi;}
void Print ( char*s) { cout << "\n string = " <<s;}
void main ()
{ int a=2;
int*b = &a;
char*c = "yyyy";
char d [] = "xxxx";
Print(a);     Print(b);     Print(c);     Print (d);}
Результат:
int=2
pointer =0x12340ffe , значение = 2
string = yyyy
string = xxxx
```

Перегрузка функций используется, как правило, когда отличаются в перегружаемых функциях и типы данных обрабатываемые функциями и отличается сам код обработки, но характер обработки имеет один и тот же смысл и целесообразно не придумывать разные названия функциям, а перегрузить функции для разных данных.

Если код функций совпадает, но отличаются обрабатываемые данные, то создается шаблон функций.

Шаблоны функций

Шаблон функций – это автоматизация создания функций, которые могут обрабатывать данные различных типов.

Шаблон семейства функций определяется один раз, но в это определение в качестве параметра, или параметров входят типы данных обрабатываемых функцией и возвращаемых ею. *Таким образом, шаблон обязан иметь параметры – типы данных.*

Шаблон состоит из двух частей – заголовка шаблона:

template <список параметров шаблона>
(<class имя_параметра1, class имя_параметра2, …>)
и определения функции, в котором используются параметры шаблона.

Параметры шаблона должны отвечать следующим требованиям:

1) имена параметров шаблона должны быть уникальными в определении шаблона;
2) список параметров не может быть пустым;
3) может быть несколько параметров

template<class T1,classT2,classT3>

все имена параметров должны быть разные, и обязательно перед каждым именем должно стоять слово – ***class;***

4) все параметры шаблона обязательно должны быть использованы в спецификации параметров функции.

Пример неправильного определения шаблона:

template<class T1,classT2,classT3>
T3 f (T1 a, T2 b) {T3 c;… } //ошибка!

Пример правильного определения шаблона функций обменивающих значения параметров:

```
template <class R>
void swap ( R&x, R&y)
{R t= x; x=y; y=t;}
```

Далее, если в программе встретится фрагмент :

```
double k= 3.7, n= 7.3;
swap(k, n);
```

компилятор на основе шаблона сформирует определение функции:

```
void swap ( double&x, double&y)
{double t=x; x=y; y=t;}
```

Затем будет выполнено обращение к ней и значения переменных ***k*** и ***n*** поменяются местами.

Шаблоны служат для автоматического формирования конкретных определений функций по тем вызовам, которые транслятор обнаруживает в тексте программы. В зависимости от вызовов создает определения функций обрабатывающих данные различных типов.

Лекция 4

МНОГОФАЙЛОВАЯ ОРГАНИЗАЦИЯ ПРОГРАММЫ (МОДУЛЬ, ПРОЕКТ, РЕШЕНИЕ, ПРОСТРАНСТВА ИМЕН, СБОРКА)

Все нетривиальные программы сбираются из нескольких раздельно компилирующих единиц. Единицей компиляции является файл. Определения и декларации глобальных объектов в различных файлах программы должны быть согласованны.

Отличительной особенностью Си является отсутствие в языке современных средств раздельной зависимой компиляции.
Раздельно зависимая компиляция облегчает возможность использования объекта, определённого в другом модуле. Для этого достаточно указать имя модуля. Извлечение описания объекта и проверка правильности его использования обеспечивается компилятором. В Си независимо раздельная компиляция наделяется свойствами зависимой при помощи препроцессора, который является неотъемлемой частью языка. Согласованное использование глобальных имён и типов достигается за счёт наличия только единственной копии их описания. Описание глобальных объектов выделяется в отдельные файлы, которые называются заголовочными. Заголовочные файлы с помощью директива препроцессора include помещаются, как в файлы, где они определяются, так и в файлы, где они используются. В результате компилятор получает возможность находить несоответствие в описаниях одного и того же объекта. Для небольших программ удобно создавать один заголовочный файл. Он должен содержать описание всех объектов, используемых по крайней мере в двух файлах.

Один из способов повышения надёжности программ заключается в разбиении её на части, которые содержат только информацию необходимую для их работы. Набор взаимосвязанных процедур и тех данных, с которыми они оперируют, называются **модулем**, а подход построения программ - модульным стилем программирования. В языке Си модуль состоит из двух файлов: заголовочного (с расширением .h) и исполняемого (.cpp). Заголовочный файл представляет интерфейс модуля, а исполняемый файл задаёт реализацию функций, содержащихся в интерфейсе. Код пользователя, использующего только интерфейс модуля, не зависит от деталей его реализации.

Программа может состоять из нескольких модулей и файлов различных типов. Совокупность всех файлов и модулей программы образует **проект (project)**.
Приложение в Visual Studio .NET может состоять из нескольких проектов, совокупность которых называется термином **решение (Solution).** В результате компиляции решения создается исполняемый файл в формате PE (PE-файл), который называется **сборкой (assembly)** . Программист работает с решением, CLR - со сборкой.

Решение содержит один или несколько проектов, ресурсы, необходимые этим проектам, возможно, дополнительные файлы, не входящие в проекты. Один из проектов решения должен быть выделен и назначен стартовым проектом. Выполнение решения начинается со стартового проекта. Проекты одного решения могут быть зависимыми или независимыми. В уже имеющееся решение можно добавлять как новые, так и существующие проекты. Один и тот же проект может входить в несколько решений. Проект - это основная единица, с которой работает программист. Он выбирает тип проекта, а Visual Studio создает скелет проекта в соответствии с выбранным типом.

Проект состоит из классов, собранных в одном или нескольких **пространствах имен (namespace)** . Пространства имен позволяют структурировать проекты, содержащие большое число классов, объединяя в одну группу близкие классы. Если над проектом работает несколько исполнителей, то, как правило, каждый из них создает свое пространство имен. Помимо структуризации, это дает возможность присваивать классам имена, не задумываясь об их уникальности. В разных пространствах имен могут существовать одноименные классы. Таким образом, пространство имен — это логическая структура для организации имен, используемых в приложении .NET. Основное назначение пространств имен — предупредить возможные конфликты между именами. Класс проекта погружен в пространство имен, имеющее по умолчанию то же имя, что и решение, и проект. Итак, при создании нового проекта автоматически создается достаточно сложная вложенная структура - решение, содержащее проект, содержащий пространство имен, содержащее класс, содержащий точку входа. Для простых решений такая структурированность представляется избыточной, но для сложных - она осмысленна и полезна.

Основным понятием при программировании в среде .NET является понятие сборки. Согласно терминологии Microsoft код, предназначенный для работы в среде выполнения .NET, — это *управляемый код* (managed code). Двоичный файл, который содержит управляемый код, называется ***сборкой*** **(assembly)**. Приложения .NET создаются путем объединения любого количества сборок. Сборка — это двоичный файл (DLL или EXE), который содержит в себе номер версии, метаданные, а также типы (классы, интерфейсы, структуры и т. п.) и дополнительные ресурсы (изображения, таблицы строковых данных и т.д.).

Сборки .NET содержат не платформенно-зависимые инструкции, а код на так называемом **промежуточном языке Microsoft (Microsoft Intermediate Language, MSIL или просто IL)**. Этот язык не зависит ни от платформы, ни от типа центрального процессора. Код IL компилируется в платформенно-зависимые инструкции только во время выполнения.

Помимо собственно инструкций на языке IL, каждая сборка .NET содержит в себе информацию о каждом типе сборки и каждом члене каждого типа. Эта информация генерируется полностью автоматически. Любая сборка .NET содержит **манифест — набор метаданных о самой сборке**. Манифест содержит информацию обо всех двоичных файлах, которые входят в состав данной сборки, номере версии сборки, а также, что очень важно, — сведения обо всех внешних сборках, на которые ссылается данная сборка.

Причиной появления понятия сборки можно считать трудности установки Windows-приложений. Обычное Windows-приложение состоит из множества файлов - запускаемые модули, библиотеки, дополнительные файлы и т.п. Помимо этого, при установке некоторых приложений (особенно COM-компонент) необходимо записывать в реестр Windows сведения о нахождении и способе вызова. Наконец, многие приложения использовали разделяемые DLL, что зачастую приводило к проблемам при установке более новых версий этой DLL.

Понятие сборки было введено для того, чтобы решить эти проблемы. Сборка представляет собой набор файлов, модулей и дополнительной информации, которые должны обеспечить простую установку приложения и последующую работу. Таким образом, можно говорить и о том, что повторное использование приложений может быть реализовано с помощью интеграции различных сборок.

Пользователю сборки гораздо важнее ее логическое представление, в котором сборка — это набор открытых типов, используемых в приложении («внутренние» типы — это, как правило, служебные типы, используемые другими типами той же самой сборки). На физическом уровне сборка — это единственный исполняемый файл , а в тоже время на логическом уровне — это иерархия взаимосвязанных типов

ИНТЕГРИРОВАННАЯ СРЕДА РАЗРАБОТКИ

MICROSOFT VISUAL STUDIO .NET

Microsoft Visual Studio .NET - это интегрированная среда разработки (IDE) для создания, документирования, запуска и отладки программ, написанных на языках .NET.

Интегрированная среда разработки IDE (Integrated Development Envirionment) Visual Studio является многооконной, настраиваемой, обладает большим набором возможностей.

Главное окно Visual Studio.NET, подобно другим приложениям Windows, содержит строку меню,

включающую в себя следующие категории

- File — открытие, создание, добавление, закрывание, печать и проч.
- Edit — стандартные команды правки: копирование, вставка, вырезание и проч.
- View — команды для скрытия и отображения всех окон и панелей инструментов.
- Project — команды для работы с проектом: добавление элементов, форм, ссылок и проч.
- Build — команды компиляции программы.
- Debug — команды для отладки программы.
- Data — команды для работы с данными.
- Format — команды форматирования располагаемых элементов (выравнивание, интервал и проч.).
- Tools — команды дополнительных инструментов и настройки Visual Studio

.NET.

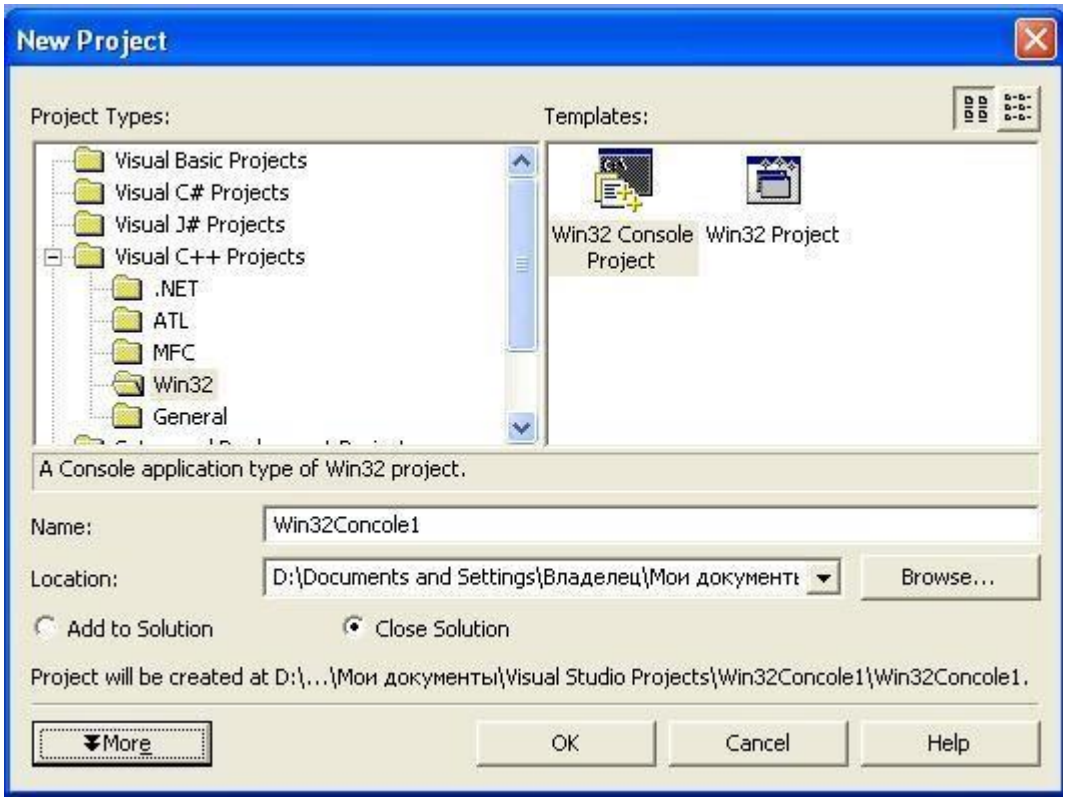
- Window — управление расположением окон.
- Help — справка.

Под строкой меню расположена *панель инструментов*, содержащая вложенные панели кнопок, запускающих те или иные команды из определенной группы или управляющих средой разработки Visual Studio.

Поместить группу кнопок на панель инструментов можно при помощи пункта меню View / Toolbars

Создание нового проекта

Если в меню среды разработки выбрать пункт File | New | Project, на экране появится диалоговая панель New Project .



Диалоговая панель New Project

При создании нового проекта в поле Location необходимо указать имя каталога, в котором следует сохранить его файлы. При этом в данном каталоге автоматически будет создан другой каталог, имя которого совпадает с именем проекта. По умолчанию проекты сохраняются в файле C:\Documents and Settings\Владелец\Мои документы\Visual Studio Projects\Имя проекта.

Виды проектов

Visual Studio .Net для языков C#, Visual Basic и J# предлагает 12 возможных видов

проектов. Среди них есть пустой проект, в котором изначально не содержится никакой функциональности; есть также проект, ориентированный на создание Web-служб.

В левой части этой диалоговой панели можно выбрать тип проекта. В общем случае можно выбрать проекты, созданные на языках программирования Visual Basic .NET, C#, C++, а также на ряде других. Этот список зависит от того, какие языки были выбраны при установке Visual Studio, а также от того, были ли приобретены и установлены дополнительные языки программирования сторонних производителей.
В правой части экрана можно выбрать один из предложенных шаблонов для данного типа проектов:

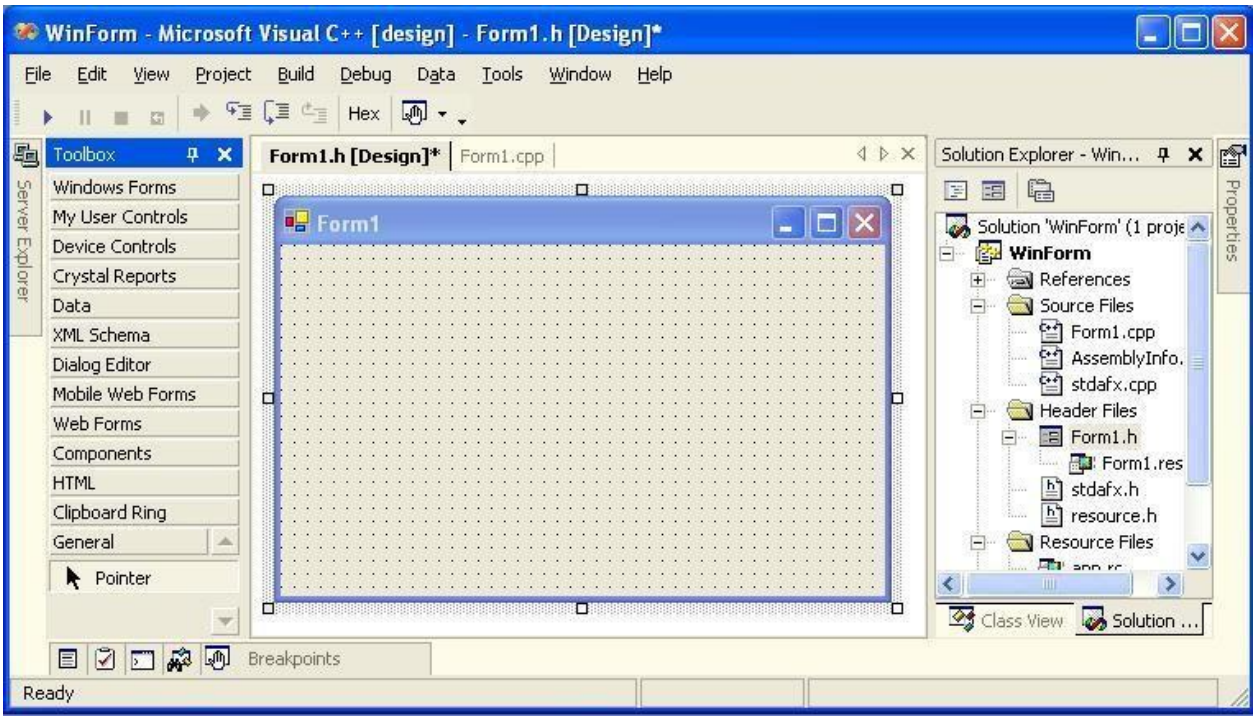
- Windows Application — шаблон для Windows-приложения;
- Class Library — шаблон для создания библиотеки классов, которая будет использоваться другими приложениями;
- Control Library — шаблон для создания элементов управления, которые будут использоваться в приложениях с графическим пользовательским интерфейсом для платформы Windows (называемых также приложениями Windows Forms);
- ASP.NET Web Application — шаблон для создания Web-приложений ASP.NET;
- ASP.NET Web Service — шаблон для создания Web-сервисов;
- Web Control Library — шаблон для создания элементов управления, которые будут использоваться в Web-приложениях;
- Console Application — шаблон для создания консольных приложений;
- Windows Service — шаблон для создания сервисов операционой системы;
- Empty Project/Empty Web Project — проект, который создается без использования шаблонов;
- New Project In Existing Folder — добавить новый проект в уже существующую папку.

Хотя при создании нового проекта в среде Visual Studio .NET предлагается довольно большой список типов проектов , но на самом деле есть всего три основные разновидности приложений - Windows Application, Console Application и Class Library . Все остальное - это их различные вариации за счет использования тех или иных шаблонов или мастеров (именно поэтому правое окно и называется Templates), обеспечивающих автоматическое выполнение каких-то начальных действий, которые при желании можно выполнить и "руками" (в том числе изменив и базовый тип приложения). Пользователь может подключить и свои собственные варианты шаблонов.

Основные части визуальной среды разработки Visual Studio

Существует три основные части визуальной среды при разработке проекта. В центре находится главное окно для создания визуальных форм и написания кода. Справа размещается окошко Solution Explorer (проводник решения), а ниже его окно инспектора

свойств Properties Explorer. Окно Solution Explorer позволяет увидеть, из каких проектов состоит решение и какие файлы входят в состав этих проектов. Окно свойств (Properties) содержит список атрибутов объекта, выделенного в данный момент. В левой части среды разработки присутствует элемент управления со значком окна Server Explorer; это окно появится, если указатель мыши окажется над данным значком. Там же имеется и значок окна Toolbox — оно появится, если поместить указатель мыши над этим значком.

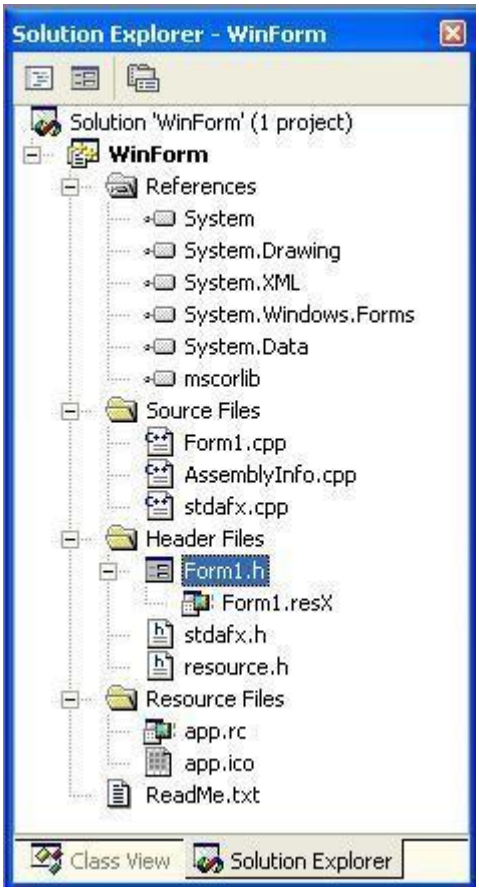


Среда разработки Visual Studio .NET содержит два типа окон — окна инструментов и окна документов. Окна инструментов (часть из которых была описана выше) доступны с помощью команд меню View и некоторых других, и их доступность зависит от типа приложения и от того, какие модули расширения (дополнительные утилиты и инструменты, в том числе произведенные сторонними разработчиками) добавлены к среде разработки. В окнах же документов можно редактировать компоненты проектов.
С окнами инструментов можно производить различные манипуляции. В частности, можно заставить их автоматически появляться и исчезать, группировать их в виде многостраничного блокнота, варьировать их расположение в среде разработки, делать их «плавающими» и даже отображать на дополнительном мониторе, если использование такового поддерживается операционной системой.
Некоторые окна инструментов, например окно Web Browser, можно создавать в виде нескольких экземпляров (это можно сделать, выбрав пункт меню Windows | New Window). Можно также заставить окна инструментов автоматически исчезать, если они в данный момент не являются активными, — в этом случае на экране отображаются название и пиктограмма окна, над которой можно поместить указатель мыши, если окно нужно отобразить целиком. Если необходимо предотвратить исчезновение окна с экрана, следует щелкнуть мышью по изображению канцелярской кнопки на заголовке окна.

Окно проводника решения (Solution Explorer)

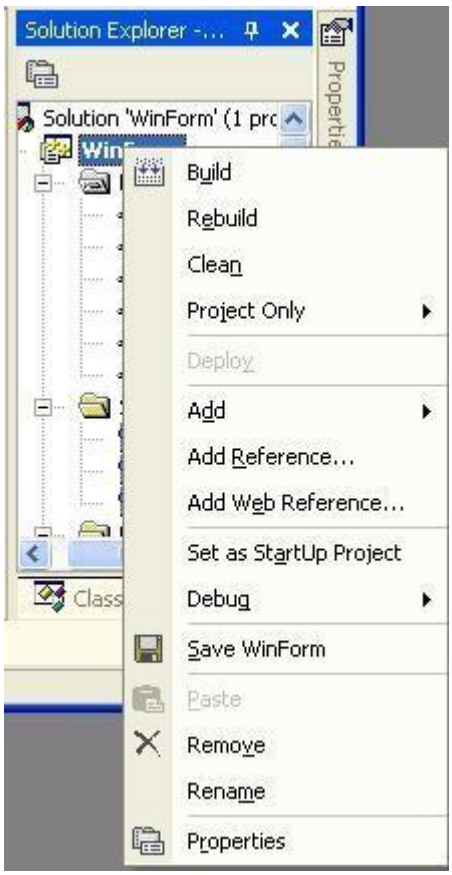
Окно Solution Explorer можно отобразить на экране с помощью команды меню View |

Solution Explorer.Окно Solution Explorer содержит древовидное представление элементов проекта, которые можно открывать по отдельности для модификации или выполне ния задач по управлению. В дереве отображаются логические отношения решения и проектов, а также элементов решения. Решение — это набор проектов, из которых состоит приложение. Компонентами проектов могут быть модули, а также другие файлы, которые требуются для создания приложения. Если нужно отредактировать компонент проекта, следует дважды щелкнуть по его имени в окне Solution Explorer.



Окно Solution Explorer

Пункты контекстное меню этого окна (вызывающегося нажатием правой кнопкой мыши) позволяют изменять содержимое проекта, а также добавлять новые компоненты. Помимо обычных программных модулей, мы можем с помощью команды File | Add Item подключать к проекту самые разные компоненты, например, HTML-страницу, которую затем можно наполнить с помощью встроенного HTML-редактора. Кроме того, в среде разработчика имеется новый дизайнер XML-документов и XSD-схем, набор графических редакторов и целый ряд других инструментов. Чтобы связать файлы с решением, но не с одним из его проектов, достаточно присоединить его прямо к решению.



С помощью кнопок, расположенных в верхней части окна Solution Explorer, можно указать, что именно должно отражаться в среде разработки:

- View Code — код, связанный с файлом, выделенным в окне Solution Explorer;
- View Designer — дизайнер (визуальный редактор) файла, выделенного в окне Solution Explorer;
- Refresh — обновить содержимое окна Solution Explorer;
- Show All Files — все файлы, включая код, связанный с формами;
- Properties — свойства выбранного файла.

При создании нового проекта Solution Explorer содержит компоненты, созданные шаблоном

Папка References содержит ссылки на классы, используемые в проекте по умолчанию. Двойной щелчок мыши на подпапках References запускает окно Object Browser (проводник объектов, View —> Object Browser, или сочетание клавиш Ctrl+Alt+J). Окно Object Browser, в свою очередь, является исчерпывающим средством получения информации о свойствах объектов. Можно получать краткое описание любого метода, класса или свойства, просто щелкнув на нем, — на информационной панели немедленно отобразится краткая справка.

Файлы проекта

В проекте Visual C++ .NET взаимозависимости между отдельными частями описаны в

текстовом файле проекта с расширением VCPROJ. А специальный тексто вый файл решения с расширением SLN содержит список всех проектов данного решения.Чтобы начать работу с существующим проектом, достаточно открыть в Visual C++ .NET соответствующий SLN-файл.Visual C++ .NET также создает промежуточные файлы нескольких типов

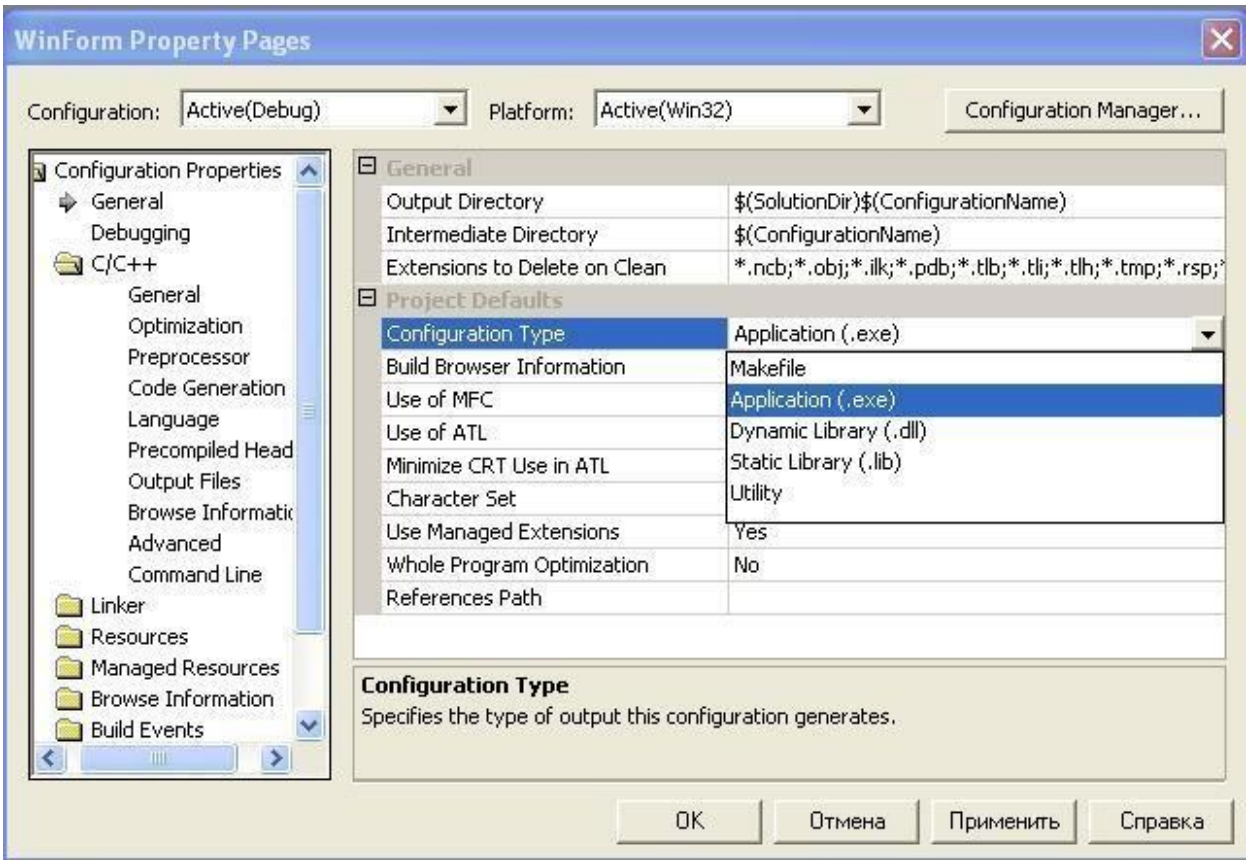
Содержимое папки Debug

RC, resX -Поддержка просмотра ресурсов
RES, RESURCES откомпилированные файлы ресурсов
NCB -Поддержка просмотра классов. Этот файл создается и затем обновляется при каждом запуске программы. Он имеет самый большой объем среди всех файлов проекта. С целью экономия места на диске ***файл с расширением NCB***, а также папку ***Debug***, которая образуется после компиляции программы, ***необходимо удалять***.
PDB файл, используемый компоновщиком для записи отладочной информации о пользовательской программе с целью ускорения редактирования связей в режиме отладки. Этот файл содержит отладочную информацию, а также информацию о состоянии проекта.
SLN Файл решения.
SUO Поддержка параметров и конфигурации решения
VCPROJ Файл проекта.
ICO Файл содержит изображение иконки, которое на форме расположено в верхнем левом углу.
Файл AssemblyInfo содержит информацию о приложении. При создании дистрибутива (установочного пакета) в этот файл помещается информация программы, используемая в технических целях, а также цифровой ключ.

Вся информация о том, из чего состоит наше приложение, находится в файле с расширением .sln (Microsoft Visual Studio Solution File). В этом файле, в частности, написано, как называется наше приложение, и какой файл проекта относится к нему. Файл проекта, имеющий расширение, отражающее выбранный нами язык программирования (WinApp.vcproj в нашем примере), — это XML-файл, содержащий все необходимые характеристики проекта. В частности, здесь есть информация о платформе, для которой создается результирующий файл (OutputType = “WinExe” в нашем примере), о начальном объекте (StartupObject = “WinApp.Form1” в нашем примере), имя корневого пространства имен (RootNamespace = “WinApp” в нашем примере). Отдельный интерес представляет список ссылок на пространства имен, доступных по умолчанию (остальные надо указывать с помощью ключевого слова using), а также список импортируемых пространств имен Список файлов, из которых состоит наше приложение, располагается в секции Files.

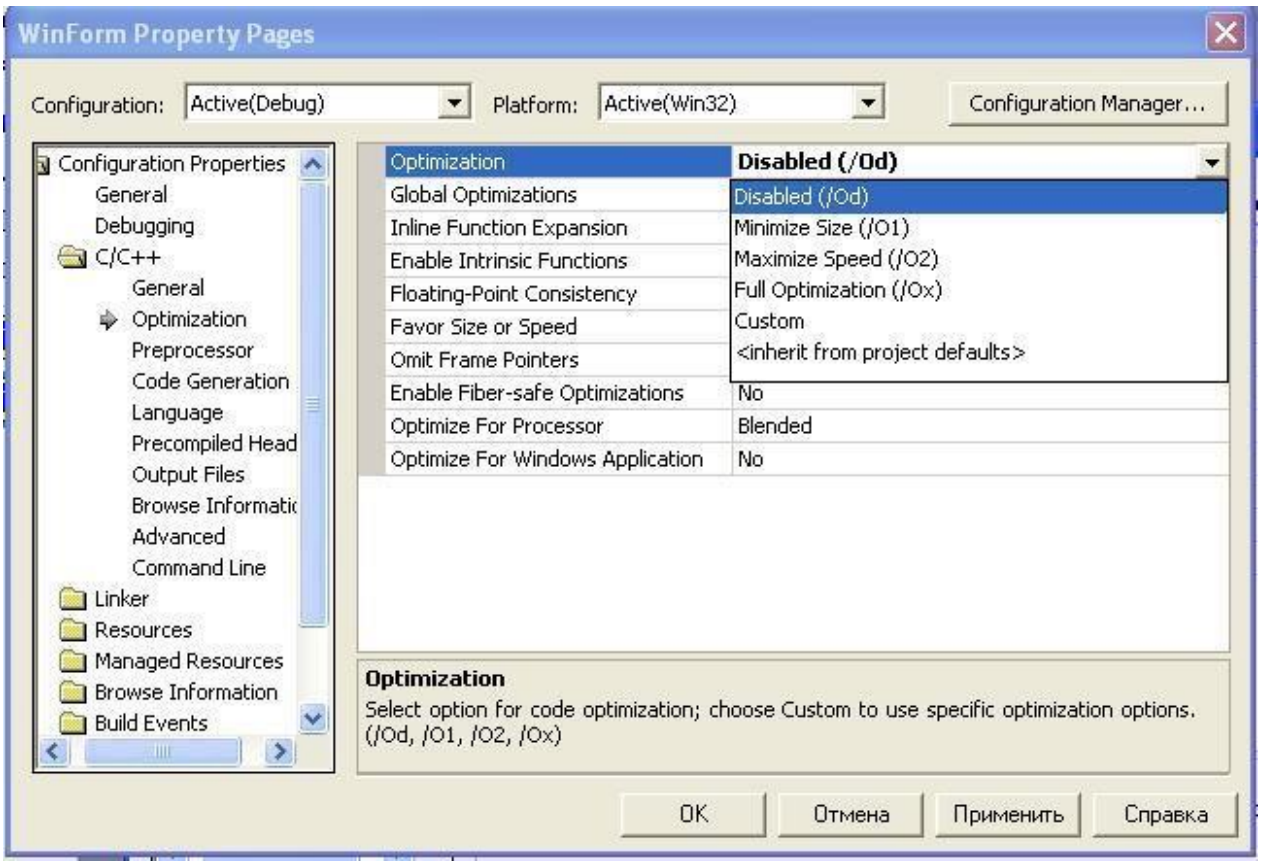
Свойства проекта

В окне Solution Explorer выделяем название проекта, щелкаем правой кнопкой мыши и отбражаемм контексное меню. В контексном меню выбираем пункт Properties.
В появившемся окне содержатся все свойства текущего проекта

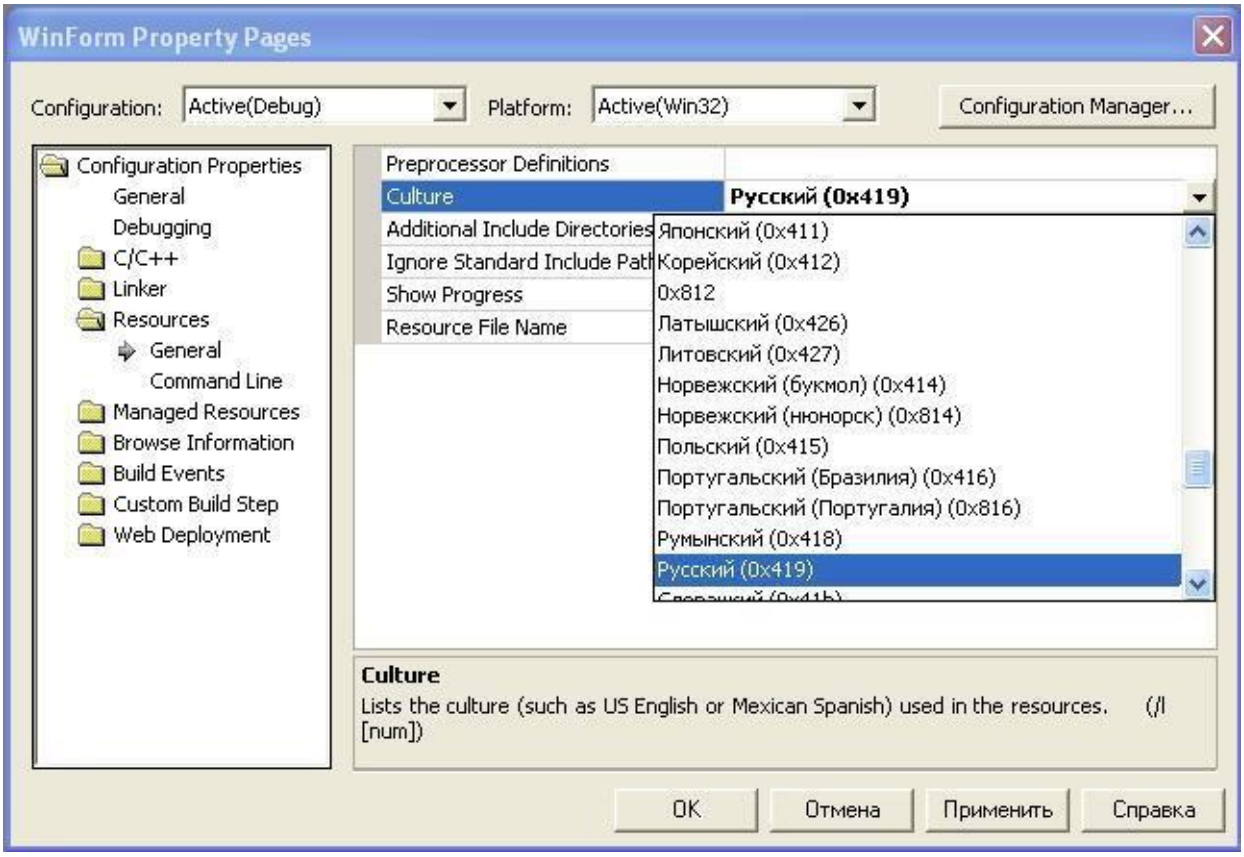


Configuration Type - тип программы, которая получается в резельтате компиляции. Только на языке C++ в Studio Net можно создать статически подключаемую библиотеку.

Optimize Code — оптимизация программы, значение этого свойства может значительно увеличить производительность приложения.

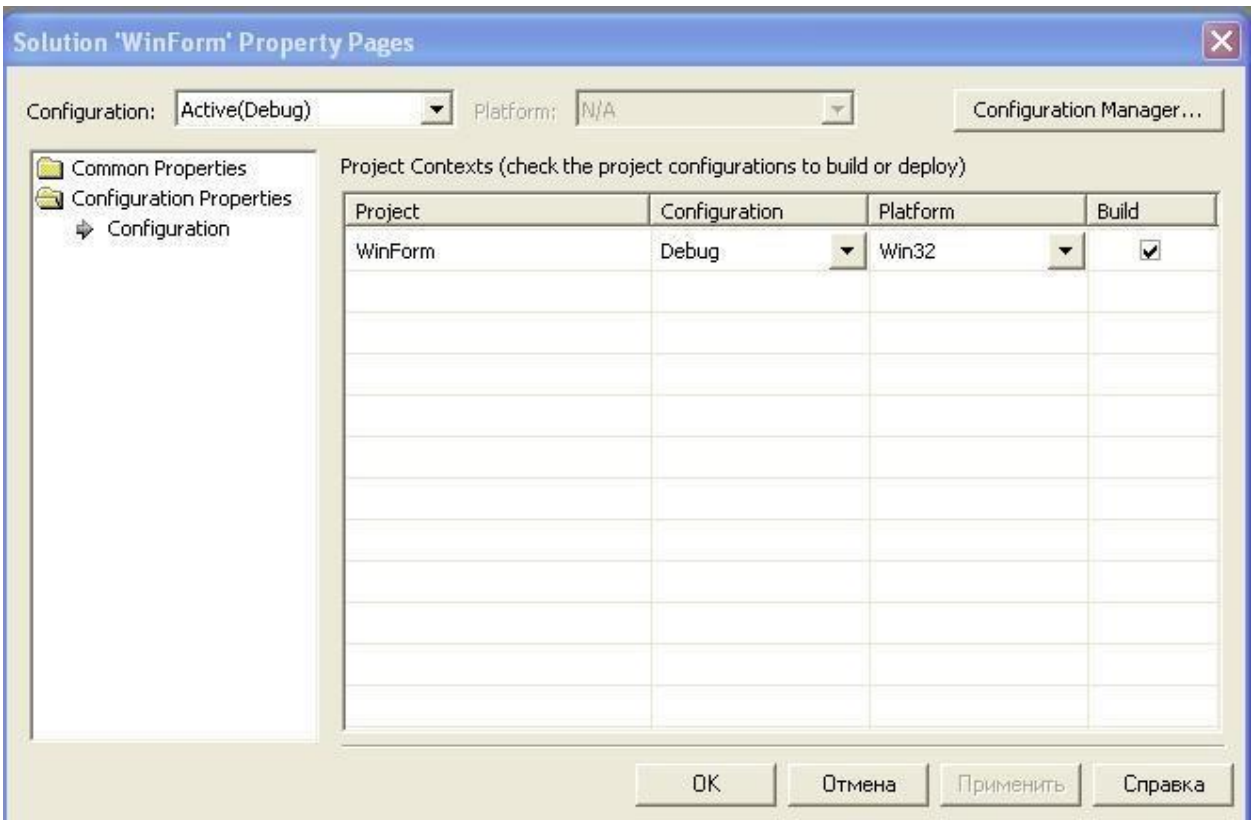


Диалоговое окно позволяет задать параметры культуры для приложения



Конфигурация проектов

В окне Solution Explorer выделяем название ***решения***, щелкаем правой кнопкой мыши и отбражаемм контексное меню. В контексном меню выбираем пункт Properties.
В появившемся окне содержатся общие свойства ***решения***



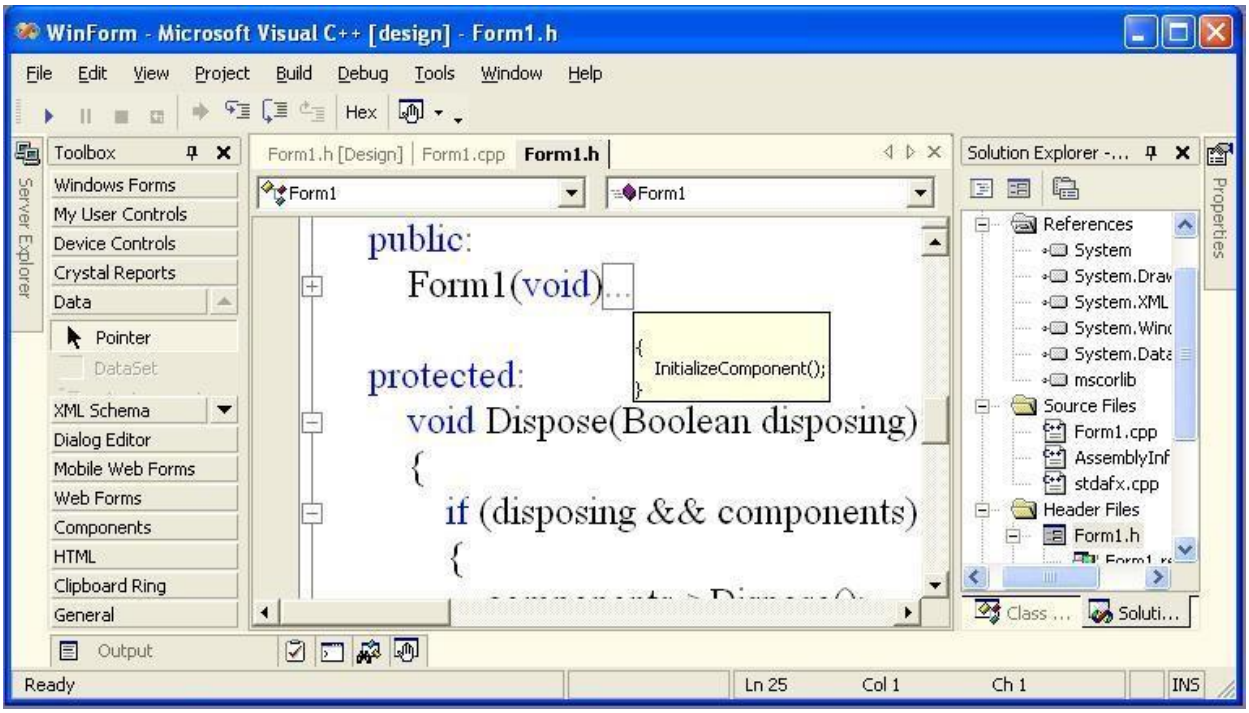
Конфигурация проекта определяет параметры компоновки приложения. Одновремен но может быть определено несколько различных конфигураций, причем приложение для каждой из них будет создаваться в отдельной папке, так что у вас есть возможность срав нить эти конфигурации. Изначально каждый проект в решении Visual Studio имеет две конфигурации — Debug (Отладка) и Release (Выпуск). При использовании конфигурации Debug (Отладка) будет создаваться отладочная версия проекта, с помощью которой можно осуществлять отладку на уровне исходного кода посредством установки точек ос танова и т.д. В папке Debug (Отладка) при этом будет находиться *файл,* используемый компоновщиком для записи отладочной информации о пользовательской программе с целью ускорения редактирования связей в режиме отладки. Этот файл имеет расширение *.pdb* и
содержит отладочную информацию, а также информацию о состоянии проекта. Необходимую конфигурацию можно выбрать с помощью элемента списка Debug (Отладка) на главной панели инструментов. То же самое можно сделать, выбрав пункт меню Builds Configuration Manager (Компоновка Диспетчер конфигурации...), что приведет к запуску диалога Configuration Manager (Диспетчер конфигурации). Из выпадающего списка Active Solution Configuration (Текущая конфигурация решения) выберите пункт Release (Выпуск). Скомпонуйте проект еще раз. Теперь создана вторая версия программы , причем на этот раз она помешается в папку Release (Выпуск).

Редактор кода

Окна документов предназначены для редактирования компонентов проектов. Их взаимное расположение зависит от выбранного режима отображения окон в среде разработки.

Иерархическая структура программного кода

программные модули реализованы в виде иерархической структуры



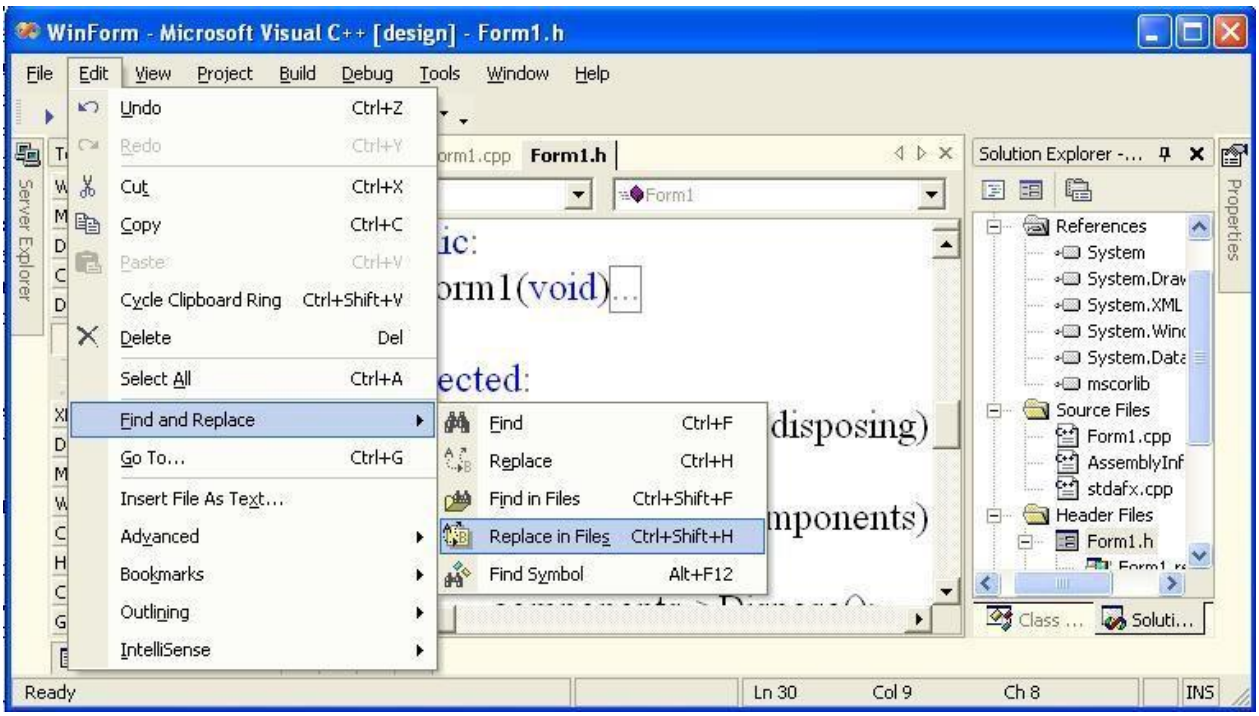
Верхний уровень иерархии определяется операторными скобками. Каждый узел на этой линейке соответствует отдельной процедуре (или аналогичной конструкции). С помощью узлов можно делать видимыми только нужные для работы фрагменты кода. Если мы закроем узел, то в заглавной строке этого блока появится небольшое окошко с многоточием. Подведя к нему мышь, мы сможем увидеть содержимое данной конструкции
очень полезное новшество — использование операторных скобок #Region, которые позволяют группировать в блоки отдельные процедуры

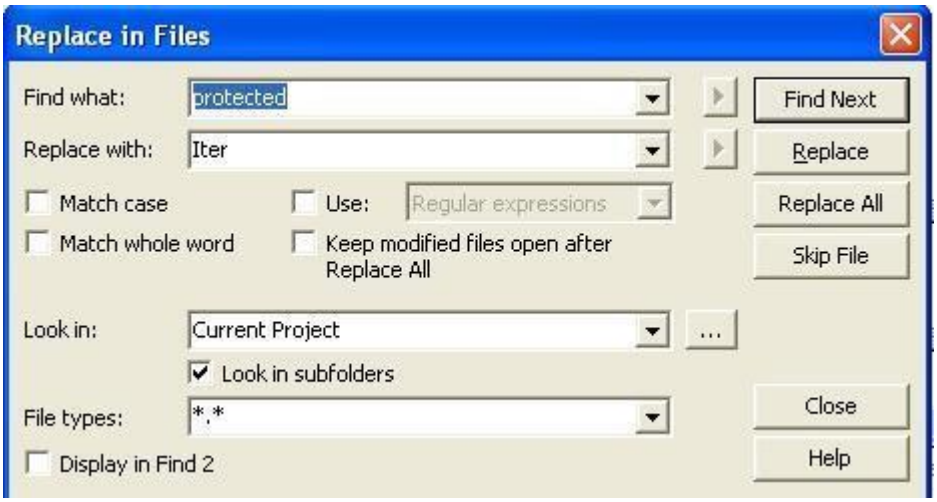
Контекстный поиск и замена

Окно редактирования можно разбить на несколько частей, в которых будут отображаться разные фрагменты кода. Допустимо также отобразить второе окно редактирования с помощью пункта меню Window | New Window.
В редакторе кода можно осуществлять контекстный поиск и замену текста в текущей процедуре, текущем модуле или в выделенном фрагменте кода с помощью стандартной диалоговой панели Windows Find and Replace.

В строке для поиска могут содержаться символы «*» и «?», означающие любую последовательность символов и любой символ соответственно.
Возможен также поиск и замена фрагментов текста во всех файлах проекта. В этом случае

следует использовать диалоговые панели Find in Files и Replace in Files.





Помимо фрагментов кода можно искать также названия классов и структур — для этой цели используется диалоговая панель Find Symbols. Результаты поиска отображаются в окне Find Symbol Results.

В редакторе кода можно установить закладку на какую-либо строку кода и вернуться к ней позже. Закладки не исчезают и при сохранении проекта.

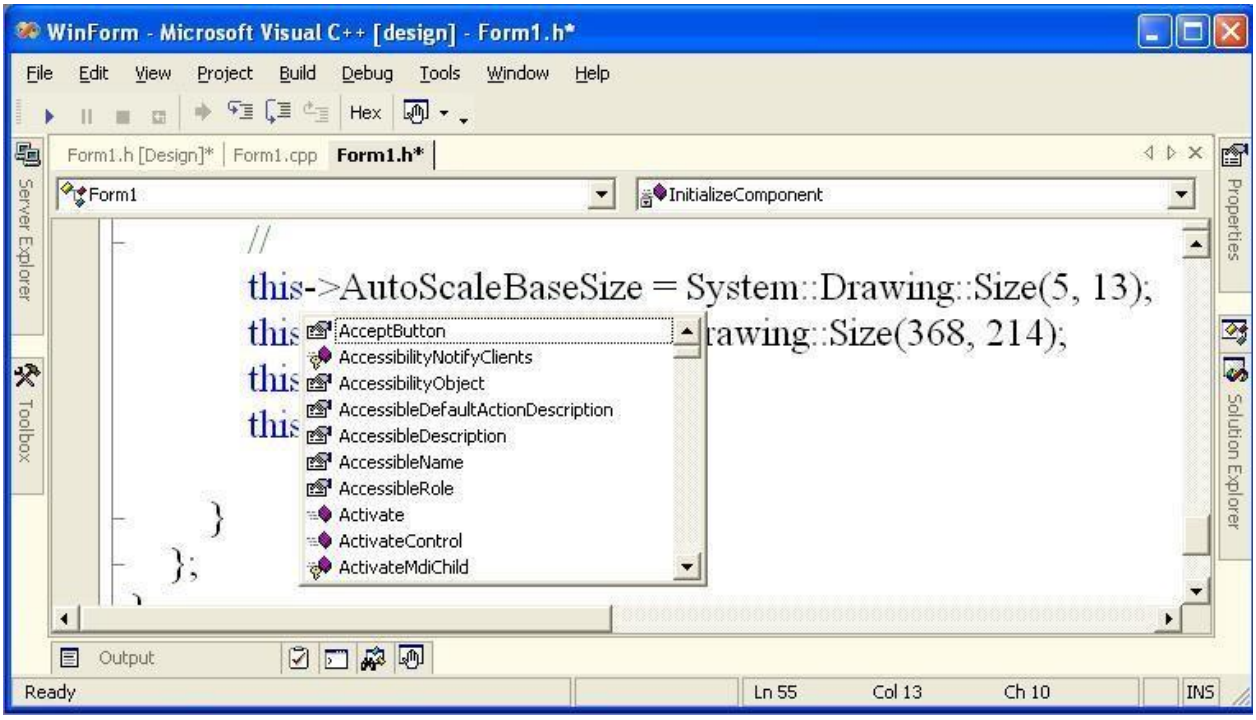
Можно также создать комментарий, связанный с выделенным фрагментом текста, с помощью команды меню Edit | Advanced | Comment Selection.

Возможно перемещение фрагментов текста посредством мыши в другое место, копирование фрагментов, а также перемещение фрагментов текста из редактора кода в окно Toolbox

IntelliSense (выпадающий список-подсказка)

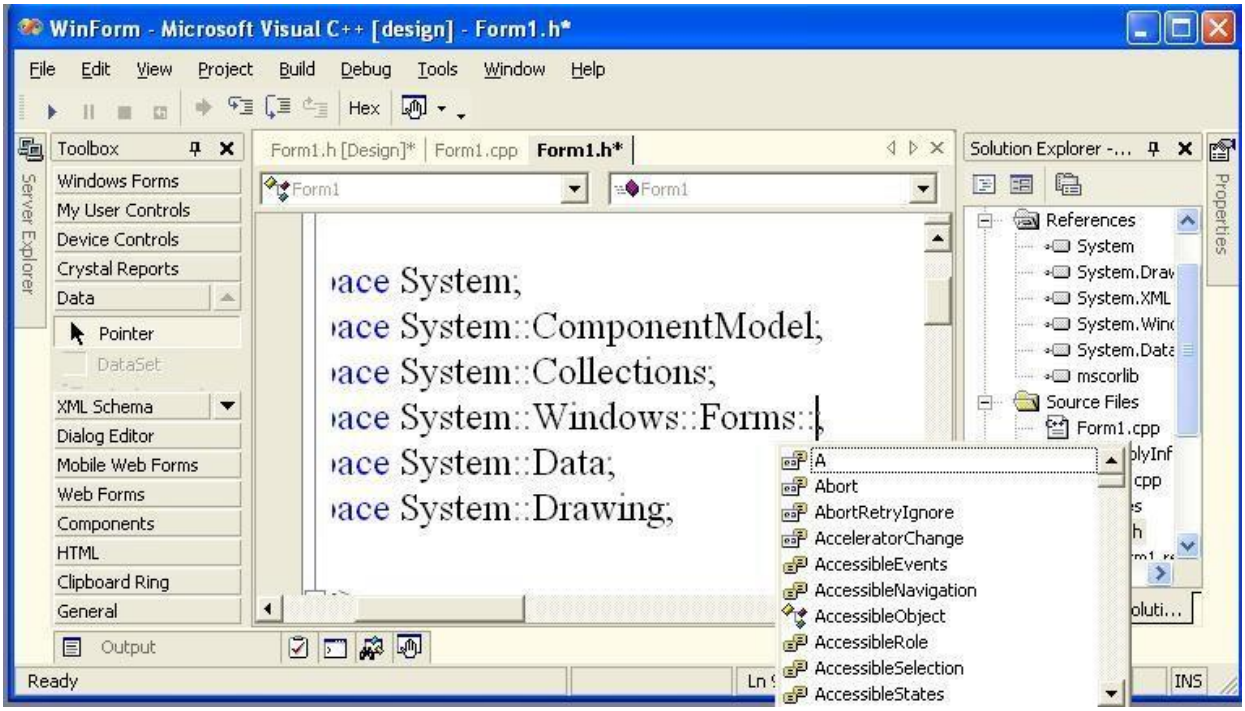
В Visual Studio после набора имени объекта и ввода точки, либо набора имени указателя на объект и стрелки (->) на экране появляется список свойств и методов данного

объекта.



При вводе имени метода (или функции) и круглой открывающейся скобки можно увидеть на экране описание метода и его параметров.

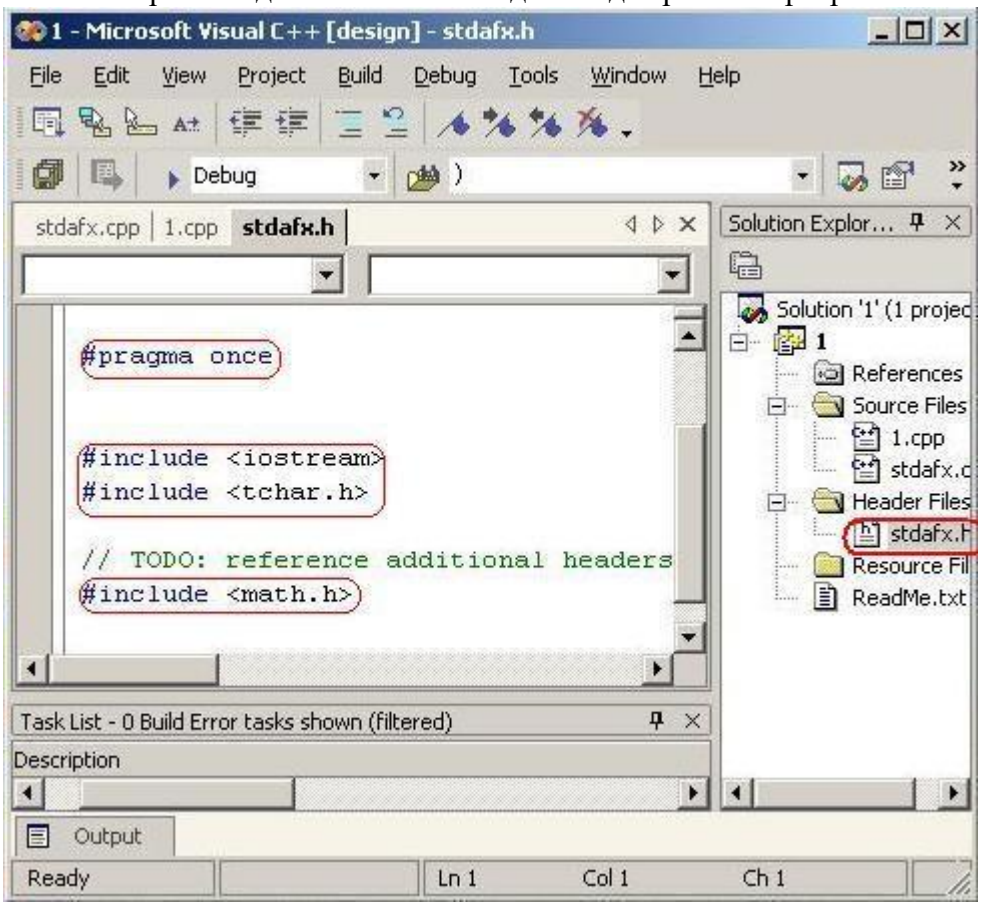
После набора имени пространства имен и :: на экране отображается его содержимое.



РАЗРАБОТКА КОНСОЛЬНОГО ПРИЛОЖЕНИЯ

Механизм предварительной компиляции заголовочных файлов

Механизм предварительной компиляции заголовочных файлов реализован при помощи пары файлов Stdafx.h и Stdafx.cpp. В файл Stdafx.h добавляются при помощи директивы #include строки подключения необходимых для работы программы заголовочных файлов.



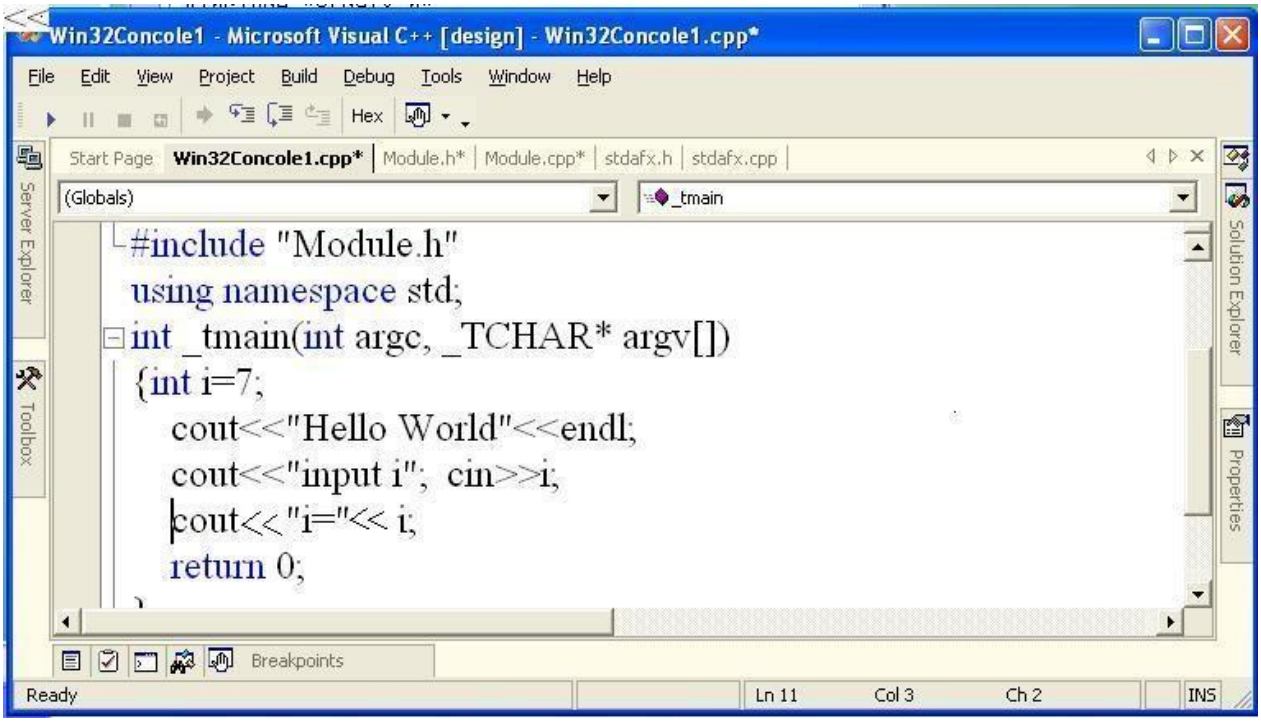
В данном примере подключается заголовочный файл <math.h> для работы с математическими функциями. Строки подключения библиотек <iostream> , <tchar.h > генерируются автоматически при создании шаблона консольного приложения. Директива препроцессора #pragma once обеспечивает включение только единственной копии заголовочных файлов

Сам файл Stdafx.h включается во все файлы реализации программы (*.CPP). Все другие директивы препроцессора, например , #define (определение именованных констант) и #include (подключения заголовочных файлов, не требующих предварительной компиляции) располагаются ниже строки #include <Stdafx.h>. При нарушении этих правил возникает ошибка на стадии компиляции – не найден конец файла.

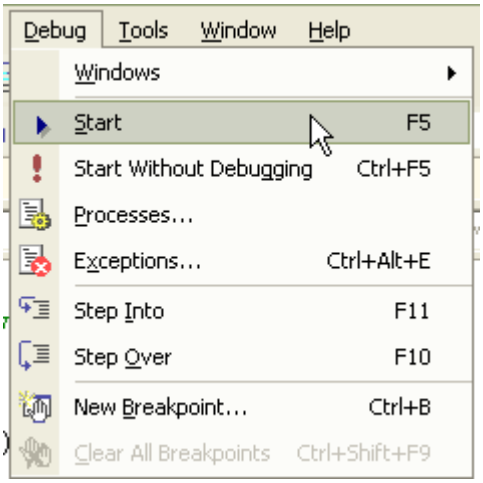
Ввод и вывод данных

В консольном приложении Win32 для ввода и вывода данных можно использовать все функции стандартной библиотеки языка Си, а также средства потокового ввода-вывода языка C++. Причем подключение заголовочных файлов stdio.h и iostream.h не требуется, поскольку сгенерированный шаблон приложения содержит строку #include <iostream>.

Средства потокового ввода-вывода языка C++ принадлежат пространству имен std. Поэтому для доступа к объектам можно явно указывать принадлежность к пространству имен, например, std::cout, либо, что более удобно, сделать доступным все пространство имен при помощи строки
using namespace std;
Данная строка помещается до основной программы.



Компиляция программы



Пункт главного меню Debug

При запуске консольных приложений при помощи команды Start Without Debugging окно с результатами работы программы исчезает только после нажатия клавиши Enter.

ОТЛАДКА ПРОГРАММЫ

Использование режима останова

Одним из способов обнаружить логическую ошибку является выполнение кода программы по одной строке и изучение изменения содержимого одной или нескольких переменных. Для этого можно при работе программы войти в режим останова, а затем просмотреть код в Редакторе кода. Режим останова дает возможность просмотреть программу во время ее исполнения. Точка останова показывает то место программы, где выполнение будет остановлено, и вы сможете использовать инструменты разработки Visual Studio. Простейший способ установить точку останова— щелкнуть на сером поле слева от окна с исходным кодом программы. Можно также переместить курсор на нужную строку и щелкнуть на кнопке "Точка останова" на панели инструментов. Щелчок на этой кнопке установит точку останова, если ее там не было, и, наоборот, уберет ее, если она уже
была установлена на этой строке. Теперь, если запустить программу в режиме отладки, ее выполнение остановится при достижении точки останова. Желтая стрелка на красном кружке, обозначающем точку останова, указывает, на какой именно точке останова прервано выполнение программы

установить точку останова

Переместите указатель мыши к полосе Margin Indicator (указатель поля - серая полоса, расположенная сразу за левым полем окна Редактора кода) рядом с оператором , а затем щелкните на этой полосе, чтобы установить точку останова. Немедленно появится красная точка останова. .

оператор, который вы выбрали для создания точки останова, выделен желтым цветом, а на полосе Margin Indicator (Указатель поля) появилась стрелка. Теперь Visual Studio находится в режиме останова, и вы можете узнать этот режим по слову "[break]", появившемуся в строке заголовка Visual Studio.

Поместите указатель мыши в Редакторе кода над переменной . Visual Studio выведет сообщение, в котором будет указано значение этой переменной.

Щелкните на кнопке Stop Debugging (Остановить отладку) панели инструментов Debug (Отладка).

Удаление точки останова

1. Щелкните в Редакторе кода на красном кружке, расположенном на полосе Margin Indicator (Указатель поля) и ассоциированном с точкой останова. Точка останова исчезнет. Это все, что касается этой задачи. Заметьте, что если у вас в программе более одной точки останова, вы можете удалить их все, щелкнув на команде Clear All Breakpoints (Снять все точки останова) из меню Debug (Отладка). Visual Studio сохраняет точки останова в вашем проекте, так что важно знать, как удалять их; в противном случае они останутся в вашей программе, даже после ее перезапуска!

2. Щелкните на кнопке Stop Debugging (Остановить отладку) панели инструментов Debug (Отладка). Выполнение программы завершится.

3. В меню View (Вид) укажите на Toolbars (Панели инструментов), а затем щелкните на Debug (Отладка). Панель инструментов Debug (Отладка) закроется.

панель инструментов Debug

При отладке вашего приложения используется панель инструментов Debug (Отладка) - специальная панель инструментов, предназначенная для поиска ошибок. . Ее можно открыть, выбрав команду Toolbars (Панели инструментов) в меню View (Вид), а затем щелкнув на Debug (Отладка).

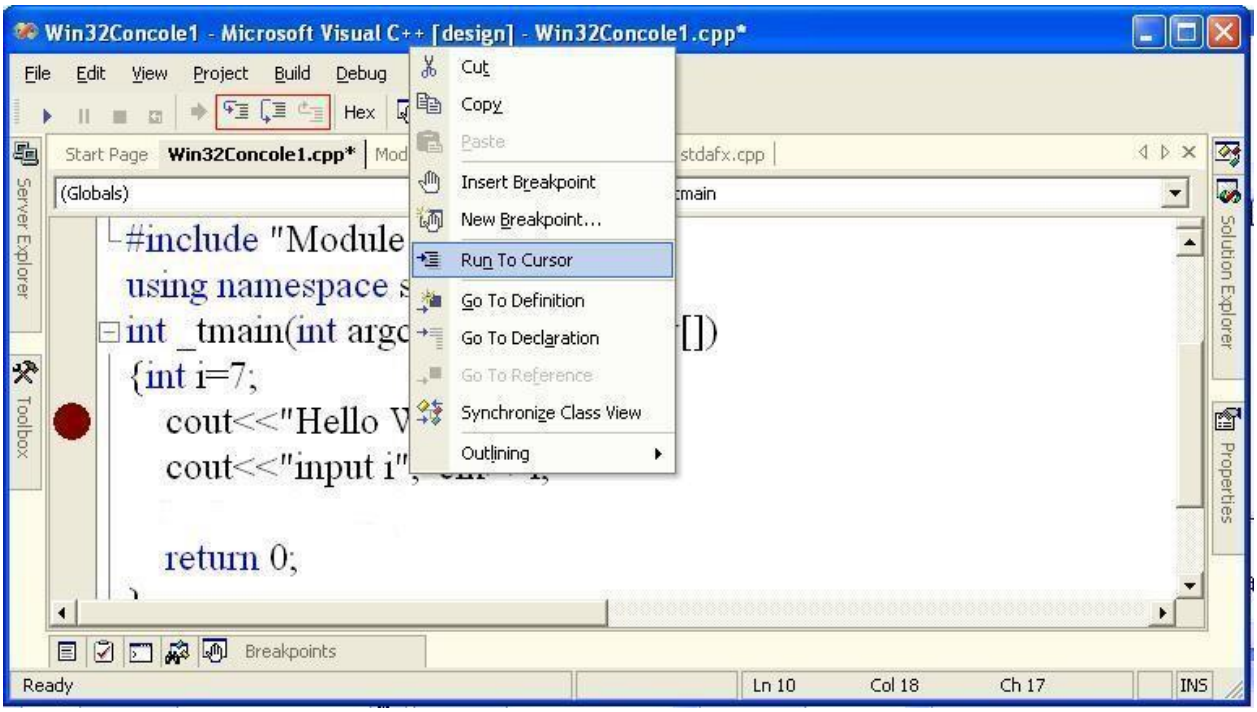
Выполнение в пошаговом режиме

После остановки программы отладчиком можно продолжить ее выполнение в пошаговом режиме. . Есть несколько кнопок, предназначенных для выполнения в пошаговом режиме. Наиболее часто используются следующие из них (в порядке расположения на панели инструментов):

• Step Into (Шаге заходом);
• Step Over (Шаг с обходом);
• Step Out (Шаг с выходом).

Есть еще одна команда контекстного меню— Run to Cursor (Выполнить до текущей позиции).

Если курсор находится на вызове какой-либо функции, то при щелчке на кнопке Step Into (Шаг с заходом) он перейдет на первую строку этой функции. Если же щелкнуть на кнопке Step Over (Шаг с обходом), произойдет вызов функции и курсор переместится на следующую строку



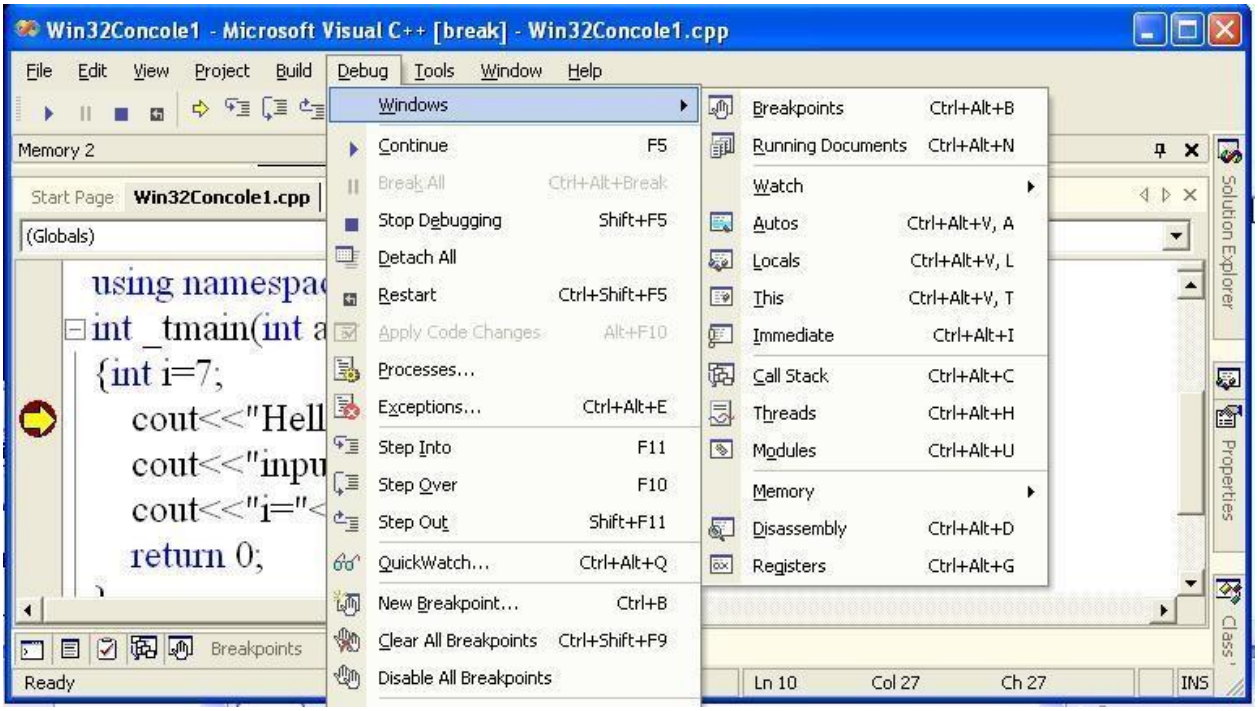
Контрольные значения

В режиме останова можно посмотреть, как обрабатывается логика вашей программы, в часности просмотреть значения переменных. . Простейший способ это сделать навести указатель мыши на переменную, и во всплывающей подсказке (на желтом фоне) около

курсора будет выведено значение переменной. . В режиме останова можно изучить значения свойств и переменных, но нельзя изменить код программы в Редакторе кода - при выполнении программы он заблокирован и защищен. Для просмотра переменных изменения их значений и вычисления выражений используются окна отладки.

Окна отладки

Открываюшийся список, который расположен последним на панели инструментов Debug , дает доступ к полному набору окон отладки, имеющихся в Visual Studio. В интерфейс пользователя Visual Studio .NET добавлено несколько новых окон отладки, включая Autos (Видимые), Command (Окно команд), Call Stack (Список задач), Threads (Потоки), Memory (Память), Disassembly (Дизассемблированный код) и Registers (Регистры).



Окно Quick Walch (Контрольное значение)

Для отображения окна необходимо щелкнуть правой кнопкой мыши на переменной и выбрать пункт Quick Walch (Контрольное значение) во всплывшем меню (или воспользоваться командой Quick Walch в пункте меню Debug). В этом окне можно изменить значение переменной

Окно Autos (Видимые)

Открытие окна Autos

В меню Debug (Отладка) укажите на Windows (Окна), а затем щелкните на Autos (Видимые).

Окно Autos (Видимые) - это автоматическое окно, показывающее состояние переменных и свойств, которые используются в текущий момент. Окно Autos (Видимые) полезно для изучения состояния отдельных переменных и свойств. Но элементы в окне Autos (Видимые) сохраняют свои значения только для текущего (выделенного в

отладчике) и предыдущего оператора (того, который только что был выполнен). Когда программа доходит до выполнения кода, не использующего эти переменные, они исчезают из окна Autos (Видимые).

Окна Watch

Открытие окна Watch

Чтобы видеть содержимое переменных и свойств на протяжении всего выполнения программы, используйте окно Watch (Контрольное значение) - специальный инструмент Visual Studio, который отслеживает нужные значения, при работе в режиме останова. В Visual Studio .NET вы можете открыть до четырех окон Watch (Контрольное значение). Эти окна пронумерованы как Watch 1 (Контрольное значение 1), Watch 2 (Контрольное значение 2), Watch 3 (Контрольное значение 3) и Watch 4 (Контрольное значение 4) и находятся в подменю Watch (Контрольное значение), которое вы можете открыть, выбрав команду Windows (Окна) в меню Debug (Отладка). Также можно добавлять в окно Watch (Контрольное значение) выражения.

Окно команд

Окно Command (Окно команд) - инструмент среды разработки Visual Studio двойного назначения. Когда окно команд находится в режиме Immediate (Интерпретация), вы можете использовать его для взаимодействия с кодом отлаживаемой программы. Когда окно команд находится в режиме Command (командном), вы можете использовать его для исполнения команд Visual Studio, таких, как Save All (Сохранить все) или Print (Печать). Если вы исполняете более одной команды, то можете использовать клавиши со стрелками для просмотра предыдущих команд и их результатов.

Открытие окна команд в режиме Immediate (Интерпретация)

В меню Debug (Отладка) укажите на Windows (Окна), а затем щелкните на Immediate (Интерпретация). Visual Studio откроет окно Command (Окно команд) в режиме Immediate (Интерпретация) - специальном состоянии, которое позволяет вам взаимодействовать с программой в режиме останова. В режиме Immediate (Интерпретация) строка заголовка окна содержит текст "Command Window - Immediate" ("Окно команд: интерпретация"). Если окно команд находится в режиме Command (Командный), вы можете переключить его в режим Immediate (Интерпретация), введя команду **immed**. У режима Immediate (Интерпретация) окна команд много способов применения: он представляет собой великолепное дополнение к окну Watch (Контрольное значение) и поможет экспериментировать с различными тестовыми ситуациями, которые было бы сложно ввести в программу другим способом.

Переключение окна команд в командный режим

Если окно команд находится в режиме Immediate (Интерпретация), вы можете переключить его в командный режим, введя команду **>cmd**. (Символ > обязателен.) Окно команд также может быть использовано для запуска команд интерфейса среды Visual Studio. Например, команда File.SaveAll сохранит все файлы текущего проекта.

Запуск команды File.SaveAll

1. Для переключения в командный режим, введите в окне команд команду >cmd, а затем нажмите на (Enter). Строка заголовка окна команд изменится на

"Command Window" (Окно команд) и в окне появится символ запроса команд ">" (визуальная подсказка о том, что окно находится в командном режиме).

2. Введите в окне File.SaveAll, а затем нажмите на (Enter). Visual Studio сохранит текущий проект, а затем снова появится запрос команды.

Окно команд использует функцию автозавершения написания команд и показывает вам все команды, начинающиеся с символов, которые вы уже ввели. Это мощная функция, с помощью которой можно найти большинство команд, выполняемых с помощью окна команд.

3. Щелкните на кнопке Закрыть окна команд.

Лекция 5

Визуальная разработка приложений.

Создание Windows-приложения заключается в расположении компонентов на форме, изменении их свойств, написании кода для обработки возникающих событий и написании кода, определяющего логику самого приложения.

Формы

Форма - это прямоугольное окно на экране. Форма — это каркас, на котором проектируется интерфейс программы. Форма — это экранный объект, который содержит элементы управления и обеспечивает функциональность программы.

Форма в .NET реализуется классом Form из глубоко вложенного пространства имен System.Windows.Forms. Для главного окна приложения Visual Studio .NET строит по умолчанию класс Form1, который является наследником класса Form и автоматически наследует его функциональность - свойства, методы, события.

public__gc class Form1 : public System::Windows::Forms::Form

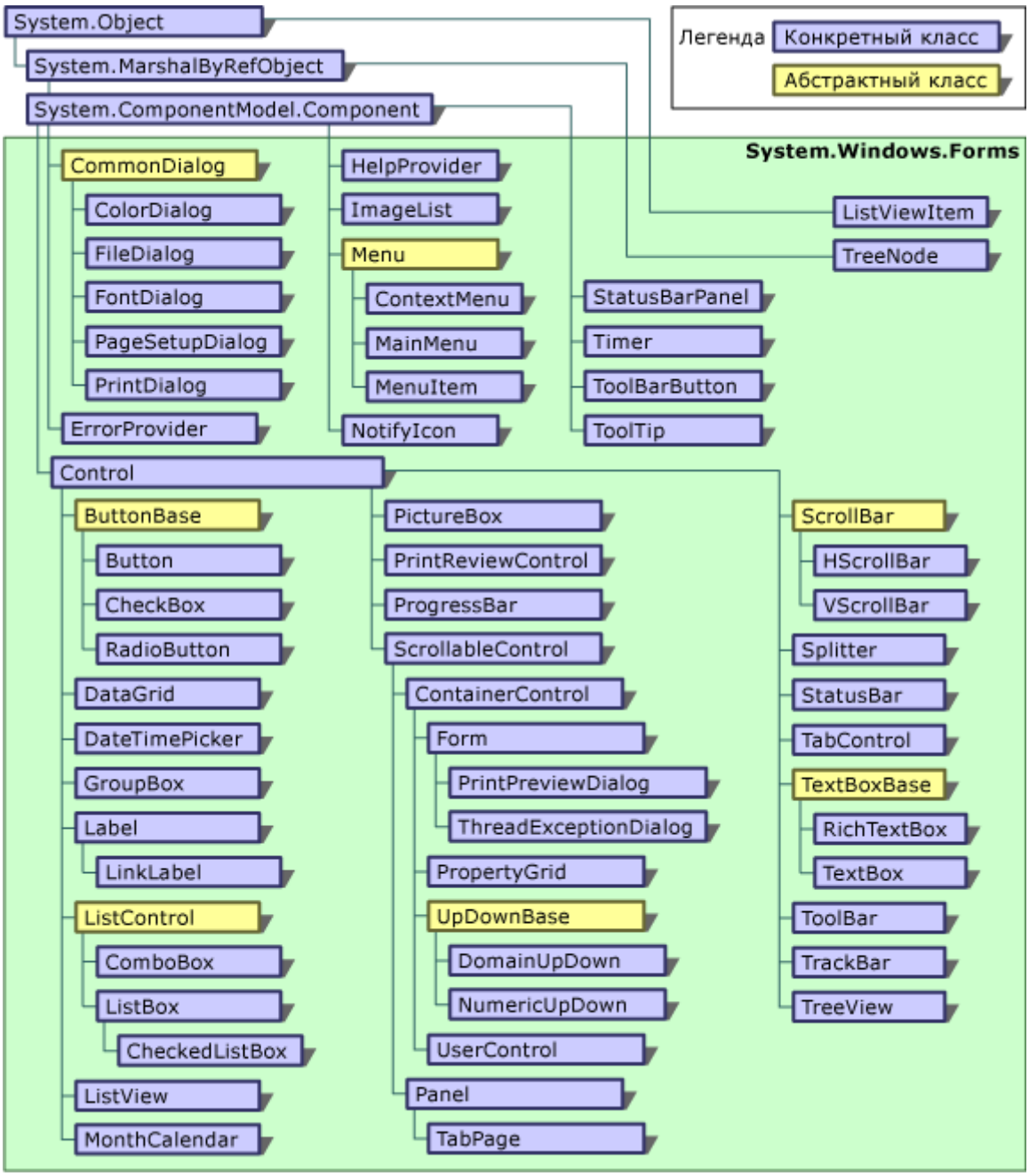
Visual Studio .NET унифицирует процедуру создания форм для Web и Windows. Кроме того, в Visual Studio .NET реализован механизм визуального наследования Windows-форм, что упрощает повторное использование кода

Компонентная модель.

Использование готовых компонентов - ключевая технология программирования на протяжении всей истории этого вида деятельности. Понятие компоненты выходит за рамки понятий интерфейсного элемента и класса. Главное отличие заключается в полной функциональности компонентов на стадии проектирования. Находясь в интегрированной среде, компоненты можно использовать непосредственно, менять их свойства, облик и

поведение или порождать производные элементы, обладающие нужными характеристиками.

Следующая иллюстрация показывает иерархию компонентов и элементов управления в пространстве имен System.Windows.Forms.



Конструирование приложения.

Конструирование приложения осуществляется по способу "drag and drop", и заключается в перетаскивании захваченных мышью визуальных компонентов из окна Toolbox на форму приложения. Форма представляет собой окно приложения с управляющими компонентами, которые переносятся на стадии проектирования или создаются динамически в процессе работы программы.

Автоматическая генерация кода.

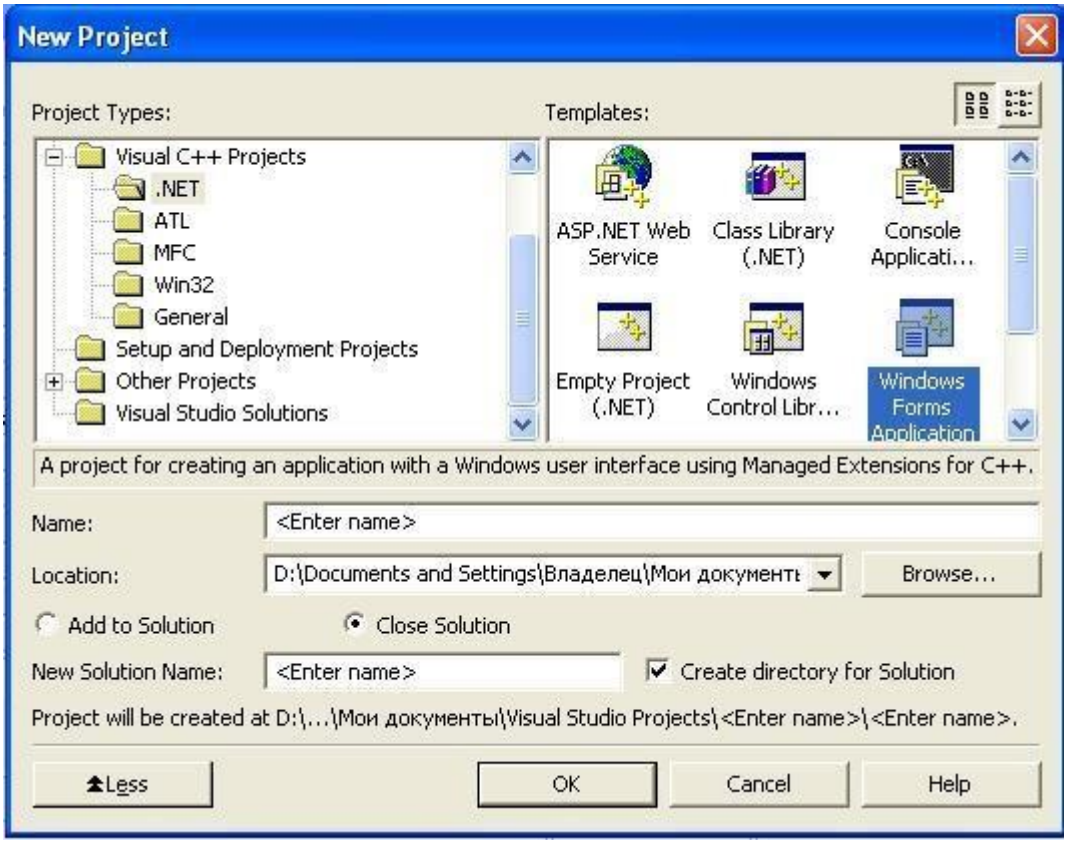
При перемещении объекта на форму приложения объявление объекта появляется в заголовочном файле. Воздействие пользователя на объект в процессе конструирования формы или выбор события из заданного списка приводит к генерации объявления метода-обработчика этого события. Синхронизация процессов проектирования формы и автоматической генерации кода ускоряет визуальную разработку приложений, полностью сохраняя контроль над исходным текстом.

Механизм двунаправленной разработки.

Технология двунаправленной разработки обеспечивает контроль над кодом посредством интегрированного и синхронизированного взаимодействия между инструментами визуального проектирования и редактором кода.

Создание проекта при помощи шаблона Windows Forms Application

Создание Windows-приложений начинается с того, что мы создаем новый проект типа Windows Application, выбирая соответствующий шаблон для одного из установленных языков программирования — C++ и т.д.



Нажатие кнопки OK приводит к загрузке выбранного нами шаблона и к появлению основных окон среды разработчика. Этими окнами являются:

- окно дизайнера форм;
- палитра компонентов;

- окно для просмотра компонентов приложения (Solution Explorer);
- окно для установки свойств (Properties).

Генерация кода

Microsoft Visual Studio .NET автоматически генерирует необходимый код для инициализации формы, добавляемых нами компонентов и т.п. Для каждой формы создается отдельный файл, который имеет то же имя, что и класс, описывающий форму.

Файл, в котором располагается код, отвечающий за функционирование формы, состоит из двух частей. Первая из них, с меткой “Windows Form Designer generated code”, содержит код, который копируется из шаблона приложения, — этот код не должен редактироваться программистами! Чтобы ограничить доступ к нему, в редакторе кода в Visual Studio .NET он помечается специальным образом.

Пространству имен предшествует 6 предложений using; это означает, что используются не менее 6-ти классов, находящихся в разных пространствах имен библиотеки FCL.

```
namespace MyApp
{
using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;
```

Одним из таких используемых классов является класс Form из глубоко вложенного пространства имен System.Windows.Forms. Построенный по умолчанию класс Form1 является наследником класса Form и автоматически наследует его функциональность - свойства, методы, события. При создании объекта этого класса, характеризующего форму, одновременно Visual Studio создает визуальный образ объекта - окно, которое можно заселять элементами управления. В режиме проектирования эти операции можно выполнять вручную, при этом автоматически происходит изменение программного кода класса. Появление в проекте формы, открывающейся по умолчанию при запуске проекта, означает переход к визуальному, управляемому событиями рограммированию. Сегодня такой стиль является общепризнанным, а стиль консольного приложения следует считать устаревшим, правда, весьма полезным при изучении свойств языка.

В класс Form1 встроено закрытое (private) свойство - объект components класса Container. В классе есть конструктор, вызывающий закрытый метод класса InitializeComponent. В классе есть деструктор, освобождающий занятые ресурсы, которые могут появляться при добавлении элементов в контейнер components.

```
public__gc class Form1 : public System::Windows::Forms::Form
{
public:
Form1(void)
{
```

```
InitializeComponent();
}

protected:
void Dispose(Boolean disposing)
{
if (disposing && components)
{
components->Dispose();
}
__super::Dispose(disposing);
}
private:
/// <summary>
/// Required designer variable.
/// </summary>
System::ComponentModel::Container * components;

    /// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
void InitializeComponent(void);
};
}
```

Режимы дизайна и кода

При создании нового проекта запускается режим дизайна — форма представляет собой основу для расположения элементов управления. Для работы с программой следует перейти в режим кода. Это можно сделать несколькими способами: щелкнуть правой кнопкой мыши в любой части формы и в появившемся меню выбрать View Code, в окне Solution Explorer сделать то же самое на компоненте Form 1.h или просто дважды щелкнуть на форме — при этом сгенерируется метод Form1_Load. После хотя бы однократного перехода в режим кода в этом проекте появится вкладка Form1.h , нажимая на которую, тоже можно переходить в режим кода. Для перехода в режим кода также можно использовать клавишу F7, а для возврата в режим дизайна — сочетание Shift+F7.

ОКНА ИНСТРУМЕНТОВ СРЕДЫ РАЗРАБОТКИ VISUAL STUDIO

Основные инструменты в среде разработки Visual Studio - это Solution Explorer (Обозреватель решений) , окно Properties (Свойства), Dynamic Help (Динамическая справка), Windows Forms Designer (Конструктор Windows Forms), Toolbox (Область элементов) и окно Output (Вывод). Есть также несколько специализированных инструментов, в их числе Server Explorer (Обозреватель серверов) и Class View (Представление классов). Они чаще всего представлены в виде закладок вдоль границ среды разработки или в нижней части окон инструментов, или вообще невидимы. Редко

используемые инструменты перенесены в подменю Other Windows (Другие окна). Если какой-либо инструмент не виден, то его можно отобразить при помощи команд меню или панели инструментов.

Точный размер и форма окон и инструментов зависят от настроек вашей среды разработки. Visual Studio позволяет располагать и закреплять окна так, чтобы сделать видимыми все необходимые элементы. Вы также можете скрыть инструменты так, что они примут форму закладок по краям среды разработки и останутся невидимыми до тех пор, пока они снова вам не понадобятся.

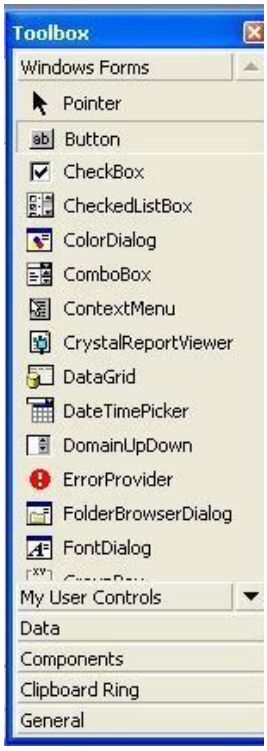
Скрывающиеся панели, расположенные по бокам окна можно выдвинуть, просто щелкнув на них. Мы можем закрепить их на экране, нажав на значок  , или совсем убрать с экрана, а затем снова отобразить, используя соответствующий пункт меню View (или эквивалентное сочетание клавиш).

Окно Toolbox

Окно Toolbox (панель инструментов, View —> Toolbox, или сочетание клавиш Ctrl+Alt+X) содержит компоненты , называемые также элементами управления, которые размещаются на форме. Элементы управления — это компоненты, обеспечивающие взаимодействие между пользователем и программой.

То, какой набор компонентов доступен в данный момент, зависит от типа разрабатываемого приложения. Например, если в данный момент разрабатывается приложение типа Windows Forms, в этом окне будут присутствовать элементы управления, которые можно использовать в Windows-приложениях; если же разрабатывается Web-форма, в этом окне будут находиться инструменты для работы с элементами управления Web Controls, и т.д. Наиболее часто употребляемой закладкой является Windows Forms. Для размещения нужного элемента управления достаточно просто щелкнуть на нем в окне Toolbox или, ухватив, перетащить его на форму.

Набор компонентов Windows Forms можно сгруппировать по нескольким функциональным группам.



Группа командных объектов Элементы управления Button, LinkLabel, ToolBar реагируют на нажатие кнопки мыши и немедленно запускают какое-либо действие. Наиболее распространенная группа элементов. **Группа текстовых объектов**
Большинство приложений предоставляют возможность пользователю вводить текст и, в свою очередь, выводят различную информацию в виде текстовых записей. Элементы TextBox, RichTextBox принимают текст, а элементы Label, StatusBar выводят ее. Для обработки введенного пользователем текста, как правило, следует нажать на один или несколько элементов из группы командных объектов.
Группа переключателей Приложение может содержать несколько предопределенных вариантов выполнения действия или задачи; элементы управления этой группы предоставляют возможность выбора пользователю. Это одна из самых обширных групп элементов, в которую входят ComboBox, ListBox, ListView,

TreeView, NumericUpDown и многие другие.

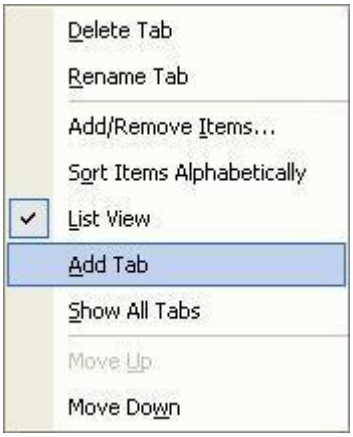
Группа контейнеров С элементами этой группы действия приложения практически никогда не связываются, но они имеют большое значение для организации других элементов управления, их группировки и общего дизайна формы. Как правило, элементы этой группы, расположенные на форме, служат подложкой кнопкам, текстовым полям, спискам — поэтому они и называются контейнерами. Элементы Panel, GroupBox, TabControl, кроме всего прочего, разделяют возможности приложения на логические группы, обеспечивая удобство работы.

Группа графических элементов Даже самое простое приложение Windows содержит графику — иконки, заставку, встроенные изображения. Для размещения и отображения их на форме используются элементы для работы с графикой — Image List, Picture Box.

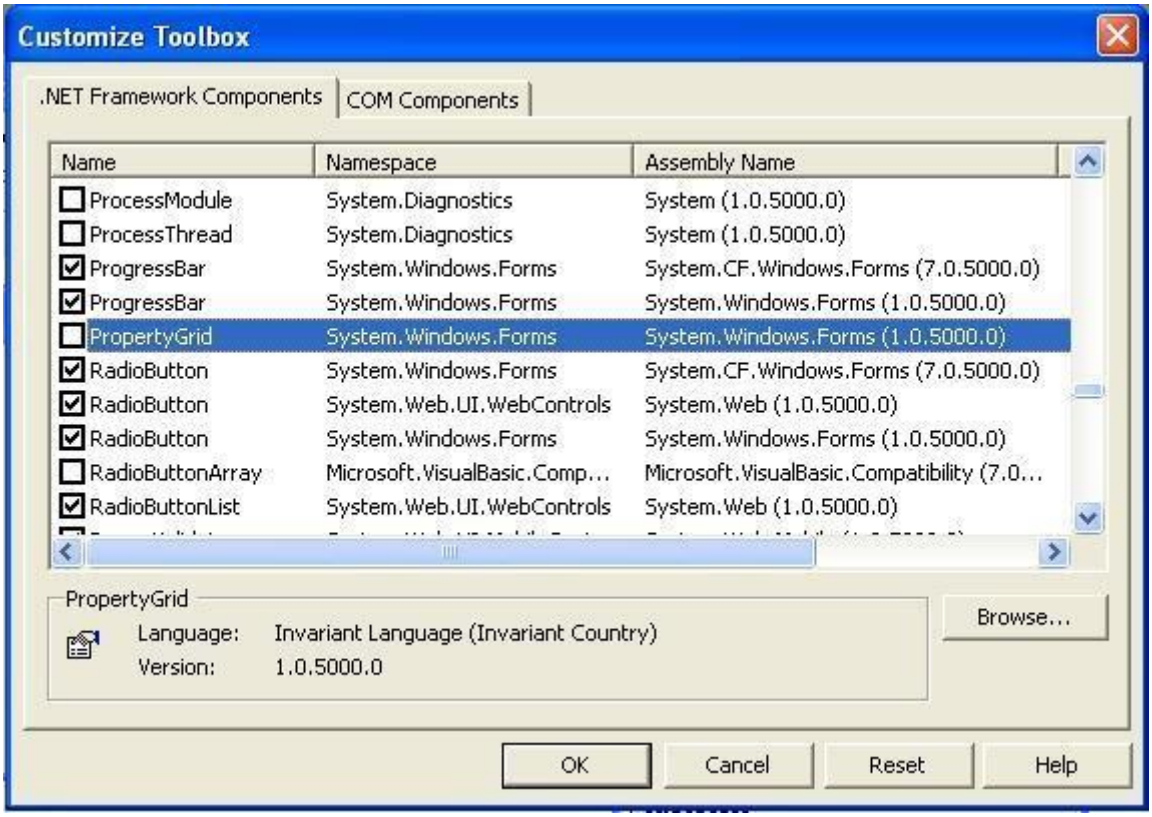
Диалоговые окна Выполняя различные операции с документом — открытие, сохранение, печать, предварительный просмотр, — мы сталкиваемся с соответствующими диалоговыми окнами. Разработчикам .NET не приходится заниматься созданием окон стандартных процедур: элементы OpenFileDialog, SaveFile Dialog, ColorDialog, PrintDialog содержат уже готовые операции.

Группа меню Меню обеспечивают доступ ко всем возможностям и настройкам приложения. Элементы MainMenu, ContextMenu представляют собой готовые формы для внесения заголовков и пунктов меню.

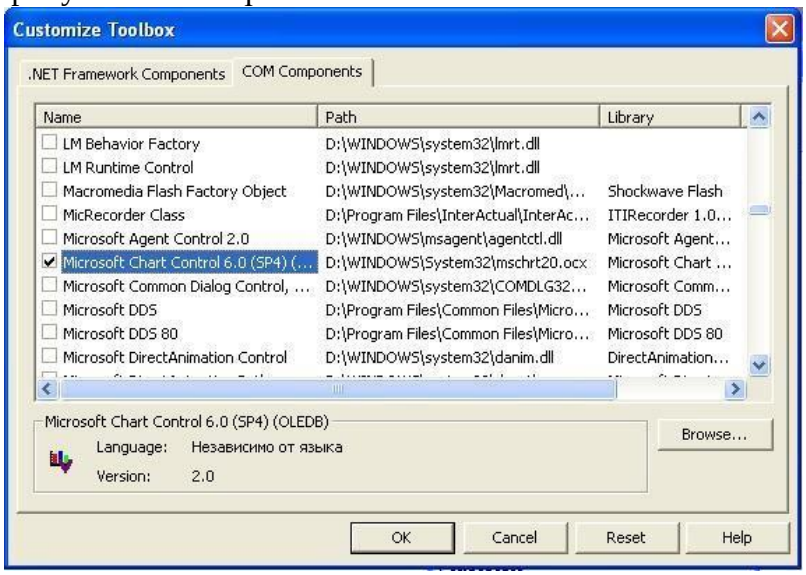
Контекстное меню содержит команды, с помощью которых можно управлять видом отображения групп компонентов, а также удалять и добавлять новые.



При необходимости можно изменить отображаемый в окне Toolbox набор элементов управления, добавив другие компоненты .NET или элементы ActiveX (в том числе созданные независимыми производителями). Для этой цели можно использовать команду меню Tools | Customize Toolbox и с помощью диалоговой панели Customize Toolbox выбрать элементы управления ActiveX или элементы управления .NET, которые мы хотим отобразить в окне Toolbox.



NET компонент PropertyGrid - это окно свойств , которое можно использовать в приложениях. На стадии выполнения программы PropertyGrid позволяет работать со свойствами элементов управления , расположенных на форме. Для того, чтобы связать PropertyGrid с элементов управления , необходимо его свойство SelectedObject установить равным имени элемента управления. Как видно из рисунка по умолчанию компонент PropertyGrid не отображается на панели Toolbox.



На вкладке COM Components содержаться доступные, зарегистрированные в реестре элементы ActiveX. Не зарегистрированные в реестре компоненты можно поместить в виде файлов dll в папку сборки приложения

Окно Toolbox состоит из нескольких закладок: My User Controls, Components, Data,

Windows Forms и General . Все доступные вкладки можно отобразить , используя команду контекстного меню Show All Tabs

Используя команду контекстного меню Add Tab можно создать свою собственную закладку и хранить собственные списки элементов управления. Наиболее часто используемые элементов управления имеет смысл перетащить на эту закладку

Переключение вида значков позволяет разместить их без полосы прокрутки, а также в виде списка.

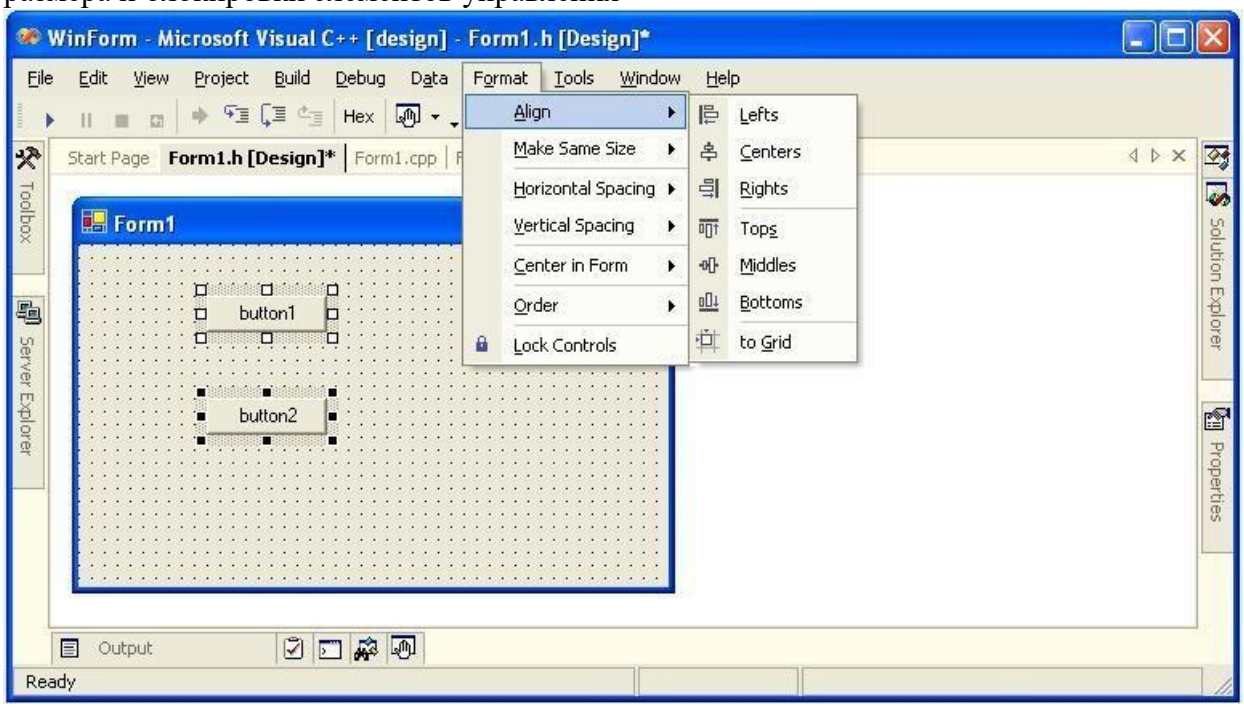
Созданные таким образом закладки можно переименовать или удалить, выбрав в

контекстном меню пункты Rename Tab и Delete Tab соответственно.

Все закладки, кроме Clipboard Ring и General, содержат компоненты, которые можно перетащить на форму. Закладка Clipboard Ring представляет собой аналог буфера обмена в Microsoft Office 2003, отображающего содержимое буфера за несколько операций копирования или вырезания. Для вставки фрагмента достаточно дважды щелкнуть по нему.

Форматирование элементов управления

Расположение элементов на форме во многом определяет удобство работы с готовым приложением. Пункт главного меню Format содержит опции выравнивания, изменения размера и блокировки элементов управления



Пункт главного меню Format

При выделении нескольких элементов управления около одного из них появляются темные точки маркера Свойства выбранных элементов будут изменяться относительно этого, главного элемента управления. Для последовательного выделения нескольких элементов удерживаем клавишу Shift, главным элементом будет последний выделенный элемент. Значение пунктов меню Format приводятся в таблице

Таблица

Пункт меню Format	Описание

Align	Выравнивание выбранных элементов управления
Make Same Size	Установка равного размера
Horizontal Spacing	Пробел между элементами по горизонтали
Vertical Spacing	Пробел между элементами по вертикали
Center in Form	Расположение элементов управления относительно формы
Order	Вертикальный порядок элементов управления
Lock Controls	Блокировка элементов

Окно свойств (Properties Window)

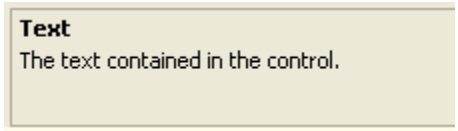
Окно свойств Properties — основной инструмент настройки формы и ее компонентов. Содержимое этого окна представляет собой весь список свойств выбранного в данный момент компонента или формы. При создании проекта, в окне Properties отображаются свойства самой формы. При активизации элемента управления на форме окно Properties автоматически покажет свойства и события, которые могут быть использованы с этим элементом управления.

Окно Properties содержит селектор объектов и две страницы свойств и событий. Выпадающие вниз список селектора объектов показывает имена и типы объектов каждого элемента управления в текущей форме, включая и сами форму. Селектор объектов используется для быстрого переключения между каждым из элементов управления.

В окне Properties (Свойства) есть две удобные кнопки, которые можно использовать для организации свойств. Кнопка Alphabetic (По алфавиту) сортирует все свойства в алфавитном порядке и сворачивает их в нескольких категориях. (Если вам известно имя свойства, и вы хотите его быстро найти, нажмите эту кнопку.) Кнопка Categorized (По категориям) распределяет свойства по различным логическим группам. (Если вы не знаете всех свойств настраиваемого объекта и хотите сгруппировать свойства по их типам, нажмите эту кнопку.)

В таблице приводится описание интерфейса окна Properties.

Таблица

Элемент	Изображение	Описание
Object name		В поле этого списка выводится название данного выбранного объекта, который является экземпляром какого-либо класса. Здесь Form1 — название формы по умолчанию, которая наследуется от класса System.Windows.Forms.Form
Categorized		При нажатии на эту кнопку производится сортировка свойств выбранного объекта по категориям. Можно закрывать категорию, уменьшая число видимых элементов. Когда категория скрыта, вы видите знак (+), когда раскрыта — (–)
Alphabetic		Сортировка свойств и событий объекта в алфавитном порядке
Properties		При нажатии на эту кнопку отображается перечисление свойств объекта
Events		При нажатии на эту кнопку отображается перечисление событий объекта
Description Pane		Панель, на которую выводится информация о выбранном свойстве. В данном случае в списке свойств формы

было выбрано свойство Text

Редакторы свойств

Каждый элемент управления обладает своим заранее определенным набором свойств. Visual Studio организует их по категориям и отображает в виде структуры. Окно Properties позволяет использовать несколько способов для показа свойств и изменения их значений. Эти механизмы называют редакторами свойств. Существует несколько типов редакторов свойств:

1. Строка ввода.
2. Редактор с выпадающим списком значений. Его признаком является стрелка вниз в колонке значений.
3. Диалоговая панель. Признак – кнопка с многоточием (…) в колонке значений.
4. Набор вложенных свойств. На существовании вложенного списка подсвойств указывает знак (+) в имени свойства.

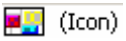
Свойств формы

Окно Properties позволяет определять в первую очередь дизайн формы и ее элементов управления. В таблице приводится описание некоторых свойств формы, обычно определяемых в режиме дизайна. При выборе значения свойства, отличного от принятого по умолчанию, оно выделяется жирным шрифтом, что облегчает в дальнейшем определение изменений.

Таблица . Некоторые свойства формы

Свойство	Описание	Значение по умолчанию
Name	Название формы в проекте. Это не заголовок формы, который вы видите при запуске формы, а название формы внутри проекта, которое вы будете использовать в коде	Form1, Form 2 и т.д.
AcceptButton	Устанавливается значение кнопки, которая будет срабатывать при нажатии клавиши Enter. Для того чтобы это свойство было активным, необходимо наличие по крайней мере одной кнопки, расположенной на форме	None
BackColor	Цвет формы. Для быстрого просмотра различных вариантов просто щелкайте прямо на названии "BackColor"	Control
BackgroundImage	Изображение на заднем фоне	None

CancelButton	Устанавливается значение кнопки, которая будет срабатывать при нажатии клавиши Esc. Для того чтобы это свойство было активным, необходимо наличие по крайней мере одной кнопки, расположенной на форме	None
ControlBox	Устанавливается наличие либо отсутствие трех стандартных кнопок в верхнем правом углу формы: "Свернуть", "Развернуть" и "Закрыть"	
Cursor	Определяется вид курсора при его положении на форме	Default
DrawGrid	Устанавливается наличие либо отсутствие сетки из точек, которая помогает форматировать элементы управления. В любом случае сетка видна только на стадии создания приложения	True
Font	Форматирование шрифта, используемого для отображения текста на форме в элементах управления	Microsoft Sans Serif; 8,25pt
FormBorderStyle	Определение вида границ формы. Возможные варианты: · None — форма без границ и строки заголовка; · FixedSingle — тонкие границы без возможности изменения размера пользователем; · Fixed3D — границы без возможности изменения размера с трехмерным эффектом; · FixedDialog — границы без возможности изменения, без иконки приложения; · Sizable — обычные границы: пользователь может изменять размер границ;	Sizable

	· FixedToolWindow — фиксированные границы, имеется только кнопка закрытия формы. Такой вид имеют панели инструментов в приложениях; · SizableToolWindow — границы с возможностью изменения размеров, имеется только кнопка закрытия формы	
Icon	Изображение иконки, располагаемой в заголовке формы. Поддерживаются форматы .ico	 (Icon)
MaximizeBox	Определяется активность стандартной кнопки "Развернуть" в верхнем правом углу формы	True
MaximumSize	Максимальный размер ширины и высоты формы, задаваемый в пикселях. Форма будет принимать указанный размер при нажатии на стандартную кнопку "Развернуть"	0;0 (Во весь экран)
MinimizeBox	Определяется активность стандартной кнопки "Свернуть" в верхнем правом углу формы	True
MinimumSize	Минимальный размер ширины и высоты формы, задаваемый в пикселях. Форма будет принимать указанный размер при изменении ее границ пользователем (если свойство FormBorderStyle имеет значение по умолчанию Sizable)	0;0
Size	Ширина и высота формы	300; 300
StartPosition	Определение расположения формы при запуске приложения. Возможны следующие значения: · Manual — форма появляется в верхнем левом углу экрана; · CenterScreen — в центре экрана; · WindowsDefaultLocation	WindowsDefaultLocation

	— расположение формы по умолчанию. Если пользователь изменил размеры формы, то при последующем ее запуске она будет иметь тот же самый вид и расположение; · WindowsDefaultBounds — границы формы принимают фиксированный размер; · CenterParent — в центре родительской формы	
Text	Заголовок формы. В отличие от свойства Name, это именно название формы, которое не используется в коде	Form1, Form 2 и т.д.
WindowState	Определение положения формы при запуске. Возможны следующие значения: · Normal — форма запускается с размерами, указанными в свойстве Size; · Minimized — форма запускается с минимальными размерами, указанными в свойстве MinimumSize; · Maximized — форма разворачивается на весь экран	Normal

События, на которые реагируют компоненты.

События, на которые может реагировать данный компонент, содержится в списке страницы событий окна Properties. Набор событий заранее определен для данного класса компонентов.

Примеры событий

On Change Происходит при изменении значения введеного в поле ввода.
On Click щелчок мыши на компоненте
On Dblclick двойной щелчок
On Mouse Down нажатие кнопки мыши когда курсор находится на компоненте
On Mouse Move перемещение курсора мыши над компонентом
On Mouse Up пользователь отпускает кнопку мыши при условии что курсор был на компоненте
On Enter компонент получает фокус ввода
On Exit теряет фокус ввода

On Key Down нажатие любой клавиши, когда компонент имеет фокус ввода
On Key Press нажатие ASCII клавиши
On Key Up пользователь отпускает нажатую клавишу

Обработка событий

Кнопка окна свойств Events (События)  переключает окно Properties в режим управления обработчиками различных событий (например, мыши, клавиатуры) и одновременно выводит список всех событий компонента.
Окно Properties в позволяет связать с событием функцию, которая будет вызываться в случае, если данное событие произойдет. Есть стандартный способ включения событий. Достаточно выделить нужный элемент в форме, в окне свойств нажать кнопку событий (со значком молнии) и из списка событий выбрать нужное событие и щелкнуть по нему. Двойной щелчок мыши на колонке значений вызывает генерацию заголовка функции для обработки события. Курсор распологается внутри фигурных скобок, где пользователь должен набрать свой собственный код. Обработчик событий может иметь параметры, которые указываются после имени в круглых скобках. Имя обработчика формируется из имени объекта, которое включает его порядковый номер на форме и названия события. Задать обработчик события для кнопки можно еще проще. Двойной щелчок по кнопке включает событие, и автоматически строится заготовка обработчика события с нужным именем и параметрами

Изменение свойств шрифта

· Нажмите кнопку многоточия в строке Font.

Visual Studio покажет диалоговое окно Font (Шрифт), в котором можно уточнить параметры шрифта для выбранной надписи. Каждому выбранному вами параметру соответствует отдельная строка в окне Properties (Свойства), и ее значение будет изменено соответствующим образом.

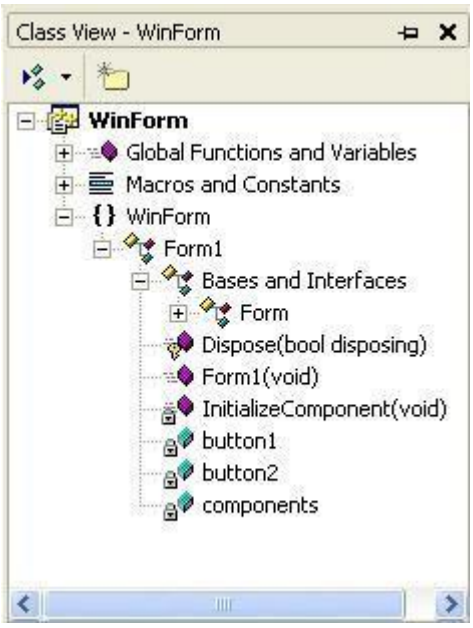
· Измените размер шрифта с 10 до 12 точек, а затем измените начертание с обычного на **курсив**. Чтобы подтвердить сделанные изменения, нажмите кнопку OK.

· Прокрутите окно Properties (Свойства) до свойства ForeColor, а затем щелкните мышкой в левом столбце.

В правом столбце нажмите кнопку раскрытия списка, а в нем выберите закладку Custom (Польз.) и на ней на синий цвет.

Окно проводника классов (Class View)

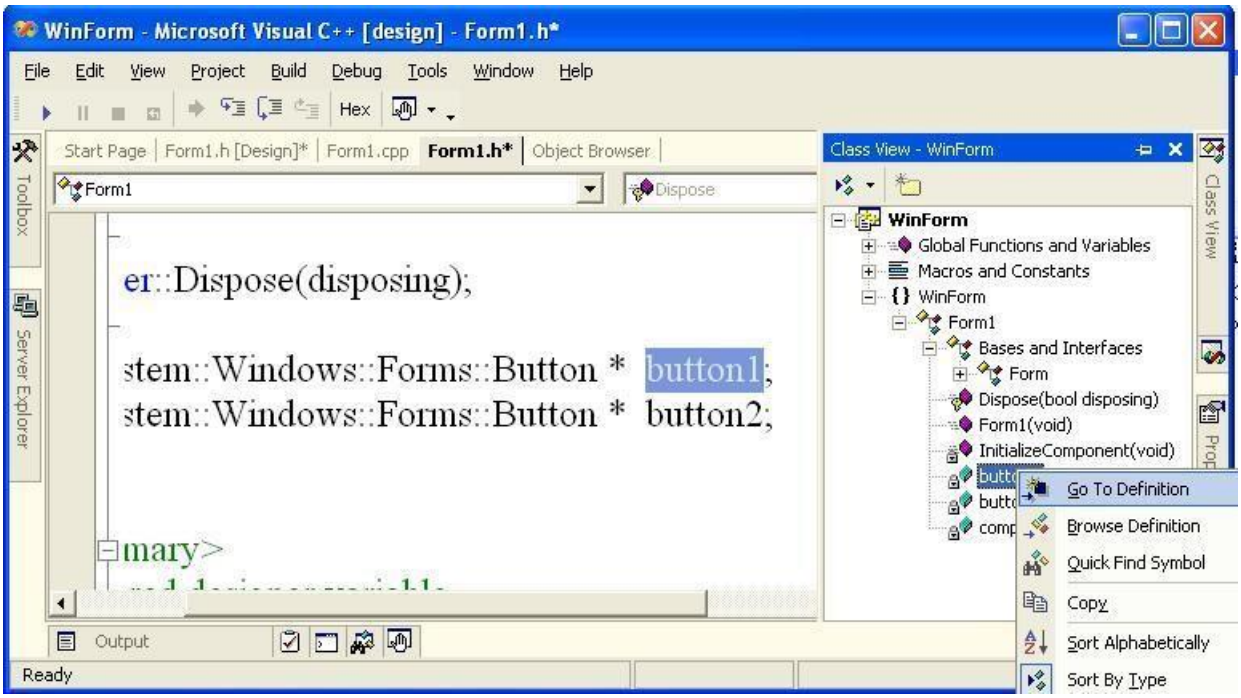
Для навигации внутри модуля используется проводник классов, содержащий в виде иерархической структуры все типы, классы, глобальные переменные и функции определённые в модуле.



Содержимое проводника классов синхронизировано с содержимым редактора кода. Изменения в коде отображаются в проводнике классов и, наоборот, изменения в проводнике приводят к соответствующимизменениям в коде приложения. ClassView позволяет при помощи drag&drop вытаскивать имена методов, свойств и классов напрямую в окно редактирования текста, . В Visual Studio .NET, Class View также содержит список методов и свойств базовых классов, что позволяет легко переходить к описанию этих методов и свойств в Object Browser (двойной щелчок по выбранному элементу списка).

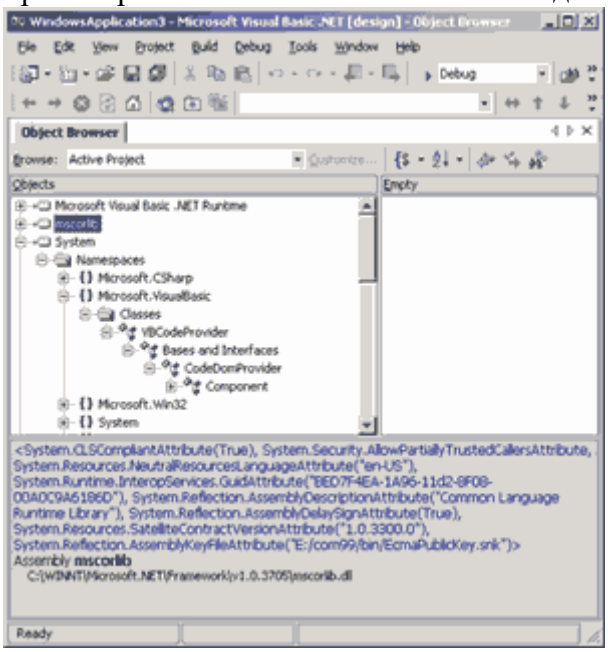
Добавить в проект новый класс

С помощью контекстного меню можно найти объявление и реализацию выбранного класса, а также осуществлять поиск определения функции.



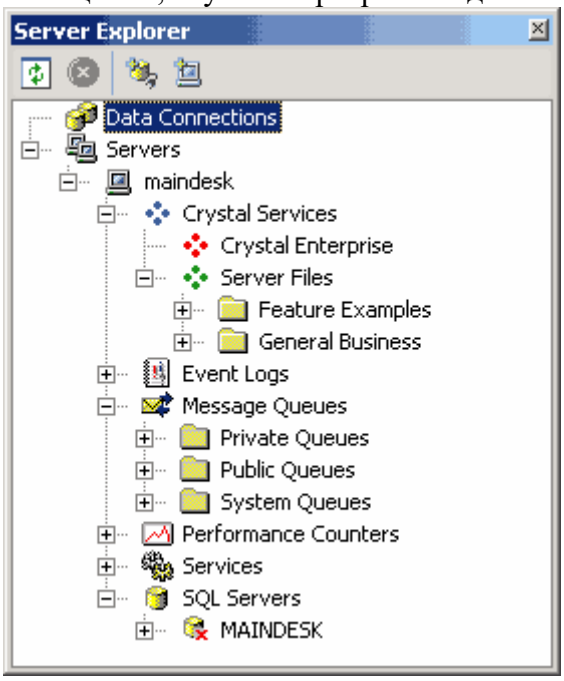
Окно Object Browser

Окно Object Browser, доступное с помощью команды меню View | Other Windows | Object Browser , так же как и окно Class View, позволяет просмотреть список классов, их свойств и методов. Однако Object Browser позволяет просмотреть все компоненты, на которые ссылается класс, а также, при необходимости, компоненты, на которые нет ссылок в данном проекте, тогда как с помощью окна Class View можно просматривать сведения только о классах из данного проекта. С помощью Object Browser можно также просмотреть объявления свойств и методов.



Окно Server Explorer

Окно Server Explorer (команда меню View | Server Explorer), позволяет просматривать сведения о службах, выполняющихся на конкретных серверах. К таким службам, в частности, относятся службы Crystal Reports Services, журнал событий, очереди сообщений, службы серверов баз данных, таких как Microsoft SQL Server.



Окно Server Explorer
Большинство этих служб представлено в окне в виде иерархического дерева, позволяющего просматривать сведения, связанные с данными службами, и иногда добавлять новые элементы. С помощью перетаскивания значка службы или ее элемента в дизайнер можно организовать ее использование в приложении. Так, при переносе значка таблицы сервера баз данных на форму разрабатываемого приложения можно создать компонент DataAdapter для извлечения данных из этой таблицы.

Окно динамическая справка (Dynamic Help)

В комплект Visual Studio .NET входит оперативное справочное руководство, в котором можно найти необходимые сведения о среде разработки Visual Studio, языках программирования ресурсах .NET Framework и остальных инструментах, входящих в набор Visual Studio.

Доступ к справочной информации

Инструмент Dynamic Help (Динамическая справка) , в зависимости от того, над чем вы работаете в данный момент, заранее подбирает справочные разделы и показывает их в своем окне. Для ограничения выбора материалов используется анализ контекста, и вы увидите только те из них, которые относятся к конкретному компилятору или инструменту. (Другими словами, в Dynamic Help (Динамической справке) не появятся заголовки, относящиеся к Visual C#, если вы не работаете с этими инструментами.)

Динамическая справка позволяет выполнять полнотекстовый поиск по справочной системе Visual Studio. Полнотекстовый поиск полезен, когда нужно разыскать текст по отдельным ключевым словам.

Получение справки с помощью Dynamic Help (Динамической справки)

1. Щелкните на закладке Dynamic Help (Динамическая справка) в среде разработки или выберите строку Dynamic Help (Динамическая справка) в меню Help (Справка). Появится окно динамической справки, показанное ниже.

Окно Dynamic Help (Динамическая справка) входит в состав Visual Studio. Его можно перемещать, изменять размер, закреплять или скрывать. Вы можете оставить его всегда открытым или открывать только тогда, когда оно нужно.

2. Выберите какой-нибудь пункт в окне динамической справки. Скорее всего, это будет текст про область элементов или про окно Properties (Свойства).

3. Просмотрите несколько других статей из списка динамической справки. Обратите внимание, насколько эти темы соответствуют реальным действиям, которые вы выполняете.

Отбор и показ справочной информации можно настроить. В меню Tools (Сервис) выберите команду Options (Параметры) и в окне свойств раскройте папку Environment (Среда). Затем выберите пункт Dynamic Help (Динамическая справка) и определите, какие темы вы хотите видеть в справке и сколько ссылок будет показано одновременно. Эти настройки помогут вам

Перемещение и изменение размеров окон инструментов

Когда на экране одновременно видны много инструментов программирования, среда разработки Visual Studio выглядит очень перегруженной. Visual Studio позволяет перемещать, изменять размеры, закреплять и использовать функцию автоматического скрытия для большинства элементов интерфейса, которые используются для создания программ.

Чтобы передвинуть любое из окон инструментов Visual Studio, просто щелкните на его строке заголовка и перетащите объект в другое место. Если вы придвинете одно окно к краю другого окна, то оно закрепится. Преимущество закрепляемых окон в том, что они всегда видны (они не скрываются за другими окнами). Если вы хотите увеличить закрепляемое окно, просто раздвиньте его границу.

Чтобы полностью закрыть окно, нужно нажать кнопку Close (Закрыть) в верхнем правом углу этого окна. Вы всегда сможете открыть это окно снова, выбрав соответствующую команду в меню View (Вид).

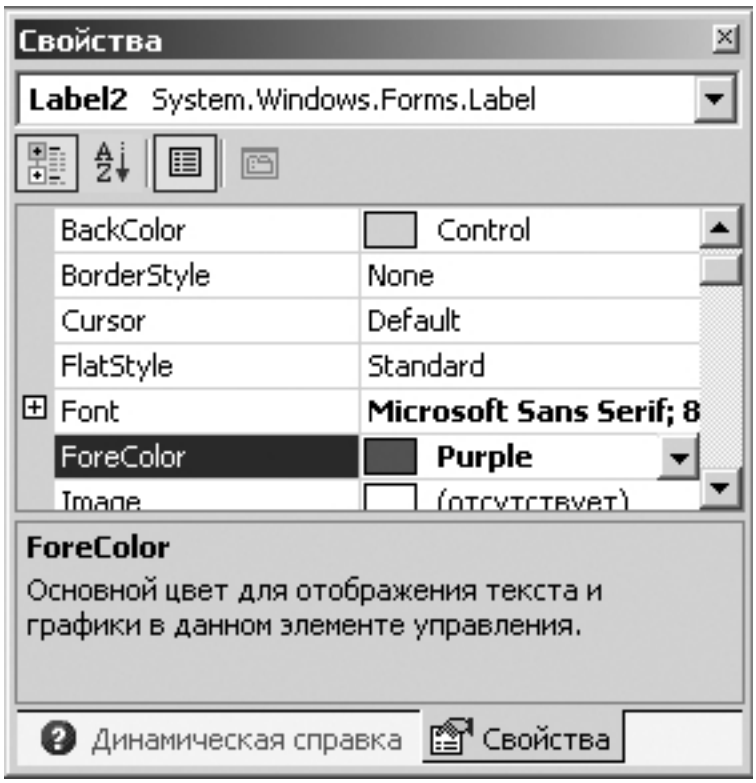
В среде разработки Visual Studio имеется функция автоскрытия окна инструмента, которую можно задействовать, щелкнув в правой части заголовка окна по канцелярской кнопке Auto Hide (Автоскрытие). Это действие убирает окно с его закрепленной позиции, а заголовок окна остается на краю среды разработки на закладке, которая не занимает много места. При выполнении автоскрытие окна, окно инструмента будет оставаться видимым до тех пор, пока вы держите на нем указатель мыши. Как только мышь перемещается в другую часть среды разработки, это окно свернется и превратится в закладку.

Чтобы восстановить окно, для которого активизировано автоскрытие, нажмите на его закладку на краю среды разработки или некоторое время подержите над ней указатель мыши. Определить, включена ли функция автоскрытия для данного окна, можно по канцелярской кнопке в его правом углу. Удерживая указатель мыши над заголовком, вы можете быстро вызвать скрытое окно, щелкнув на его закладке, проверить или задать требуемую информацию и затем передвинуть мышь, чтобы окно исчезло. Если вам нужно, чтобы окно было открыто постоянно, снова нажмите кнопку Auto Hide (Автоскрытие), чтобы острие канцелярской кнопки смотрело вниз. Окно останется видимым.

Перемещение и изменение размеров окна Properties (Свойства)

1. Если окно Properties (Свойства) не отображается в среде разработки, нажмите кнопку Properties Window (Окно свойств) на стандартной панели инструментов. При этом окно появится, а его заголовок будет подсвечен.

2. Чтобы сделать окно Properties (Свойства) плавающим (незакрепленным), дважды щелкните мышью на заголовке окна. Окно будет выглядеть так, как показано на следующем рисунке.



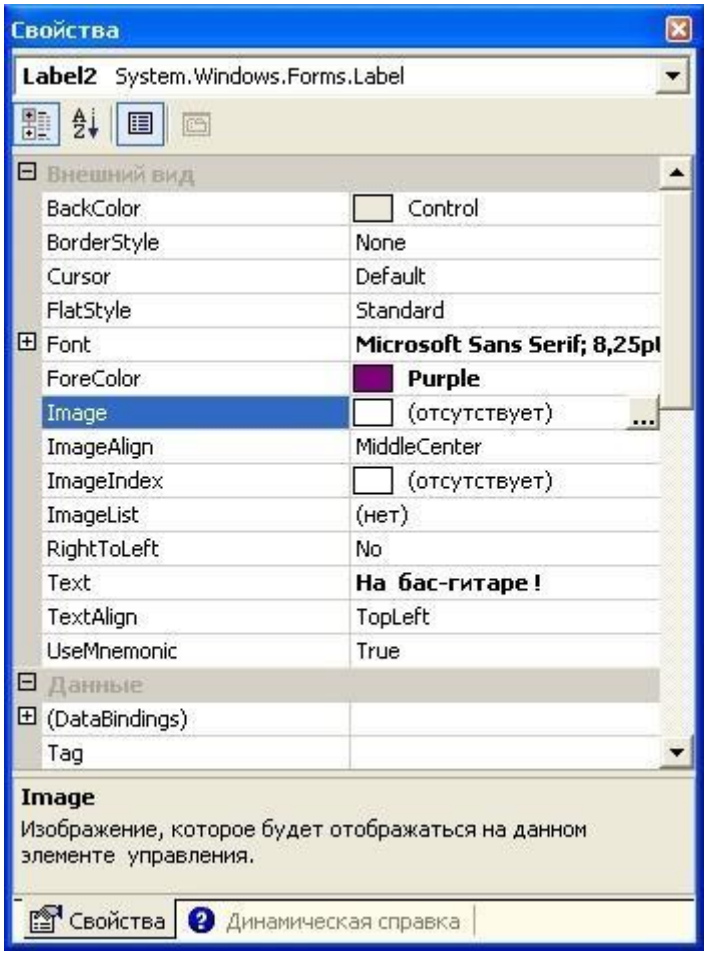
3. Используя строку заголовка окна Properties (Свойства), перетащите его в другое место в среде разработки, но не позволяйте ему закрепиться. Возможность перемещать окна инструментов в среде разработки Visual Studio позволяет организовать ее удобным для вас образом. Теперь давайте изменим размеры окна Properties (Свойства), чтобы увидеть больше свойств объекта.

4. Передвиньте указатель мыши к нижнему правому углу окна Properties (Свойства), так, чтобы он принял форму стрелки для изменения размера.

Размеры окон Visual Studio можно изменить тем же способом, что и окна других приложений в операционной системе Microsoft Windows.

5. Чтобы увеличить окно Properties (Свойства), потяните нижний правый угол окна вниз и вправо.

Теперь окно стало больше.



Окно большего размера позволяет работать быстрее и легче находить нужную информацию. Вы можете свободно изменять размеры окна, если необходимо увидеть большую часть его содержимого.

Закрепление инструмента в Visual Studio

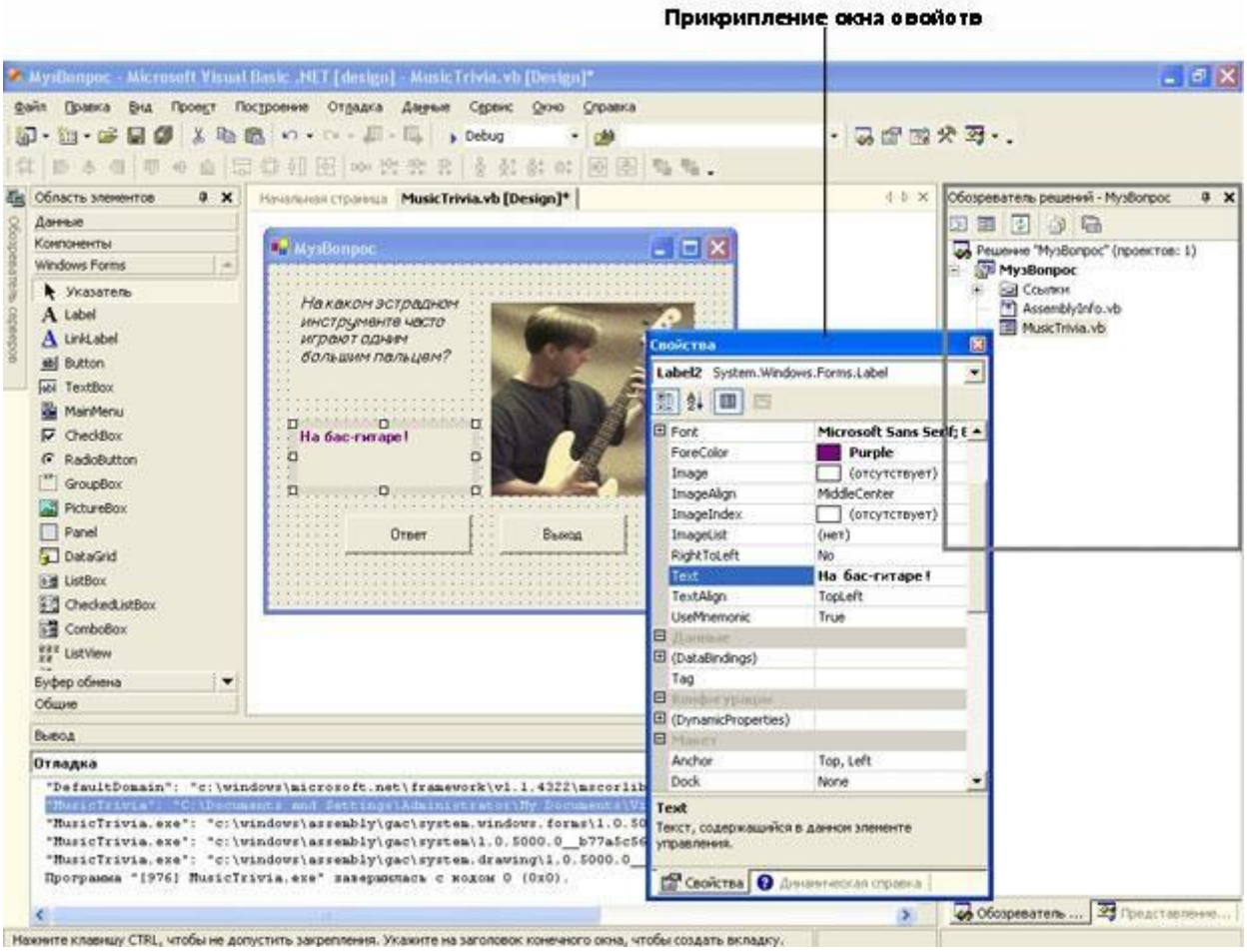
Если инструмент в среде разработки представлен плавающим окном, вы можете вернуть его в его первоначальное закрепленное положение, дважды щелкнув мышью на строке его заголовка. Когда окно находится в плавающем незакрепленном режиме, его можно закрепить в новом месте.

Закрепление окна Properties (Свойства)

1. Убедитесь, что окно Properties (Свойства) (или другой инструмент, который вы хотите передвинуть) представлено в среде разработки Visual Studio плавающим окном и не закреплено.

2. Перетащите окно Properties (Свойства) за его строку заголовка к верхнему, нижнему, правому или левому краю среды разработки, пока граница окна не "прилипнет" к краю окна среды разработки.

3. Чтобы закрепить окно Properties (Свойства), отпустите кнопку мыши. Окно будет находиться в том месте, где вы его оставили.



Чтобы предотвратить закрепление при перетаскивании окна, нажмите и удерживайте при перетаскивании клавишу (Ctrl). Если вы хотите, чтобы передвигаемое окно было совмещено с другим окном в виде закладки, перетащите это окно непосредственно на строку заголовка другого окна. Когда окна связаны друг с другом подобным образом, в нижней части их общего окна появится строка с закладками для каждого из них, и вы сможете переключаться между этими окнами, выбирая их по закладкам. Окна с закладками позволяют эффективно использовать пространство одного окна для двух и более задач. (Например, часто объединяют в виде закладок окна Solution Explorer (Обозреватель решений) и Class View (Представление классов).)

Скрытие инструмента в Visual Studio

В Visual Studio .NET имеется механизм для быстрого скрытия и отображения инструментов, который называется автоскрытие. Функция автоскрытия доступна для большинства окон инструментов. Преимущество автоскрытия для окон заключается в том, что при этом они освобождают значительное место в рабочей области Visual Studio, оставаясь при этом доступными.

Чтобы скрыть окно инструмента, нажмите кнопку Auto Hide (Автоскрытие) в правой части заголовка окна, что приведет к сворачиванию его в закладку на краю среды разработки. А чтобы закрепить окно, нажмите кнопку автоскрытия еще раз. Те же действия можно выполнить с помощью команды Auto Hide (Автоскрытие) из меню Window (Окно). Заметьте, что функция и кнопка автоскрытия доступны только для закрепленных окон. Вы не увидите команды Auto Hide (Автоскрытие) или этой кнопки

для активного окна, плавающего поверх среды разработки.

Использование функции автоскрытия

1. Найдите в среде разработки Toolbox (Область элементов) (ее окно обычно открыто в левой части Конструктора Windows Forms). В области элементов находятся элементы управления, которые можно использовать для создания приложений .

2. Найдите кнопку Auto Hide (Автоскрытие) на заголовке окна Toolbox (Область элементов). Канцелярская кнопка на ней нарисована в вертикальном, или приколотом, положении, что означает, что область элементов "приколота" в открытом положении и автоскрытие отключено.

3. Нажмите кнопку автоскрытия в строке заголовка и удерживайте указатель мыши в окне области элементов. Кнопка автоскрытия теперь выглядит как канцелярская кнопка, которая смотрит влево. Это говорит о том, что область элементов больше не "приколота" в открытом положении и будет работать автоскрытие. На левой стороне среды разработки появилась закладка с надписью Toolbox (Область элементов). Также обратите внимание, что Конструктор Windows Forms сдвинулся влево. Однако если указатель мыши все еще находится в рамках окна области элементов, в самой области элементов ничего не изменится. Разработчики Visual Studio решили, что будет лучше, если окно с включенным автоскрытием не будет исчезать до тех пор, пока пользователь не передвинет указатель мыши в другую область среды разработки.

4. Передвиньте мышь из окна Toolbox (Область элементов). Как только вы это сделаете, Toolbox (Область элементов). скроется за пределы экрана и останется только маленькая закладка. (Над закладкой области элементов также видна закладка Server Explorer (Обозреватель серверов) - указание на то, что автоскрытие включено еще для одного окна инструмента. В зависимости от настроек Visual Studio, могут быть видны и другие закладки для окон среды разработки, для которых включено автоскрытие.)

5. Подержите указатель мыши над закладкой Toolbox (Область элементов). (При желании можно щелкнуть на этой закладке.)

Ее окно немедленно выдвинется обратно в пределы видимости, и вы сможете использовать элементы управления для создания интерфейса пользователя.

6. Передвиньте указатель мыши за пределы области элементов, и инструмент снова исчезнет.

7. Наконец, снова раскройте область элементов, а затем нажмите кнопку автоскрытия в строке ее заголовка. Область элементов вернется в знакомое закрепленное положение, и вы сможете использовать ее постоянно.

Лекция 6

Структуры данных. Стек. Очередь. Дек

Контейнерные классы. Стандартная библиотека шаблонов STL. Последовательные и ассоциативные контейнеры. Итераторы. Последовательные контейнеры: векторы, двусторонние очереди, списки, очереди, стеки, очереди с приоритетами. Примеры задач.

Общее представление

Перед написанием программы важно правильно выбрать структуру данных, обеспечивающую эффективное решение задачи. Одни и те же данные можно сохранить в структурах, требующих различного объема памяти, а алгоритмы работы с каждой структурой могут иметь различную эффективность. Если выбрана наиболее подходящая структура и вы в совершенстве владеете методами работы с ней, то разработка алгоритма уже не вызовет затруднений, а сам алгоритм решения будет оптимален и по объему занимаемой памяти, и по времени работы алгоритма. Структуры данных определяет *объекты*, организованные определенным образом и *операции*, которые можно выполнять над объектами. Доступ к объектам и все операции с ними осуществляются только через интерфейс. Интерфейс отделяет реализацию структуры данных от клиента, и является «непрозрачным» для клиента. Структура данных может быть представлена как некий «черный ящик» с интерфейсами.

Контейнерные классы предназначены для хранения данных, организованных определенным образом. Для каждого типа контейнера определены методы работы с его элементами, не зависящие от конкретного типа данных, которые хранятся в контейнере. Поэтому один и тот же вид контейнера можно использовать для хранения данных различных типов. Возможность работы с контейнерными классами реализована с помощью стандартной библиотеки шаблонов (STL Standard Template Library), в которую входят контейнерные классы, алгоритмы и итераторы. Использование STL позволяет повысить надежность программ и уменьшить сроки их реализации.

Контейнеры можно разделить на два типа: *последовательные* и *ассоциативные*. *Последовательные контейнеры* обеспечивают хранение конечного количества однотипных элементов в виде непрерывной последовательности. К последовательным контейнерам относятся: векторы (*vector*), двусторонние очереди (*deque*), списки (*list*), а также так называемые адаптеры (варианты контейнеров): стеки (*stack*), очереди (*queu*) и очереди с приоритетами (*priority_queue*). Каждый вид контейнера обеспечивает свой набор действий над данными и выбор того или иного контейнера зависит от того, что именно требуется делать с данными в программе. Ассоциативные контейнеры обеспечивают быстрый доступ к данным по ключу. Такие контейнеры построены на основе сбалансированных деревьев. К ассоциативным контейнерам относятся: словари (map), словари с дубликатами (multiset), множества (set), множества с дубликатами (multiset) и битовые множества (bitset).

Каждая структура предоставляет алгоритмы, работающие с данными с той или иной сложностью. **Сложность** понимается как количество элементарных операций, выполняемых вычислительной машиной для решения задачи в зависимости от размера задачи. Размер задачи определяется входными данными. Так, например, для простейшей операции сложения двух натуральных чисел, размером задачи можно считать максимальную длину в битах одного из слагаемых. Для оценки сложности алгоритмов применяется верхняя оценка сложности, выражаемая в O –символике. Алгоритм имеет сложность O(n), если время его работы (количество элементарных операций) есть величина <=cn (меньшая или равная cn), где c – некоторая постоянная. Например, поиск элементов в одномерном массиве имеет сложность O(n) – линейную сложность. «Пузырьковая сортировка» массива имеет сложность O(n^2) – квадратичную сложность. O(1) – сложность, представляющая собой константу. Чем меньше сложность, тем быстрее работает алгоритм.

Двусторонняя очередь (дек) – последовательный контейнер, поддерживающий доступ к произвольным элементам и обеспечивает вставку и удаление из обоих концов очереди за постоянное время. Операции с элементами внутри очереди занимают время, пропорциональное количеству перемещаемых элементов. Доступ к элементам очереди осуществляется за постоянное время (оно несколько больше, чем для вектора).

Список – последовательный контейнер, обеспечивающий вставку и удаление элементов за постоянное время. Не предоставляет произвольный доступ к своим элементам. Операции с элементами внутри списка (вставка элемента, удаление элемента) занимают постоянное время.

Стек – последовательный контейнер, обеспечивающий вставку элемента в вершину стека и удаление элемента из вершины стека.

Очередь – последовательный контейнер, обеспечивающий добавление элементов в конец очереди и извлечение элементов с начала очереди.

Очередь с приоритетом – структура, реализованная при помощи очереди на основе контейнера, допускающего произвольный доступ к элементам (например, вектора или двусторонней очереди). Первым параметром при описании очереди с приоритетами является тип ключа, вторым последовательный контейнер, третьим – функция определения приоритета.

STL определяется в следующих заголовочных файлах: algorithm, deque, functional, iterator, list, map, memory, numeric, queue, set, stack, utility, vector.

Итераторы

Итератор является аналогом указателя на элемент и используется для просмотра контейнера в прямом и обратном направлении. Итератор ссылается на элемент и реализует операцию перехода к следующему элементу. Константные операторы используются в том случае, когда значения соответствующих элементов контейнера не изменяются. В таблице приведены варианты итераторов и методы доступа к элементам при помощи итераторов.

Поле	**Пояснение**
iterator	Итератор
const_iterator	Константный итератор
reverse_iterator	Обратный итератор
const_reverse_iterator	Константный обратный итератор
Методы доступа	
iterator begin() const_iterator begin()	Указывает на первый элемент
iterator end() const_iterator end()	Указывает на элемент, следующий за последним
reverse iterator rbegin() const_reverse_iterator rbegin()	Указывает на первый элемент в обратной последовательности
reverse_iterator rend() const_reverse_iterator rend()	Указывает на элемент, следующий за последним в обратной последовательности

В каждом контейнере эти типы и методы определяются способом, зависящим от их реализации. Для итераторов реализованы операции ++, -- после которых итератор указывает на следующий, предыдущий элемент в контейнере в порядке обхода. Доступ к элементу осуществляется при помощи операции разадресации *.

Общие методы и алгоритмы для последовательных контейнеров

Для работы с последовательными контейнерами используется ряд общих для них методов, предназначенных для получения сведения о размере контейнера или выполнения тех или иных операций с контейнером.

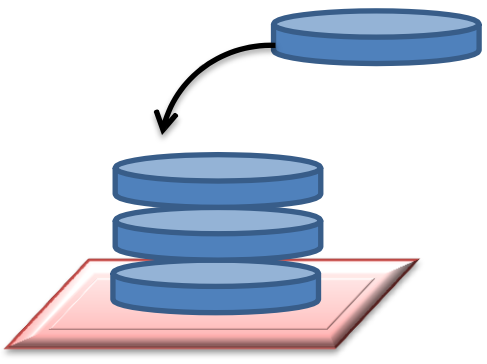
Методы получения сведений о размере контейнера	
size()	Получение сведений о размере контейнера (число элементов)
max_size()	Максимальный размер контейнера
empty()	Логическая функция, показывающая – пуст ли контейнер
Методы работы с элементами контейнера	
push_front	Вставка в начало дека

pop_front	Удаление из начала дека
push_back	Вставка в конец дека
pop_back	Удаление из конца дека
push	Вставить элемент в вершину стека или очереди
pop	Удалить элемент из вершины стека или очереди
top	Посмотреть элемент в вершине стека или очереди
insert	Вставка элемента во множество или словарь
erase	Удаление элемента из множества или словаря. Удаление элементов в диапазоне итераторов
at []	Произвольный доступ к элементу в векторе или двусторонней очереди
Алгоримы	
clear()	Очищает контейнер (для всех контейнеров)
count (it1, it2, value)	Считает количество вхождений элемента value в указанное множество, задаваемое двумя итераторами.
find (it1, it2, value)	Находит итератор, соответствующий элементу, равному value. Возвращает итератор на элемент или end() если элемент не найден
min_element(it1, it2) max_element(it1, it2)	Находит итератор, соответствующий минимальному (максимальному) элементу
binary_search(it1, it2, value)	Возвращает true, если элемент найден в указанном множестве и false если такого элемента нет
lower_bound(it1, it2, value) upper_bound(it1, it2, value)	Возвращает итератор, соответствующий первому/последнему элементу в множестве [it1, it2), перед которым можно вставить элемент value, сохраняя отсортированность множества.
sort(it1, it2) stable_sort(it1, it2)	Сортирует элементы Сортирует элементы, сохраняя относительный порядок элементов
replace(it1, it2,value1,value2)	Заменяет все вхождения value1 на value2 в указанном множестве, задаваемым итераторами it1, it2
reverse(it1, it2)	Меняет порядок на противоположный
merge(it11, it12, it21,it22, out_it)	Сливает два отсортированных множества, задаваемых итераторами первого и второго множества, копируя элементы в новое множество. Возвращает итератор end() нового множества a[3]={2,3,4}; b[3]={3,4,5}; *c; merge(a,a+3,b,b+3,c); Получим c={2,3,3,4,4,5}
unique(it11, it12)	Сжимает отсортированное множество, удаляя одинаковые элементы
random_shuffle(it11, it12)	Случайным образом перемешивает элементы множества
next_permutation(it11, it12) prev_permutation(it11, it12)	Генерирует следующую (предыдущую) лексикографическую последовательность множества. Возвращает true, если такая нашлась и false в противном случае.

Для понимания логики построения той или иной структуры полезным упражнением является самостоятельная реализации структур, например стека при помощи массива.

Реализация структуры стек при помощи статического массива

Стек можно представить себе как стопку дисков (см. рисунок), на которую сверху можно класть диски, и брать их можно тоже только сверху.



Стек (*stack)* является структурой данных, поддерживающей две операции: добавление элемента в вершину стека push (value) и удаление элемента из вершины стека pop (value). Дисциплина работы стека обозначается LIFO, последним пришел — первым ушел (Last In First Out). Стек можно реализовать при помощи массива.

```
#include<iostream>
#include<new.h>
using namespace std;
const int MAX_SIZE=10; //Количество элементов в массиве (Стеке)

struct my_stack //Описываем структуру стек
{
     int data[MAX_SIZE];//Массив с количеством элементов MAX_SIZE
     int last;          //Указатель на вершину стека
};
void push(my_stack &s, int x)//Процедура добавления элемента
{
     if (s.last==MAX_SIZE){ //Сообщение о переполнении стека
          cout<<"Stack Overflow";
          return;}
     s.data[s.last++]=x;
}
int pop(my_stack&s) //Получение элемента с вершины стека
{
     return s.data[--s.last];//Возвращаем элемент с вершины стека
}

int main()
{
     my_stack a;
     a.last=0;
     push(a,3);
     push(a,6);
     push(a,2);
     cout<<pop(a)<<" "; //Программа выведет 2
     cout<<pop(a)<<" ";//Программа выведет 6
     cout<<pop(a)<<" ";//Программа выведет 3
     return 0;
}
```

Как мы видим, заполняя стек методом push , мы помещаем каждый новый элемент в конец массива (Вершину стека). Указателем на вершину стека служит поле last в структуре стека. Снимая элемент с вершины стека методом pop в нашей реализации, мы возвращаем значение элемента, а затем удаляем этот элемент (уменьшая указатель last на единицу). Сложность каждой описанной операции *O(1)*. Таким образом, обращаясь к вершине стека после его заполнения, мы разворачиваем последовательность элементов. *Стек можно применять для разворота последовательности элементов*. Приведенную программу логично дополнить проверкой стека на наличие в нем элементов – методом empty(). Функция empty() вернет true, если стек пуст и false в противном случае.

```
bool empty(my_stack &s)
{
     return (s.last==0);
}
```

Метод size() позволяет определить количество элементов в стеке.

```
int size(my_stack &s)
{
     return (s.last);
}
```

Метод top() позволяет обращаться к вершине стека – возвращать элемент, находящийся в вершине стека, но при этом не удалять его (в отличие от метода pop()).

```
int top(my_stack&s)
{
     int i=s.last-1;
     return s.data[i];
}
```

Полезным упражнением может быть также реализация алгоритмов работы со стеком, например, count или find (перечень алгоритмов работы с контейнерами приведены в таблице в начале лекции).

Задачи, решаемые при помощи стека

1. Грамматика правильной скобочной последовательности.

Правильная скобочная последовательность – последовательность, состоящая из символов – «скобок», в которой каждой отрывающейся скобке соответствует закрывающаяся скобка такого же типа, что и открывающаяся скобка. Например, правильными будут следующие последовательности: [([])((([[[]]])))]{()}, ()((()))[[]]. Не будут являться правильными скобочные последовательности [[)) (несоответствие типа закрывающихся скобок типу открывающихся), }{ (закрывающая скобка стоит раньше открывающейся), [[{{}}] (не каждой открывающейся скобке соответствует закрывающаяся).

Решение задачи для скобочной последовательности, состоящей из одного типа скобок, заключается в вычислении баланса скобочной последовательности. Каждой открывающейся скобке ставим в соответствие число +1, каждой закрывающейся скобке число – 1. Считаем баланс последовательности balans, двигаясь слева направо. Если баланс в процессе подсчета станет равным отрицательному числу - balans<0, то это означает, что не для всех закрывающихся скобок в последовательности имелись открывающиеся и последовательность не является правильной. По достижении конца строки баланс должен быть равным нулю – balans=0. В этом случае последовательность правильная, иначе – неправильная.

Решение задачи для скобочной последовательности, состоящей из скобок различного типа, удобно выполнить при помощи структуры стек. При движении слева направо по строке в стек заносятся открывающиеся скобки. При добавлении в стек закрывающейся скобки проверяем наличие в вершине стека открывающейся скобки такого же типа. Если таковая скобка в вершине стека есть, то очередная скобка не добавляется, а имеющаяся в вершине удаляется. Рассмотрим пример для правильной скобочной последовательности [([])(([]))].

Последовательность	[	(	[	]	)	(	(	[	]	)	)	]
Состояние стека	[	[(	[([	[(	[	[(	[((	[(([	[((	[(	[	

В конце работы программы стек оказывается пустым – в этом случае скобочная последовательность правильная. Как видим, задача проверки скобочной последовательности на правильность, имеет линейную сложность O(n). Приведем пример программы, определяющей является ли правильной скобочная последовательность.

Программа использует стек STL.

```
#include<iostream>
#include<stack>
#include<string>
using namespace std;
int main()
{
     string s;
     stack<char> b;
     cin>>s;
     for(auto c=s.begin(); c!=s.end();c++)
     {
          if (*c==')'&&b.top()=='('|| *c==']'&&b.top()=='['||
*c=='}'&&b.top()=='{')
               b.pop();
          else if (*c==')'||*c==']'||*c=='}')
               {
                    cout<<"NO";
                    return 0;
               }
          else
               b.push(*c);
     }
     b.empty()?cout<<"YES":cout<<"NO";
     return 0;
}
```

2. Вычисления арифметических выражений.

Арифметическое выражение состоит из чисел, знаков арифметических действий и скобок. Например, рассмотрим арифметическое выражение (7+6)/(26-13). Его можно записать и других формах. *Префиксная форма записи* арифметического выражения представляет выражение таким образом, что знаки арифметических операций предшествуют операндам, над которыми эти операции выполняются. А *постфиксная форма записи (обратная польская нотация)* представляет выражение таким образом, что знаки арифметических операций указываются после операндов.

Арифметическое выражение	**(7+6)/(26-13)**
Префиксная форма записи	**/ + 7 6 – 26 13**
Постфиксная форма записи	**7 6 + 26 13 - /**

Алгоритм вычисления значения арифметического выражения, записанного в постфиксной форме, имеет линейную сложность – O(n). Алгоритм использует стек. При чтении выражения слева направо в вершину стека помещаются операнды. Как только при чтении встречается знак арифметической операции, из стека извлекаются два последних операнда, к ним применяется текущая операция, и результат записывается обратно в вершину стека. По завершении работы алгоритма в стеке оказывается один элемент – значение арифметического выражения.

Рассмотрим алгоритм вычисления арифметического выражения, использующий в неявном виде обратную польскую нотацию. Заводим два стека. Один – для чисел, второй для знаков арифметических операций и скобок. Читаем выражение слева направо, и, встретив число, помещаем его в первый стек. Если текущий символ – закрывающаяся скобка, то выполняем вычисления до тех пор, пока не встретим парную ей открывающуюся скобку. Если текущий символ – знак арифметической операции и на вершине стека операции с таким же или большим приоритетом, то выполняем все необходимые для нее действия. По завершении алгоритма в стеке операций могут остаться еще не выполненные операции, их надо выполнить, как было описано выше. Рассмотрим некоторые идеи для написания программы, вычисляющей значение арифметического выражения при условии, что арифметическое выражение имеет правильный формат записи. Прежде всего, договоримся, что исходная строка – арифметическое выражение,

текущая позиция строки, два стека **num** - для хранения чисел и **op** - знаков арифметических операций и скобок будут объявлены как глобальные переменные.

При проходе по строке слева направо нам требуется уметь определять – является ли текущий символ цифрой или знаком операции, скобкой. В этом помогут следующие функции.

```
bool LT_END() //Функция определяет тип лексемы - конец строки
     {return (cur_pos>=s.size());}
bool LT_N()    //Тим лексемы - цифра
     {return (s[cur_pos]>='0'&&s[cur_pos]<='9');}
bool LT_OB(char c) //Тип лексемы - открывающаяся скобка
     {return (c=='(');}
bool LT_CB(char c) //Тип лексемы - закрывающаяся скобка
     {return (c==')');}
bool LT_OP(char c) //Тип лексемы - знак арифметической операции
     {return(c=='+'||c=='-'||c=='*'||c=='/');}
```

Как только в строке встретится символ – цифра, необходимо «собрать» все число целиком. Для этого предусмотрим функцию `get_num()`.

```
int get_num()
{
     int value=0;
     while (!LT_END()&&LT_N()){
          value=value*10+(s[cur_pos]-48); //Собираем число
          cur_pos++;}
     cur_pos--;
     return value;
}
```

При выполнении арифметических действий необходимо учитывать приоритет арифметических операций. Более высокий приоритет имеют операции * и /.

```
int pr(char c)
{
     if (c=='+'||c=='-') return 1;
     if (c=='*'||c=='/') return 2;
}
```

Для вычисления простейших арифметических выражений вида: <число> <знак арифметической операции> <число> реализуем функцию get_res(), получающую на вход операцию, которая будет произведена с двумя числами, находящимися в вершине стека **num** с числами. После этого эти числа удалятся, а полученный результат будет добавлен в вершину стека **num**.

```
void get_res(char op){
     int r=num.top(); num.pop();
     int l=num.top(); num.pop();
     switch (op){
     case '+': num.push(l+r); break;
     case '-': num.push(l-r); break;
     case '*': num.push(l*r); break;
     case '/': num.push(l/r); break;
     }
}
```

Наконец, функция calc() принимает на вход исходную стоку – арифметическое выражение, и анализируя символы этой строки, делает все необходимые действия алгоритма.

```
int calc(string s){
     while(!LT_END()){
//Если встретили закрывающуюся скобку, то вычисляем выражение
между скобками, пока не встретим открывающуюся скобку
          if (LT_CB(s[cur_pos])){
               while (!LT_OB(op.top())){
                    get_res(op.top());
                    op.pop();
               }
               op.pop();
          }
//Открывающуюся скобку сразу заносим в стек op
          else if (LT_OB(s[cur_pos]))
               op.push(s[cur_pos]);
//Если встретили арифметическую операцию, то пока в стеке
операций находится операция с большим или равным ей приоритетом,
выполняем арифметические действия с числами стека num
          else if (LT_OP(s[cur_pos])){
          char cur_op=s[cur_pos];
          int cur_pr=pr(s[cur_pos]);
          while (!op.empty()&&LT_OP(op.top())&&pr(op.top())>=cur_pr){
                    get_res(op.top()); op.pop();
               }
               op.push(cur_op);
          }
          else if (LT_N())
               num.push(get_num());
          cur_pos++;
     }
     while (!op.empty()){ //В конце в стеках могут остаться
данные – необходимо «довычислить» выражение.
                    get_res(op.top()); op.pop();
               }
     return num.top();
}
```

3. Ближайший меньший слева и справа.

Дан массив чисел. Требуется вывести ближайший меньший слева и справа для данного элемента. Например, для массива a[9]={6 5 9 8 7 1 2 3 5} и числа 7 ближайшим меньшим слева

будет 6 с индексом 2, а ближайший меньший справа будет 1 с индексом 6. Будем искать ближайший меньший справа. Начинаем последовательность брать элементы с конца массива и записывать их индексы в стек.

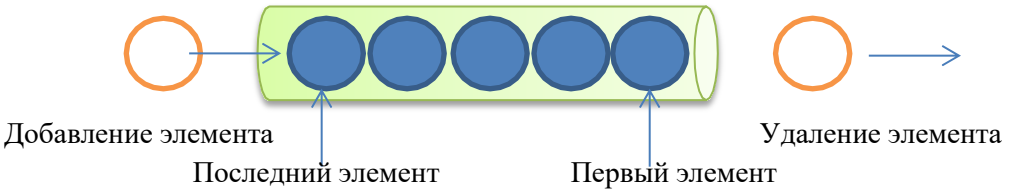
На первой итерации добавим в стек (9) – индекс последнего элемента в массиве. На второй итерации запоминаем (8) – индекс предпоследнего элемента в массиве. Обращаемся к вершине стека. Пока на вершине стека индексы элементов, которые больше, чем текущий или равны ему, пропускаем эти элементы, удаляя их из стека. Запоминаем ответ (вершину стека) и добавляем новое число (индекс просматриваемого элемента) в стек. Продолжаем алгоритм далее, пока не получим ответ для последнего числа. Таким образом, сложность алгоритма линейная – **O(n).**

1 итерация. Стек	9	Добавляем в стек индекс последнего элемента массива. В ответ для него записываем -1.							
2 итерация. Стек	8		Берем из массива элемент 3 с индексом 8. В вершине стека индекс элемента 5, который больше, чем 3. Поэтому удаляем из стека индекс 9, и записываем в стек индекс 8. В ответ записываем – 1.						
3 итерация. Стек	7			Берем из массива элемент 2 с индексом 7. Просматриваем стек. На вершине стека индекс 8 (элемента 3. 3 больше 2), поэтому удаляем из стека 8, записываем 7. В ответ записываем – 1.					
4 итерация. Стек	6				Для элемента 1 с индексом 6 ответом также является -1				
5 итерация. Стек	6	5				Ответ для текущего элемента 7 с индексом 5 – индекс 6.			
6 итерация. Стек	6	5	4				Ответ для элемента 8 - индекс 5		
7 итерация. Стек	6	5	4	3				Ответ для элемента 9 с индексом 3, индекс 4.	
8 итерация. Стек	6	2							Ответ 6
9 итерация. Стек	6	2	1						2
Исходный массив	6	5	9	8	7	1	2	3	5
ans	2	6	4	5	6	-1	-1	-1	-1

По полученному ответу легко восстановить значения ближайших минимальных элементов ко всем элементам исходного массива. Индексы ближайших минимальных справа к каждому из элементов массива будут храниться в массиве ans[9]. Аналогично решается задача поиска ближайших минимальных слева элементов.

Реализация структуры очередь при помощи статического массива

Очередь **queue** является контейнером позволяющим добавлять элементы в конец (хвост tail) очереди - push(), удалять элементы в начале (голове head) очереди - pop(). Очередь можно представить себе как очередь людей, в которую каждый новый приходящий попадает в конец очереди, а покидают очередь стоящие в ней из ее начала. Дисциплина работы очереди обозначается FIFO, первым пришел — первым уйдешь (First In First Out).



Очередь, как и стек, можно реализовать при помощи статического массива. Если ее реализовывать так, как показано на рисунке выше, то возникают следующие проблемы логики работы очереди: при удалении первого элемента необходимо будет передвигать всю очередь на место первого элемента – на это потребуется время, пропорциональное количеству элементов в очереди. Удобнее будет «закольцевать очередь» *по модулю MAXSIZE*.

```
const int MAX_SIZE=10;
```

Определим класс queue, и опишем во внутреннем разделе класса очередь как статистический массив, имеющий указатели на первый - first элемент и последний – last элемент, которые указывают на голову и хвост очереди.

```
private:
     int a[MAX_SIZE]; // Статический массив для хранения
элементов очереди
     int first;       // Указатель на первый элемент очереди
     int last;        // Указатель на конец очереди
```

В открытом разделе класса инициализируем очередь, поместив указатели first и last на нулевой индекс массива.

```
class queue{
     public:
     queue(){
          last=0;
          first=0;
     }
```

Реализуем метод push() для добавления элементов в конец очереди. Новые элементы добавляются в позицию last по модулю MAXSIZE, после этого указатель last перемещается на следующую позицию. Защита от ошибок предусмотрена в том случае, если количество элементов в очереди больше максимально возможного.

```
     void push(int x){
          if(last-first>=MAX_SIZE){
               cout<< "queue overflow";
               exit(-1);
          }
          else
               a[(last++)%MAX_SIZE]=x;
     }
```

Метод pop() реализован так, что он возвращает первый элемент, а затем указатель first передвигается на следующий элемент.

```
     int pop(){
          return a[(first++)%MAX_SIZE];
     }
```

Размер очереди можно определить при помощи метода size(), которая возвращает значение (last-first)%(MAX_SIZE+1). Очередь пустая, если ее размер равен нулю.

```
     int size(){
          return (last-first)%(MAX_SIZE+1);
     }
```

Для очереди предусмотрены две операции просмотра элементов – элемента в начале очереди top() и элемента в конце очереди back().

```
     int front(){        return a[first];    }
     int back(){         return a[last-1];   }
```

В STL очередь реализована при помощи структуры queue, к которой можно применить стандартные методы, описанные выше и алгоритмы, представленные в начале лекции. Для работы с очередью STL необходимо подключить заголовок #include <queue>. Сложность операции добавления элементов в очередь и их извлечения – O(1).

Задача нахождения минимального элемента на фиксированном отрезке. Реализация очереди для нахождения минимума при помощи двух стеков.

Постановка задачи. Дан массив **A**, состоящий из **n** элементов. Необходимо найти минимум на всех подотрезках массива фиксированной длины **K** за **O(n).** Например, для массива A[10]={1, 8, 4, 3, 5, 7, 4, 5, 6, 7} при K=3 получим ответ ans={1, 3, 3, 3, 4, 4, 4, 5}.

Для начала рассмотрим идеи по реализации очереди при помощи двух стеков. Первый стек head отвечает за уход элементов из очереди, и представляет собой начало очереди. Второй стек tail отвечает за приход элементов в очередь, и представляет собой хвост очереди. Например, в очередь постепенно добавляются элементы: 1, 8, 4, 3, а некоторые из добавленных удаляются. Если при попытке извлечь элемент стек head оказался пустым, то переносим все элементы из tail в head (при этом элементы будут перенесены в обратном порядке, что и нужно для извлечение первого элемента из очереди).

Операции	Состояние стека tail	Состояние стека head
Добавление элемента 1 в очередь	1	Стек пустой
Добавление элемента 8 в очередь	8 (в вершине стека элемент 5) 1	Стек пустой
Добавление элемента 4 в очередь	4 8 1	Стек пустой
Удаление элемента из очереди. Удаляется первый элемент – 1. Так как стек head пустой – переносим все элементы из стека tail в стек head. Затем удаляем 1	Все элементы переносим в стек head	1 – удаляем элемент 1 8 4
Добавляем элемент 3 в очередь	3	8 4
Удаляем элемент из очереди	3	8 – удаляем элемент 8 4
Удаляем элемент из очереди	3	4 – удаляем элемент из очереди
Удаляем элемент из очереди	Элементы переместили во второй стек. Стек пустой	3 – удаляем элемент из очереди

Как мы видим, каждый элемент один раз помещается в стек, один раз перекладывается в другой стек, и один раз удаляется из стека. Операция добавления выполняется за O(1). Операция удаления в худшем случае выполняется за O(n).

Модифицируем реализацию очереди при помощи двух стеков для нахождения минимума на отрезке за O(1). Будем хранить в стеках пары <элемент, текущий минимум стека>.

```
stack< pair<int,int> > tail, head;
```

В алгоритме реализуется идея скользящего окна – по массиву движется окно фиксированного K размера, в котором определяется минимум. В первый стек помещаем K элементов, записываем в ответ текущий минимум. Затем в первый стек добавляем (K+1) элемент. Извлекаем первый элемент очереди – для этого перекладываем все элементы из стека tail в стек head с новым минимумом для второго стека. Удаляем элемент из вершины второго стека. На вершине второго стека будет находиться минимум для второго подотрезка – записываем ответ. Добавим новый элемент в очередь, выведем ответ – минимум из двух вершин стека. Удаляем элемент из очереди. Продолжаем алгоритм далее. Иллюстрацию приведем на массиве A[10]={1, 8, 4, 3, 5, 7, 4, 5, 6, 7}. K=3

Первый стек **tail**	Ответ **ans**	Второй стек **head**
(4,1) Стек заполнили K элементами. (8,1) **Записали** ответ из вершины стека. (1,1)	1	
(3,1) **Добавили** в стек следующий элемент.	1	

(4,1) (8,1) (1,1)		
Перекладываем элементы во второй стек с новым минимумом для второго стека и удалили элемент с вершины стека	1 3	(1,1) – **удалили** элемент (8,3) – **записали** ответ (4,3) (3,3)
(5,5) **добавили** элемент в очередь	1 3	(8,3) – **удаляем** элемент (4,3) (3,3)
(5,5) **Записали** ответ –минимум из двух вершин стеков	1 3 3	(4,3) (3,3)
(7,5) **Добавили** элемент в очередь (5,5)	1 3 3 3	(4,3) – **удаляем** элемент (3,3)
Продолжаем алгоритм далее.		

Таким образом, извлечение минимума происходит с вершин стеков если они оба не пустые, или из вершины непустого стека если имеется один пустой стек.

```
if (tail.empty() || head.empty())
     min = tail.empty ? head.top().second : tail.top().second;
else
     min = min (tail.top().second, head.top().second);
```

При добавлении элементов в виде пары <элемент, минимум>, нужно определять текущий минимум стека, который становится вторым элементом пары.

```
int min = tail.empty()?new_el : min (new_el,tail.top().second);
tail.push (make_pair (new_el, min));
```

Извлечение элемента происходит из стека head если он не пустой, в противном случае элементы их первого стека перекладываются в стек head, а затем элемент извлекается из вершины стека.

```
if (head.empty())
     while (!tail.empty()) {
       int elem = tail.top().first;
       tail.pop();
       int min = head.empty()?elem:min(elem,head.top().second);
       head.push (make_pair (elem, min));
     }
res = head.top().first;
head.pop();
```

Минимум на одном подотрезке модификацией очереди в виде двух стеков описанным способом определяется за O(1). Таким образом, сложность работы всего алгоритма – O(n).

Дек STL. Задача нахождения минимального элемента на фиксированном отрезке при помощи деков.

Дек **deque** (двусторонняя очередь) является контейнером позволяющим добавлять элементы в конец push_back() и начало очереди push_front() и удалять элементы в начале pop_back() и конце pop_front() очереди. deque поддерживает доступ к произвольному элементу.

Решим задачу нахождения минимального на фиксированном отрезке при помощи деков. Рассмотрим сначала ситуацию, когда нам нужно найти минимальный на одном подотрезке.

В очереди нужно хранить не все элементы, а только нужные для определения минимума. В этом случае очередь представляет собой неубывающую последовательность чисел.

```
deque<int> q;// Дек для хранения неубывающей последовательности
```

В вершине такой очереди хранится минимум последовательности.

```
min = q.front();
```

Добавление элементов в очередь происходит следующим образом: пока в конце очереди элементы, большие или равные данному, то удаляем их, и добавляем новый элемент в конец очереди.

```
while (!q.empty() && q.back() > new_el)
     q.pop_back();
q.push_back (new_el);
```

Тем самым мы, с одной стороны, не нарушим порядка, а с другой стороны, не потеряем текущий элемент, если он на каком-либо последующем шаге окажется минимумом. Но при извлечении элемента из головы очереди его там, может и не оказаться. Это происходит потому, что модифицированная очередь могла выкинуть этот элемент в процессе перестроения. Поэтому при удалении элемента необходимо значение извлекаемого элемента - если элемент с этим значением находится в голове очереди, то он извлекается, а в противном случае никаких действий не производится.

```
if (!q.empty() && q.front() == rem_el)
     q.pop_front();
```

Минимум на одном подотрезке описанным алгоритмом определяется за O(1).

Для применения этого алгоритма в случае определения минимума на всех подотрезках фиксированной длины заданного массива необходимо, как и в случае очереди на двух стеках делать следующие действия. Сначала записать указанным способом в дек K элементов, **записать** ответ – текущий минимум на подотрезке, **добавить** новый элемент в дек, **удалить** (если это возможно) первый элемент из дека, **записать** новый ответ из начала дека, **добавить** элемент в конец дека, **удалить** (если это возможно) элемент из начала очереди и так далее.

Сложность работы всего алгоритма – O(n).

Реализация задачи для нахождения минимума на фиксированном подотрезке при помощи дека проще, чем при помощи модификации очереди двумя стеками. Но для способа с деком придется хранить весь массив, а в случае с двумя стеками весь массив хранить не нужно – нужно лишь знать значение очередного i элемента. заполняются два вспомогательных массива – массив префиксных минимумов на блоках длины K и максим постфиксных минимумов для блоков длины K. Для массива A[10]={1, 8, 4, 3, 5, 7, 4, 5, 6, 7} при K=3 получим prefix[10]={1,1,1,3,3,3,4,4,4,7} и suffix[10]={1,4,4,3,5,7,4,5,6,7}.

Результирующий массив ответов получаем по следующему закону: минимум на отрезке [i, i+K-1] равен min(suffix[i], prefix[i+K-1]). ans={1, 3, 3, 3, 4, 4, 4, 5}.

Структуры данных. Словари. Множества

Стандартная библиотека шаблонов STL. Ассоциативные контейнеры: множества, словари. Пары. Методы работы с множествами и словарями. Представление о бинарном дереве поиска. Примеры задач, решаемых с использованием структур set и multiset.

Общее представление

Ассоциативные контейнеры обеспечивают быстрый доступ к данным по ключу. Такие контейнеры построены на основе сбалансированных деревьев. К ассоциативным контейнерам относятся: словари (map), словари с дубликатами (multimap), множества (set), множества с дубликатами (multiset) и битовые множества (bitset). Для использования указанных контейнеров необходимо подключить заголовочные файлы: map, set, bitset.

Множество set

Множество состоит из элементов. Запись $x \in A$ означает, что x является элементом множества A. Примерами множеств являются множество натуральных чисел $N = \{1, 2, 3, \dots\}$, множество простых чисел $P = \{2, 3, 5, 7, \dots\}$. С математической точки зрения относительно каждого объекта можно лишь утверждать – принадлежит ли он множеству или не принадлежит, другие условия на элементы множества не накладываются. Элементы в множестве могут быть представлены в неупорядоченном виде, а также в множестве могут быть несколько экземпляров

одного элемента. Элементами множества могут являться сложные объекты, например пары. Примером множества, состоящего из пар целых чисел, произведение которых равно 15, является множество {(5,3), (1,15), (-1, -15), …}. Множество может состоять и из геометрических объектов, например, множество окружностей на плоскости, множество прямых – касательных к заданной окружности. Особо выделяют пустое множество ∅ – множество, не содержащее ни одного элемента.

В языке программирования C++ контейнер **set** является множеством, в котором элементы упорядочены, и каждый элемент представлен в единственном экземпляре (нет повторяющихся элементов). Операции добавления, удаления и поиска в множестве выполняются за $O(\log n)$, где n – текущий размер множества. При объявлении множества указывают его имя и тип элементов, из которых множество будет состоять.

```
set<int> x; // Объявляется множество x целых чисел
```

Для работы с множеством используются следующие методы.

Метод	**Описание метода**
insert(value)	Добавляет элемент в множество
erase(value)	Удаляет элемент из множества
erase(it)	Удаляет элемент, соответствующий итератору it
find (it1, it2, value)	Возвращает итератор на элемент value в множестве. Если такого элемента нет, то возвращает end()

lower_bound(it1, it2, value)	Возвращает итератор на первый элемент в упорядоченном диапазоне, который имеет значение большее или равное val. Если такого элемента нет, то метод возвращает итератор it2.
	Пример.
	Вектор `v[8] = ( -1, -1, 1, 2, 2, 3, 3, 4 ).`
	`auto result = lower_bound(v.begin(), v.end(), 3);`
	`//*result=3`
	`auto result = lower_bound(v.begin(), v.end(), 5);`
	`//result=v.end()`

upper_bound(it1, it2, value)	Возвращает итератор на первый элемент в упорядоченном диапазоне, который имеет значение строго больше val. Если такого элемента нет, то метод возвращает итератор it2.
	Пример.
	Вектор `v[8] = ( -1, -1, 1, 2, 2, 3, 3, 4 ).`
	`auto result = upper_bound(v.begin(), v.end(), 3);`
	`//*result=4`
	`auto result = upper_bound(v.begin(), v.end(), 4);`
	`//result=v.end()`

Рассмотрим работу с множеством на примере ряда несложных задач.

Количество различных элементов последовательности

На координатную прямую по одной добавляют N точек. Нужно узнать, сколько среди них различных, после каждого добавления новой точки. Так как множество хранит уникальные элементы, то каждый раз при добавлении элемента, уже имеющегося в множестве, добавления на самом деле не происходит. Таким образом, количество элементов в множестве после добавления очередной точки будет являться ответом к задаче.

Заметим также, что на каждом шаге алгоритма мы получаем отсортированное множество

```
#include<iostream>
#include<set>
using namespace std;
int main()
{
     int n,x;
     set<int> a;
     cin >>n;
     for (int i=0;i<n;++i)
     {
          cin >>x;
          a.insert(x);
          cout<<a.size()<<endl;
     }
     return 0;
}
```

Поиск элементов в множестве. Вывод элементов множества

Рассмотрим добавление точек на плоскость. Будем добавлять точки в множество точек **p**.

Для оперирования точками удобно использовать структуру пар **pair**, позволяющую обрабатывать два объекта как один объект. При помощи множества реализуется добавление точек только в одном экземпляре. Заметим, что множество пар будет отсортировано. Пары сортируются сначала по первой координате, затем по второй. После заполнения множества точек поступает запрос на наличие точки с заданными координатами во множестве. Для ответа на вопрос используется метод **find**.

```
int main()
{
     int n,x,y;
     pair<int,int> c_p; //пара состоит из двух координат точки
     set<pair<int,int>> p; //множество точек
     cin>>n;
     for (int i=0;i<n;++i)
     {
          cin >>x>>y;          //вводятся координаты точек
          c_p=make_pair(x,y); //конструируется пара
          p.insert(c_p);     // точка добавляется в множество.
     }
     cin>>x>>y; //координаты точки,
     c_p=make_pair(x,y);
     if (p.find(c_p)!=p.end()) cout<<"YES"; else cout<<"NO";
     return 0;
}
```

Элементы множества можно вывести на экран. Это делается при помощи итератора, указывающего на объект множества. Поскольку итератор только указывает на элемент, то для печати на экран самого значения элемента необходимо разыменовать итератор при помощи операции «звездочка» *, которая записывается перед итератором.

```
     for (auto j=p.begin();j!=p.end();++j)
     {
          c_p=(*j);
          cout<<c_p.first<<" "<<c_p.second<<endl;
     }
     return 0;
```

Заметим, что элементы множества вывелись на экран в отсортированном (по возрастанию порядке). Для того, чтобы вывести элементы множества в отсортированном по убыванию порядке можно использовать методы доступа rbegin(), rend().

```
for (auto j=p.rbegin();j!=p.rend();++j)
{
    c_p=(*j);
    cout<<c_p.first<<" "<<c_p.second<<endl;
}
```

Рейтинг объектов

При помощи множества удобно ранжировать объекты по какому либо признаку. Например, если в множестве хранить объекты пар: (средняя оценка учащегося, номер учащегося), то мы получим рейтинг учащихся. То есть, объекты будут отсортированы по средней оценке. Этот факт удобно использовать в ряде задач.

Будем добавлять в множество пары, состоящие из среднего балла ученика и его номера в журнале. Выведем на экран элементы множества – получим рейтинг учеников по среднему баллу.

```
#include<iostream>
#include <set>
using namespace std;
int main()
{
    int n;
    double x; // Средняя оценка ученика

    pair<double,int> c_p; //Пара - ср.оценка, номер по журналу
    set<pair<double,int>> p; //объявление множества
    cin>>n;
    for (int i=1;i<=n;++i) //Цикл ввода средних баллов
    {
        cin >>x;
        c_p=make_pair(x,i);
        p.insert(c_p);
    }

    for (auto j=p.rbegin();j!=p.rend();++j) //Вывод элементов
    {
        c_p=(*j);
        cout<<c_p.second<<" "<<c_p.first<<endl;
    }
    return 0;
}
```

В set все элементы последовательности записывались в одном экземпляре в отсортированном виде. Если использовать структуру множество для сортировки элементов, то необходимо, чтобы в множество можно было добавить все элементы, а не только уникальные. Такую логику работы с данными обеспечивает множество с дубликатами **multiset**.

Множество с дубликатами multiset

В языке программирования C++ контейнер **multiset** является множеством, в котором элементы упорядочены и могут храниться повторяющиеся элементы. Операции добавления, удаления и поиска, также как и в обычном множестве, в множестве с дубликатами выполняются за $O(\log\ n)$, где n – текущий размер multiset. Объявление множества с дубликатами осуществляется по тем же правилам, что и для обычного множества.

```
multiset<int> x; //Объявляется мультимножество x целых чисел
```

Для работы с множеством используются такие же методы, что и для set.

С помощью множества с дубликатами удобно сортировать объекты – одинаковые объекты останутся в нужном количестве экземпляров в множестве в отсортированном виде. Используя структуру multiset и стандартные методы работы с ней, удобно отвечать на запрос – какое количество одинаковых элементов в множестве, равных заданному значению.

Количество одинаковых элементов, равных заданному значению

```
int n,x,val_search,count=0;
cin>>n;
multiset<int> a;
for (int i=0;i<n;++i){
     cin>>x;
     a.insert(x);   //заполнение множества с дубликатами
}
cin>>val_search;
for(auto i=a.lower_bound(val_search);i!=a.upper_bound(val_search); i++){
               count++;
}
cout<<count;
return 0;
```

Рассмотрим работу программы на примере множества с дубликатами {1,1,2,2,2,3,3}. Вводится *val_search=2*. Программа определяет количество элементов множества, равных 2. На рисунке показано, на какой элемент указывает итератор a.lower_bound(2) – на первый элемент, больший или равный 2, и на какой элемент указывает итератор a.upper_bound(2) – на первый строго больший 2. Таким образом, программы выведет значение count=3.

1		2	2	2	3	3
		↑a.lower_bound(2)			↑a.upper_bound(2)	

Если в множестве нет элементов, равных 2, то указатели будут находится на первом элементе, большим 2.

1	1	3	3	3	3	4
					↑a.lower_bound(2)↑a.upper_bound(2)	

Заметим, что можно было получить значение количества элементов, равных val_search при помощи стандартного метода count().

```
cout<<a.count(val_searsch);
```

Использование предиката greater и собственных предикатов

В стандартной библиотеке определены шаблоны функциональных объектов для операций сравнения и логических операций, определенных в языке C++. Они возвращают значения типа bool – их называют предикатами. По умолчанию элементы в множествах и словарях в STL упорядочены в порядке возрастания. Это обеспечивает по умолчанию предикат less. В явном виде можно использовать другой предикат, обеспечивающий сортировку элементов по убыванию – greater.

```
multiset<int,greater<int>> a;
```

Для использования предиката greater при создании множества (словаря) необходимо указать его после типа элементов множества. Для предиката указывается тип сравниваемых элементов.

Чтобы применить свой критерий сортировки, надо представить бинарный предикат в форме класса или структуры, реализующий оператор operator(). Ниже в примере приведен текст программы, которая заполняет множество ***a*** точками таким образом, чтобы они сортировались по убыванию расстояний от них до начала координат. Бинарный предикат представлен в форме класса Sort_set.

```
struct point{
          int x;
          int y;
};
class Sort_set{
     public:
          bool operator() (const point& p1,const point& p2){
               return (p1.x*p1.y>p2.x*p2.y);
          }
};
int main()
{
     set <point,Sort_set> a;
     int n;
     cin>>n;
     for(int   i=0;i<n;++i){
          int  x_c,y_c;
          point p_c;
```

```
          cin>>x_c>>y_c;
          p_c.x=x_c; p_c.y=y_c;
          a.insert(p_c);
     }
     return 0;
}
```

Словари map

Словарь построен на основе пар значений. Первое значение пары – ключ для идентификации элементов, второе - собственно элемент. Например, в телефонном справочнике номеру телефона соответствует фамилия абонента. *В словарях элементы хранятся в отсортированном по ключу виде.* Поэтому для ключей должно быть определено отношение «меньше». В словаре map, в отличие от словаря с дубликатом (multimap) все ключи являются уникальными.

Рассмотрим задачу определения слова – синонима. Вначале работы программы заполняется словарь, в котором ключ будет являться первым словом, а синоним – вторым словом. Далее используется функция поиска элементов в словаре по ключу. Если поиск дал результат, то итератор будет указывать на найденный элемент словаря, если же поиск не дал результатов, то итератор будет указывать на end() словаря.

```
#include<iostream>
#include<map>
#include<string>
using namespace std;
int main()
{
    string word_first, word_second,word_find;
    map<string, string> voc;
    int n;
    cin >> n;
    for (int i = 0; i < n; ++i){
        cin >> word_first >> word_second;
        voc[word_first] = word_second; }// Создание элемента map
    cin >> word_find;
    if (voc.find(word_find) != voc.end())//Поиск по ключу
        cout << voc[word_find];
    else
        for (auto c : voc){ //Проход по элементам map
            if (c.second == word_find) cout << c.first;}
    return 0;
}
```

Методы работы с map

lower_bound(key)	Указывает на первый элемент, ключ которого равен или больше указанному ключу key или на end(), если такой ключ не найден
upper_bound(key)	Указывает на элемент, ключ которого больше указанного ключа key или end(), если такого нет.
find(key)	Возвращает итератор на найденный элемент
count(key)	Возвращает количество элементов, ключ которых равен key

Словарь map хранит элементы в упорядоченном по возрастанию ключа порядке. Для реализации другого порядка упорядочивания элементов необходимо использовать собственные предикаты или предикат greater.

Словари с дубликатами multimap допускают хранение элементов с одинаковыми ключами. Элементы с одинаковыми ключами хранятся в словаре в порядке их занесения в словарь. При удалении элементов удаляются все экземпляры элементов, соответствующие ключу.

Бинарное дерево поиска

Реализация set/map в STL основана на красно-черных деревьях, представляющих собой один из видов сбалансированных деревьев. Представление о деревьях и их использовании для реализации ассоциативных контейнеров дает бинарное дерево поиска (двоичное дерево поиска). Рассмотрим идею построения бинарного дерева поиска и работу с ним. Предварительно приведем основные понятия, необходимые для изучения бинарного дерева поиска.

Ориентированным деревом называется ориентированный граф без циклов, в котором входящие степени всех вершин, кроме одной, равны 1.

Единственная вершина, входящая степень которой равна 0 (в нее не входит ни одного ребра), называется *корнем дерева*. Вершины двоичного дерева, исходящая степень которых равна 0 (из них не выходит ни одного ребра) называются *листьями дерева*.

Ориентированное двоичное дерево – это ориентированное дерево, в котором исходящие степени вершин равны 0, 1 или 2.

Структура данных «двоичное дерево» - это множество записей вида (key, l, r), где key –

ключ, l, r – указатели на две другие записи. Записи называются узлами дерева (tree nodes). Узлы , на которые указывают l и r называют левым и правым ребенком узла n=(key, l, r), причем ключ в любом узле больше или равен ключам во всех узлах левого поддерева этого узла и меньше или равен ключам во всех узлах правого поддерева этого узла. То есть, ключ родителя находится между ключами своего левого и правого ребенка: $n \to l \to key < n \to key < n \to r \to key$.

Указатели l и r могут быть пустыми, то есть равными специальной константе NULL. Совокупность узлов структуры данных «двоичное дерево» образует граф, в котором вершинами являются узлы, а ребрам соответствуют указатели l и r. Этот граф является ориентированным двоичным деревом.

Для обозначения бинарного дерева поиска используют аббревиатуру BST – binary search tree. Ниже на рисунке изображено двоичное дерево поиска, каждая вершина которого хранит ключ в виде целого числа. Также приведен массив чисел в порядке их добавления в бинарное дерево поиска.

Операции, поддерживаемые для структуры данных «двоичное дерево поиска»: добавление элементов, поиск элементов, удаление элементов. Эти операции довольно просты, и обычно реализуются рекурсивно.

1. **Операция добавить**. **insert** (value). Если дерево пусто, в h (корень дерева) заносится ссылка на новый узел, содержащий данный элемент. Если ключ добавляемого элемента меньше ключа в корне, то вызывается функция вставки элемента в левое поддерево, иначе вызывается функция вставки элемента в правое поддерево. Функция принимает дерево в качестве первого параметра и ключ вставляемого элемента в качестве второго. В том случае, если левое (правое) поддерево не содержит детей, то создаем ребенка: помещаем в него данные, и устанавливаем на него указатель.
2. Операция поиска. find (value). Если дерево пусто, то, очевидно, поиск неудачен. Если ключ поиска равен ключу в корне, то поиск успешен. Иначе выполняется рекурсивно поиск в соответствующем поддереве. Функция принимает дерево в качестве первого параметра и ключ в качестве второго. Поиск завершает свою работу, либо когда текущее поддерево станет пустым (неудачный поиск), либо когда будет найден элемент с нужным ключом (успешный поиск).

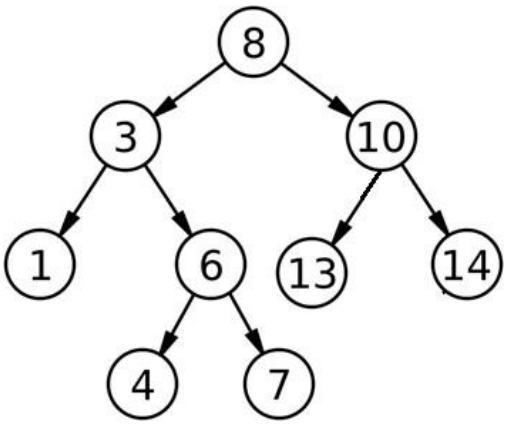


Рис.1. Бинарное дерево поиска

8	3	10	6	14	1	4	7	13

Приведем текст программы, в которой описывается структура node для построения двоичного дерева поиска. Реализуется функция добавления элементов в дерево insert(value), функция поиска элементов find(value).

```
#include<iostream>
using namespace std;

struct node{
     int key; //ключ узла
     node *l, *r; //ссылки на левое и правое поддеревья
     };
node *tree=NULL; //инициализация BST дерева tree
void insert(node*& t, int x){
          if (t==NULL){ //если дерево пустое
               t=new node; //выделили память
               t->key=x; // добавили элемент x
               t->l=NULL; t->r=NULL;
               return;
          }
          if (x<t->key) // вставляемый элемент меньше ключа
               insert(t->l,x);//переход в левое поддерево
          else
               insert(t->r,x); //в противном случае в правое
     }
int find(node *t, int x){
          if (t==NULL) return 0; // дерево пустое
          if (x==t->key) return 1; //ключ совпал с искомым эл.
          if (x<t->key) //искомый элемент меньше ключа
               return find(t->l,x); //переход в левое поддерево
          else
               return find(t->r,x); //иначе в правое
     }
```

```
int main()
{
     int n;
     cin>>n; //количество элементов в дереве
     for (int i=0;i<n;++i){// цикл добавления элементов
          int a;
          cin>>a;
          insert(tree, a); ..
     }
     cout<<find(tree,100);// поиск элемента
return 0;
}
```

Чтобы разобраться в логике работы программы рекомендуем заполнить множество элементами массива, приведенного на рисунке 1, и изучить структуру полученного дерева при помощи окна «Контрольные значения» Visual Studio.

Оценка сложности операций, производимых на бинарном дереве поиска

Глубиной узла дерева n называется величина d(n), равная количеству ребер в пути от корня до этого узла. *Высотой дерева* H называется глубина самого глубокого листа. Рекурсивные процедуры добавления и поиска для двоичного дерева работают за время, ограниченное сверху высотой дерева. В худшем случае число шагов равно глубине самого глубокого узла, то есть высоте дерева, а в среднем – средней глубине узлов. Полное двоичное дерево – это дерево, у которого путь от корня до любого листа содержит H ребер. Если в таком дереве n узлов, то высота дерева оценивается как log n. Приведем оценку сложности операций на полном двоичном дереве поиска. Заметим, что в общем случае сложность операций зависит от высоты дерева.

Операция	Сложность. BST	Сложность. Отсортированный массив
Вставка элемента	O(log n)	O(n)
Поиск элемента по ключу	O(log n)	O(log n)
Другие операции с BST: получение отсортированного массива из BST за O(n), поиск порядковой статистики (k-я порядковая статистика – k-е по величине число в массиве) за O(log n), определение порядкового номера элемента за O(log n) (в отсортированном массиве, соответствующему элементам дерева).		

Реализация функции удаления элементов из бинарного дерева поиска зависит от типа удаляемого элемента в дереве:

- при удалении листьев очищается указатель из предка на данный лист;
- при удалении узла с одним потомком переписывается ссылка из предка удаляемого узла на потомка удаляемого узла (раньше она указывала на удаляемый узел);
- при удалении узла, у которого два потомка действуем по алгоритму: выбираем самый правый узел в левом поддереве удаляемого узла (максимальный элемент из элементов, меньших удаляемого узла), записываем его значение в удаляемый узел, удаляем найденный элемент.

Проиллюстрируем функцию удаления элемента 8 в бинарном дереве поиска на рисунке 2. Сначала находим самый правый элемент в левом поддереве элемента с ключом 8, это элемент с ключом 6. Заменяем ключ 8 в «удаляемом» элемента на ключ 6. Удаляем элемент с ключом 6 из дерева. Найденный элемент будет либо листом, либо будем иметь одного левого ребенка.

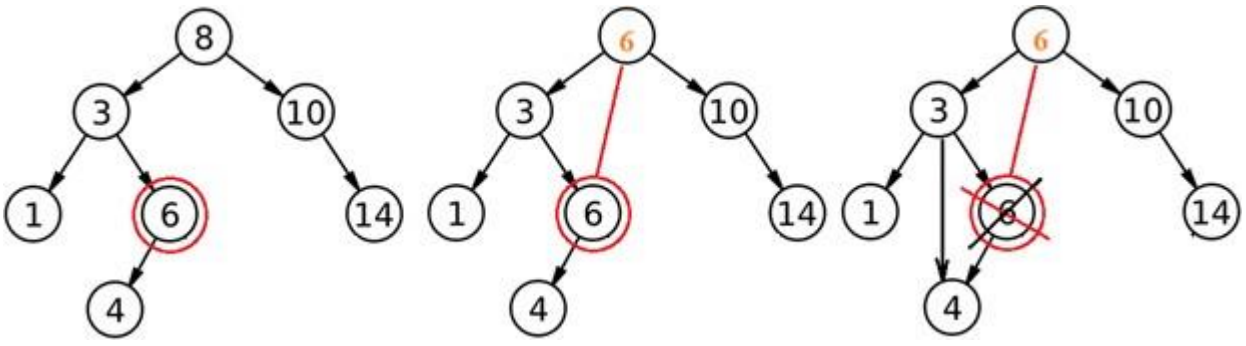


Рис.2. Удаление элемента key=8 в бинарном дереве поиска

Удаляя элементы, которые имеют двух детей, можно искать не самый правый элемент в левом поддереве, а самый левый элемент в правом поддереве.

Лекция 7

Бинарный поиск

Последовательный и бинарный поиск. Сложность последовательного и бинарного поиска. Поиск элементов в массиве, поиск по целым числам. Нижняя и верхняя граница искомого числа. Поиск по вещественным числам. Определение корней функции. Примеры задач.

Общее представление

Игра «Угадай число». Вам загадали число от 1 до 1000, $x \epsilon [1;\ 1000]$. Вы должны его отгадать, задавая вопросы, на которые можно получить ответ «да» или «нет». В худшем случае Вы последовательно называете числа: 1, 2, 3,…999. В общем случае $x \epsilon [1;\ n]$ и сложность такого алгоритма – $O(n)$. То есть в худшем случае вы зададите $n\ -\ 1$ вопросов, чтобы отгадать число. Идея оптимизации алгоритма заключается в делении области множества чисел, в которой производится поиск, пополам.

На рисунке изображено множество чисел $[1;\ 16]$. Загадано число 13. Каждый раз Вы называете число *x*, и задаете вопрос «Больше или равно *x* загаданное число?». Первый раз Вы называете число 8 – середину области поиска, и, узнав, что загаданное число больше 8, переходите в область поиска $[9;\ 16]$. Таким образом, область поиска уменьшилась в 2 раза. Называете число 12, и переходите в область поиска $[13;\ 16]$. Называете число 14, и переходите в область поиска $[13;\ 14]$. Наконец, вы называете число 13 и получаете ответ, что это и есть загаданное число. Вам потребовалось 4 вопроса.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Так как каждый раз область поиска уменьшалась в два раза, то в худшем случае потребуется $\log_2 n$ вопросов для отгадывания числа. Получаем логарифмическую сложность алгоритма поиска числа – $O(\log\ n)$. *Бинарный поиск (двоичный поиск) – алгоритм поиска в упорядоченном множестве чисел, использующий метод деления области поиска пополам и имеющий логарифмическую сложность*.

Алгоритм бинарного поиска

Применим бинарный поиск для поиска искомого числа в упорядоченном массиве. Обозначим правую границу поиска переменной *right*, левую границу – *left*. Середину области поиска – *middle*. *middle=(left+right)/2.*

```
cin >> b; // Искомый элемент
left = -1;
right = n - 1; // Левая и правая граница поиска
while (right - left > 1) // Пока правая граница правее левой
{
     middle = (left + right) / 2; // Середина области поиска
     if (a[middle] >= b)
          right = middle; // Передвигаем правую границу
     else
          left = middle; // Иначе передвигаем левую границу
}
if (a[right] == b)
     cout << right;
```

```
else
     cout << -1;
```

Программа выведет индекс искомого элемента, либо -1, если такого элемента в массиве нет.

После выполнения алгоритма переменная *right* будет указывать на первый элемент в массиве, который равен искомому элементу или больше искомого. Переменная *left* будет указывать на самый большой элемент, меньше искомого. Если элемента, равного искомому или больше искомого нет, то переменная *right* будет указывать на последний элемент массива.

Встроенные функции C++

В C++ существует встроенная функция **binary_search**, которая проверяет, есть ли в отсортированном диапазоне элемент, равный указанному значению. Функция возвращает значение true если указанный элемент имеется в диапазоне и false – если такой элемент отсутствует. Функция требует заголовка <algorithm>.

```
#include <iostream>
#include <algorithm>
using namespace std;
int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int main()
{
	int b;
	cin >> b;
	cout << binary_search(a, a + 10, b);
	// Поиск числа b в массиве a
}
```

Программа выведет 1, если элемент b найден в массиве и 0 – если не найден.

Приведем и другие полезные функции C++. Функция **lower_bound** возвращает итератор первого элемента в отсортированном диапазоне, который равен или больше искомого элемента.

```
#include <iostream>
#include <algorithm>
using namespace std;
int a[10] = { 1, 2, 3, 4, 5, 5, 5, 8, 9, 10 };
int main()
{
	int b = 5;
	cout << *(lower_bound(a, a + 10, b)) << endl;
	b = 6;
	cout << *(lower_bound(a, a + 10, b));
}
```

Программа выведет значения 5, так как в массиве присутствует элемент 5 и значение 8 – значение первого элемента, большего 6.

Аналогично функции **lower_bound** работает функция **upper_bound**, которая возвращает итератор на элемент, который строго больше искомого в заданном диапазоне упорядоченных элементов.

Сложности работы алгоритма бинарного поиска

Сложность работы алгоритма бинарного поиска – $O(\log\ n)$. Предположим, нам нужно найти число в диапазоне $[1; 1000]$. Каждый шаг алгоритма уменьшает область поиска в два раза:

1000 чисел → 500 чисел → 250 чисел → 125 чисел → 63 числа →32 числа →16 чисел→8 чисел→4 числа→2 числа→1 число. Нам потребовалось $[\log\ 1000] = 10$ итераций. То есть достаточно 10 итераций. Покажем, что меньше итераций быть не может.

Приведем доказательство при помощи протоколов работы алгоритма. Укажем все возможные числа, которые мы можем искать в диапазоне $[1;\ 1000]$ в первом столбце. В первой строке указываем номера вопросов, которые мы задаем при поиске числа. Предположим, нам

потребовалось 9 вопросов. В каждой ячейке ставим знак «+» или «-», в зависимости от того получили вы ответ «да» или «нет» на ваш вопрос. Получаем некоторую таблицу – протокол игры. Каждая строка показывает, как было угадано число, указанное слева в строке.

Искомое число	1	2	3	4	5	6	7	8	9
1	+	-	-	-	-	-	-	-	-
2	-	+	+	+	+	+	+	+	+
3	-	-	-	-	-	+	+	+	+
….	-	+	+	+	+	+	+	+	+
….									
1000	+	-	-	+	-	-	-	-	-

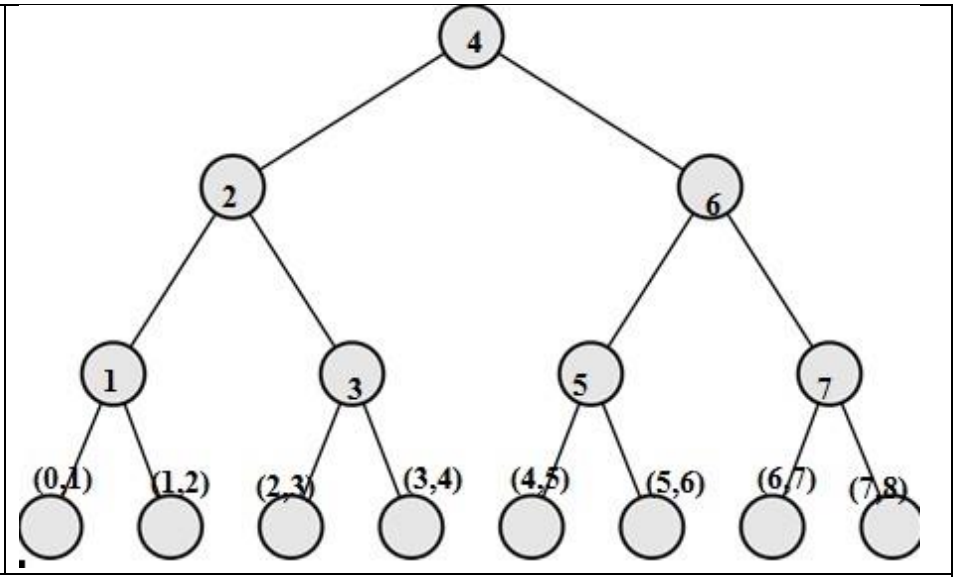
Всего получается 2^9=512 вариантов различных строк (отличающихся друг от друга). Строк в таблице 1000 и получается, что по принципу Дирихле хотя бы для одна комбинация будет соответствовать не менее, чем $\left[\frac{1000}{512}\right] = 2$ строкам протокола. Получается, что существует хотя бы 2 строки, в которых одинаковые комбинации знаков «+» и «-». Поскольку разные числа не могли быть угаданы алгоритмом на основе одних и тех же ответов на одни и те же вопросы, то мы получаем противоречие. То есть для алгоритма бинарного поиска потребуется $\lceil \log n \rceil$ операций.

Работу алгоритма бинарного поиска можно изобразить в виде дерева. Корень дерева – это первое деление диапазона поиска на две части. Предположим, что массив поиска a[8] = { 0, 1, 2, 3, 4, 5, 6, 7 }. Присваиваем значения границ: l = 0, r = 8. Корень дерева соответствует m = (l + r) / 2 = 4. Из корня дерева выходят два ребра – одно (левое) соответствует ветке «да», другое (правое) – ветке «нет» условия if (a[m] > b), где b – искомое число. Две дочерние вершины корня – это следующие переходы по границе *middle*.

Корень дерева – первая проверка
l = 0; r = 8; m = 4

l = 0; r = 4; m = 2
l = 4; r = 8; m = 6

l = 0; r = 2; m = 1
l = 2; r = 4; m = 3
l = 4; r = 6; m = 5
l = 6; r = 8; m = 7



Листья дерева – указатели на r и l после работы алгоритма. l будет указывать на искомое число, если таковое есть.

Глубина дерева – количество уровней в нем. Глубина дерева определяет сложность алгоритма – показывает сколько итераций совершил алгоритм бинарного поиска. Количество листьев дерева определяется как 2^n, где n – номер уровня дерева (уровень вершины-корня нулевой). Вернемся к вопросу сложности алгоритма бинарного поиска. Покажем при помощи дерева работы алгоритма, что меньше, чем 10 итераций для поиска искомого числа среди 1024 чисел, алгоритм сделать не можем. Действительно, если итераций будет выполнено 9, то листьев в дереве получим 512, то есть все наши числа найдены быть не могут. Для работы алгоритма бинарного поиска потребуется $\lceil \log n \rceil$ операций.

Поиск корня уравнения алгоритмом бинарного поиска

Алгоритм бинарного поиска удобно применять для определения корней уравнения (так называемый вещественный бинарный поиск).

Задача. Найдите такое число x, что $x^2 + \sqrt{x} = C$, с точностью не менее 6 знаков после точки.

Входные данные

В единственной строке содержится вещественное число $1.0 \leq C \leq 10^{10}$.

Выходные данные

Выведите одно число — искомый *x*.

Примеры

Входные данные	Выходные данные
2.0000000000	1.000000000
Входные данные	Выходные данные
18.0000000000	4.000000000

На рисунке представлен график функции $y = x^2 + \sqrt{x} - 2$, соответствующей первому входному тесту. Заметим, что функция является возрастающей функцией на всей области определения, как сумма двух возрастающих функций. $y(0) \leq 0$, $y(C) \geq 0$. Значит, функция имеет на промежутке $[0; C]$ единственный корень. Поиск корня будем производить с точностью $\varepsilon = 10^{-10}$. Будем уменьшать диапазон поиска, сдвигая правую или левую границу поиска, пока не достигнем заданной точности. Программа приведена ниже.

```
#include <iostream>
#include <cmath>
using namespace std;
const long double eps = 1e-10;
long double f(long double x){
    return x * x + sqrt(x);
}
int main(){
long double c, left = 0, right = 1e15, middle;
     cin >> c;
     while (fabs(right - left) > eps) {
     // for (int i = 0; i < 100; ++i) на практике используют for
          middle = (left + right) / 2.0;
          if (f(middle) - c < 0)
               left = middle;
          else
               right = middle;
    }
    cout << fixed;
    cout.precision(7);
    cout << right;
    return 0;
}
```

В приведенной программе окончание поиска происходит в том случае, когда рассматриваемый отрезок станет меньше заданной погрешности. Примерное количество итераций алгоритма будет равно $log\left(\frac{R-L}{s}\right)$. В рассматриваемом примере смещение границ поиска происходило на основе сравнения значения функции с нулем: f(middle) – c < 0. Если нет информации относительно монотонности функции, но точно известно, что на интервале поиска она имеет единственный корень, то разумнее будет проверять наличие разных знаков функции на границах новой области поиска: в случае $f(l) * f(m) \leq 0$ передвигать правую границу (переходим в левый промежуток), а в случае $f(r) * f(m) \leq 0$ передвигать левую границу (переходим в правый промежуток).

Решение задач. Бинарный поиск по ответу

Задачи, в которых требуется найти какое-либо значение, часто могут быть решены при помощи алгоритма бинарного поиска. В этом случае говорят о бинарном поиске по ответу. Для решения таких задач необходимо определить исходную область поиска: левую и правую границу поиска. Область поиска как раз и представляет собой упорядоченное множество потенциальных ответов, среди которых нужно выбрать тот, который удовлетворяет условию задачи. Важно предварительно сформулировать условие перехода в левую или правую половину области поиска на каждом шаге алгоритма. Для этого, как правило, происходит вычисление значения данной в задаче характеристики и переход в левую или правую область поиска в зависимости от того,

больше или меньше эта характеристика, чем та, которая требуется для ответа. Рассмотрим пример.

Задача «Дипломы» (Региональный этап Всероссийской олимпиады школьников по информатике 2009 - 2010 учебного года)

К окончанию школы у Пети накопилось *n* дипломов, причём все они имели одинаковые размеры: *w* — в ширину и *h* — в высоту. Петя решил украсить свою комнату, повесив на одну из стен свои дипломы. Он решил купить специальную доску, чтобы прикрепить её к стене, а к ней — дипломы. Петя хочет, чтобы доска была квадратной. Каждый диплом должен быть размещён строго в прямоугольнике размером *w* на *h*. Дипломы запрещается поворачивать на 90 градусов. Прямоугольники, соответствующие различным дипломам, не должны иметь общих внутренних точек. Требуется написать программу, которая вычислит минимальный размер стороны доски, которая потребуется Пете для размещения всех своих дипломов.

```
l = 0; r = (w + h) * n + 1;
while (r - l > 1)
{
     int x = (r + l) / 2;
     if ((x / w) * (x / h) >= n)
          r = x;
     else
          l = x;
}
cout << r;
```

Для решения задачи предварительно была выбрана область поиска: минимальный размер квадрата – 0, максимальный размер квадрата (с избытком) таков, что квадрат превращается в линию длины *(w + h) * n*. На каждой итерации алгоритма мы вычисляем *x* – середину области поиска. Это возможная длина искомого квадрата. Такой квадрат проверяется на то, вместится ли в него больше, чем нужно дипломов или нужное количество дипломов. Если это так, то передвигаем правую границу поиска, то есть будем уменьшать искомую длину квадрата (ведь вместилось больше, чем нужно дипломов). И повторяем поиск до тех пор, пока левая и правая граница поиска не окажутся рядом.

Сортировка подсчетом и применение встроенных сортировок

Стандартная библиотека шаблонов STL: встроенные сортировки и их применение. Использование собственных компараторов в алгоритмах сортировки. Стабильные сортировки. Быстрая сортировка. k-я порядковая статистика. Сортировка подсчетом. Пузырьковая сортировка, сортировка выбором, сортировка вставками. Сложность алгоритмов сортировки. Примеры задач, решаемых с использованием алгоритмов сортировки.

Общее представление

Алгоритмическая задача сортировки формулируется следующим образом: дан массив из **n** чисел. Необходимо упорядочить элементы массива так, чтобы они располагались в массиве по возрастанию (убыванию). Существует достаточно много алгоритмов сортировки, которые имеют различную сложность работы – от квадратичной за O(N^2), до сложности $O(Nlog\ N)$. Разумеется, для успешного выступления на олимпиадах надо понимать логику работы большинства алгоритмов сортировки и уметь пользоваться стандартными алгоритмами библиотеки STL.

Алгоритмы сортировки библиотеки STL

Сортировки контейнеров могут быть выполнены при помощи стандартных алгоритмов сортировки библиотеки STL. Алгоритмы сортировки требуют подключения заголовка <algorithm>.

Приведем перечень алгоритмов сортировки и алгоритмов partition, понимание логики работы которых будет необходимо при изучении быстрой сортировки.

Алгоритм	**Описание алгоритма**
sort(it1, it2, Pred)	Упорядочивает элементы в указанном диапазоне итераторов в порядке возрастания или согласно указанному критерию упорядочивания, заданному бинарным предикатом Pred (компаратором). Сложность алгоритма в среднем – $O(NlogN)$.
stable_sort(it1, it2, Pred)	Упорядочивает элементы при помощи стабильной сортировки, при которой сохраняется относительный порядок сортируемых элементов. Сложность алгоритма больше, чем сложность алгоритма sort(), в лучшем случае она равна $O(NlogN)$, в худшем – $O(N(logN)^2)$. Работает медленнее, чем sort().
partition (it1, it2, Pred)	Разделяет элементы диапазона на два непересекающихся подмножества, при этом элементы, удовлетворяющие унарному предикату, предшествуют тем, которые не удовлетворяют ему. Возвращает итератор на начало второго подмножества. Сложность алгоритма разбиения - $O(N)$.
stable_partition (it1, it2, Pred)	Разделяет элементы диапазона на два непересекающихся множества, при этом элементы, удовлетворяющие унарному предикату, предшествуют тем, которые не удовлетворяют ему, сохраняя относительный порядок элементов массива. Сложность алгоритма разбиения $O(NlogN)$.

Наглядное представление о результате работы каждого из указанных алгоритмов дают нижеприведенные рисунки, на которых a[16]=(5, 1, 9, 2, 0, 5, 7, 3, 4, 5, 8, 5, 5, 5, 10, 6) – исходный массив элементов.

Исходный массив a[16]														
5	1	9	2	0	5	7	3	4	5	8	5	5	5	10	6
Массив после применения алгоритма sort(a, a+16)															
0	1	2	3	4	5	5	5	5	5	5	6	7	8	9	10

Массив после применения алгоритма partition (a,a+16,pr6), где pr6 – унарный предикат. `bool pr6(int val1){ return val1>6;}`															
10	8	9	7	0	6	2	3	4	5	1	5	5	5	5	5
Массив после применения алгоритма stable_partition (a,a+16,pr6)															
9	7	8	10	5	1	2	0	5	3	4	5	5	5	5	6

Стабильная сортировка

Часто применяются методы сортировки элементов с несколькими ключами. Например, если задан закодированный список учеников класса с указанием количества решенных ими задач и требуется упорядочить этот список по убыванию количества решенных задач, то нестабильные алгоритмы сортировки могут существенно перемешать исходные данные. Рассмотрим результаты сортировки записей алгоритмом sort() и алгоритмом стабильной сортировки stable_sort(), приведенные ниже. Записи сортируются по ключу - количеству решенных задач. Как видно из примера, стабильная сортировка сохраняет относительный порядок элементов в исходном массиве. Заметим, что увидеть отличие между результатами работы стандартных алгоритмов сортировки библиотеки STL: сортировки sort() и стабильной сортировки stable_sort() можно только на больших массивах исходных данных.

Исходный массива	После сортировки sort()	После сортировки stable_sort()
Val0 6, Val1 6, Val2 4	Val16 4, Val30 4, Val2 4	Val2 4, Val5 4, Val6 4
Val3 6, Val4 6, Val5 4	Val29 4, Val27 4, Val5 4	Val8 4, Val16 4, Val19 4
Val6 4, Val7 6, Val8 4	Val6 4, Val24 4, Val8 4	Val21 4, Val22 4, Val24 4
Val9 5, Val10 6, Val11 6	Val19 4, Val22 4, Val21 4	Val27 4, Val29 4, Val30 4
Val12 5, Val13 6, Val14 5	Val31 4, Val12 5, Val14 5	Val31 4, Val9 5, Val12 5
Val15 5, Val16 4, Val17 6	Val15 5, Val32 5, Val18 5	Val14 5, Val15 5, Val18 5
Val18 5, Val19 4, Val20 6	Val23 5, Val25 5, Val28 5	Val23 5, Val25 5, Val28 5
Val21 4, Val22 4, Val23 5	Val9 5, Val10 6, Val17 6	Val32 5, Val0 6, Val1 6
Val24 4, Val25 5, Val26 6	Val7 6, Val13 6, Val26 6	Val3 6, Val4 6, Val7 6
Val27 4, Val28 5, Val29 4	Val4 6, Val20 6, Val3 6	Val10 6, Val11 6, Val13 6
Val30 4, Val31 4, Val32 5	Val1 6, Val11 6, Val0 6	Val17 6, Val20 6, Val26 6

Использование собственных предикатов для алгоритмов сортировки

Функция sort() может принимать в качестве третьего параметра функцию – критерий сортировки (компаратор). Приведем программу, в которой в векторе хранятся элементы структуры list_fam. Структура позволяет описать элементы, состоящие из двух объектов – фамилии и оценки ученика.

```
struct list_fam{
          string fam;
          int value;
     };
```

Сравнение двух элементов, имеющих тип описанной структуры, должно производиться по какому-либо критерию сортировки. Например, при сравнении элементов только по баллу ученика и упорядочивании элементов по убыванию, можно использовать следующий компаратор.

```
bool comp (list_fam a, list_fam b){
          return a.value>b.value;
     }
```

Алгоритм сортировки в этом случае будет вызываться с указанием компаратора.

```
stable_sort(rating.begin(),rating.end(),comp);
```

В случае необходимости сортировки по убыванию, можно либо написать собственный компаратор, либо выполнить сортировку, а потом воспользоваться методом reverse(), который расположит элементы контейнера в обратном порядке, либо передать сортировке итераторы rbegin(), rend().

Алгоритм partition

Алгоритм partition() позволяет разбить множество на два подмножества по критерию, описанному в унарном предикате. В примере ниже предикат возвращает значение true, если элемент не кратен 3. После применения алгоритма partition(), вначале массива будут расположены не кратные 3 числа, затем кратные 3 числа. Таким образом, массив **a[]** будет разбит на два подмножества – одно из них состоит из чисел {1, 2, 8, 4, 5, 7}, другое из чисел {6, 3, 9}. Алгоритм partition() возвращает итератор на первый элемент второго подмножества.

```
int a[]={1, 2, 3, 4, 5, 6, 7, 8, 9};
bool comp (int l) { return (l%3); }
int main()
{
     int *i;
     i = partition (a, a+9, comp);
     for (int *j=a; j!=a+9;++j)
          cout<<(*j)<<" ";
     cout<<endl;
     for (int *j=a; j!=i;++j)
          cout<<(*j)<<" ";
```

Сложность работы алгоритма O(N), где N – количество элементов в массиве.

Быстрая сортировка QuickSort

Рассмотренные до этого алгоритмы сортировки Алгоритм быстрой сортировки QuickSort имеет в среднем сложность $O(Nlog\ N)$ и работает по принципу «разделяй и властвуй». Алгоритм QuickSort является составной частью стандартного алгоритма sort() библиотеки STL.

В общем виде работу алгоритма можно описать следующим образом: выбирается опорный элемент - partitioning element. Сортируемый массив переупорядочивается относительно опорного элемента, и разбивается на две части: левая часть состоит из элементов, не больших опорного, а правая часть состоит из элементов, не меньших опорного. Полная упорядоченность массива достигается разбиением его на части с последующим рекурсивным применением к ним этого же метода. Заметим, что быстрая сортировка не является стабильной сортировкой.

Приведем пример работы алгоритма быстрой сортировки на примере массива

a[7]={7, 8, 1, 2, 4, 3, 6}.

Вначале работы алгоритма l=0; r=N-1; *Выбираем опорный элемент*. Существуют разные способы выбора опорного элемента p, например, p=a[m]=a[(l+r)/2] или p=a[l]. Теоретически можно выбирать случайный элемент между левой и правой границей, но функция генерации случайного числа работает медленнее, чем простые арифметические действия, поэтому выбор случайного числа может снизить производительность алгоритма. Компромиссным вариантом между выбором конкретного элемента последовательности и случайным его выбором, является способ выбора m=(x*l+y*r)/(x+y), где x и y - два произвольных числа.

После выбора опорного элемента *массив переупорядочивается* (происходит разбиение массива) таким образом, чтобы элементы не больше опорного находились бы вначале массива, а элементы, не меньшие опорного – в правой части массива. Это напоминает работу алгоритма *partition()*. На рисунке мы видим трассировку алгоритма – пока указатель i находится на элементе, меньшим, чем опорный, он передвигается вправо. Аналогично, пока указатель j находится на элементе, большим, чем опорный он передвигается влево. Указатели остановятся на «неправильно» расположенных элементах, в этом случае их надо поменять местами, и после этого сдвинуть указатели i и j на одну позицию вправо и влево соответственно. Процедура продолжается до тех пор, пока выполняется условие i<=j. В конце работы процедуры переупорядочивания элементов указатели i и j поменяются местами.

i	j	p	0	1	2	3	4	5	6
0	6	p=a[3]=2	7↑i	8	1	2	4	3	6↑j
0	5	2	7↑i	8	1	2	4	3↑j	6
0	4	2	7↑i	8	1	2	4↑j	3	6
0	3	2	7↑i ⟸	8	1 ⟹	2 ↑j	4	3	6
1	2	2	(2)	8↑i ⟸⟹	1↑j	7	4	3	6
2	1	2	2	1↑j	8↑i	7	4	3	6

Левая часть массива a[l, i-1] состоит из элементов не больших, чем опорный (на рисунке выделена серым цветом). Правая часть массива a[j+1, r] состоит из элементов не меньших, чем опорный.

Далее *вызывается рекурсивно процедура QuickSort* для левой и правой частей массива. Рекурсивный вызов осуществляется при условии, что l<j (для левой части) и r>i (для правой части).

Рекурсивный вызов Quicksort(l=0,j=1) от левой части массива									
0	1	2	(2) ↑i ⟸⟹	1↑j	8	7	4	3	6
0	1	2	1↑j	(2) ↑i	8	7	4	3	6
Рекурсивный вызов Quicksort(l=2,j=6) от правой части массива									
2	6	4	1	2	8↑	7	(4)	3	6↑
2	5	4	1	2	8↑i ⟸	7	(4) ⟹	3↑j	6
3	4	4	1	2	3	7↑i ⟸⟹	(4) ↑j	8	6
4	3	4	1	2	3	(4) ↑j	7↑i	8	6
Рекурсивный вызов Quicksort(l=4,j=6)									
4	6	8	1	2	3	4	7↑i	8	6↑j
5	6	8	1	2	3	4	7	8 ↑i	6↑j
6	5	8	1	2	3	4	7	6↑j	8 ↑i
Рекурсивный вызов Quicksort(l=4,j=5)									
4	5	7	1	2	3	4	7 ↑i	6↑j	8
5	4	7	1	2	3	4	6↑j	7 ↑i	8

Приведем программную реализацию функции быстрой сортировки.

```
void quick_sort(int l, int r){
     int i=l, j=r, p=a[(l+r)/2],temp;
     while(i<=j){
          while(a[i]<p) i++;
          while(a[j]>p) j--;
          if(i<=j){
               temp=a[i];
               a[i]=a[j];
               a[j]=temp;
               i++; j--;
          }
     }
     if (l<j) quick_sort(l,j);// Левая часть
     if (i<r) quick_sort(i,r); //Правая часть
}
```

Оценка сложности быстрой сортировки

Алгоритм быстрой сортировки в среднем использует $O(Nlog\ N)$ сравнений и $O(Nlog\ N)$ присваиваний, $O(log\ N)$ дополнительной памяти для хранения стека рекурсивных вызовов.

Быстрая сортировка неэффективна на некоторых простых массивах данных, например, если она применяется для сортировки уже отсортированных массивов. В худшем случае алгоритм имеет сложность $O(N^2)$. Кроме того, рекурсивные вызовы являются затратными операциями, и для небольших массивов данных быстрая сортировка будет работать не очень эффективно. Поэтому, в случае небольших объемов данных (<K, где K – некоторое небольшое число, например, 32), быстрая сортировка также работает не очень хорошо. В случае небольшого количества данных используют нерекурсивные методы сортировки, например, сортировку вставками или выбором, что дает прирост производительности алгоритма сортировки.

Рассмотрим *худший случай работы алгоритма быстрой сортировки* на примере упорядоченного массива данных, в котором N элементов. При первом разбиении массива процедурой partition производится N сравнений. Рекурсивный вызов от левой части массива размером (N-1)/2 и в дальнейшем от правой части массива размером (N-1)/2 приводит в общей сложности к N сравнениям. Рассуждая далее, получим общее количество сравнений для упорядоченного файла, равное N+(N-1)+(N-2)+…+2+1=(N+1)*N/2, то есть порядка N^2.

В среднем быстрая сортировка выполняет $2NlogN$ сравнений. Проведем оценку количества сравнений, выполняемых во время быстрой сортировки N>=2 случайно распределенных различных элементов.

$$C_N = N + 1 + \frac{1}{N} \sum_{1 \le k \le N} (C_{k-1} + C_{N-k})$$

N+1 – количество сравнений опорного элемента на первом шаге алгоритма с *N-1* элементом и еще двумя, когда индексы пересекутся. Каждый *k*-й элемент может быть центральным с вероятностью *1/N*. После выбора *k-го* опорного элемента получаем частично упорядоченные части массива с размерами *k-1* и *N-k*. Заметим, что $C_0 + C_1 + \cdots C_{N-1} = C_{N-1}+C_{N-2}+\ldots+C_0$. Значит,

$$C_N = N + 1 + \frac{2}{N}\sum_{1 \le k \le N} C_{k-1}.$$

Умножим обе части сравнения на N и вычтем эту же формулу для *N-1*.

$$NC_N - (N-1)C_{N-1} = N(N+1) - (N-1)N + 2C_{N-1}.$$

$$NC_N = 2N + (N+1)C_{N-1}.$$

Поделим обе части на $N(N+1)$ и получим соотношение $C_N/(N+1) = 2/(N+1) + C_{N-1}/N$.

Применив дальнейшие преобразования, получим $\frac{CN}{N+1} = \frac{2}{N+1} + \frac{2}{N} + \frac{CN-2}{N-1} = \cdots = \frac{C2}{3} + \sum_{3 \le k \le N} \frac{2}{k+1}$.

Из курса математического анализа в дальнейшем вы узнаете, что приблизительное значение $\frac{C_N}{N+1} \approx 2\sum_{1\le k<N}\frac{1}{k} \approx 2lnN$. То есть, сложность алгоритма быстрой сортировки в среднем равно $O(Nlog\ N)$.

Поиск k-й порядковой статистики

Поиск *k*-й порядковой статистики заключается в поиске *k*-го элемента в упорядоченном по неубыванию массиве *A*, состоящего из *N* элементов. Первая порядковая статистика (*k=1*)– это минимум массива, *N*-я порядковая статистика (*k=N*) – это максимум массива. Медиана – это порядковая статистика, где *k=N/2*. Приведем пример k-х порядковых статистик на примере конкретного массива.

Исходный массив a[16]															
5	1	9	2	0	5	7	3	4	5	8	5	5	5	10	6

При *k=1*, *k*-я порядковая статистика равна 0. При *k=3*, *k*-я порядковая статистика равна 2.

Конечно, чтобы найти порядковую статистику, сначала можно отсортировать массив (например, при помощи быстрой сортировки за *NlogN*), а затем вывести *k*-й элемент. Но оптимальный алгоритм поиска *k*- порядковой статистики работает за линейное в среднем время со сложностью *O(N)*.

Идея алгоритма схожа с алгоритмом быстрой сортировки. На вход алгоритма подаются *l* и *r* – левая и правая граница поиска *k* -ой порядковой статистики, число *k*. Перестановку элементов делаем так: находим первый «неправильный» элемент слева, затем первый «неправильный» элемент справа, и если не сошлись указатели обмениваем найденные значения. В зависимости от того, в какой части частично-упорядоченного массива находится индекс *k*, переходит в левую или правую часть.

```
int k_st(vector<int>a, int l, int r,int k)
{
     if (l>=r) return a[l];
     int i=l, j=r, p=a[(l+r)/2];
     while (i<j){
          while(a[i]<p) i++;
          while(a[j]>p) j--;
          if (i<j) {swap(a[i],a[j]); i++; j--;}
     }
    if (k>=l&&k<i)
          return k_st(a,l,i-1,k);
     else if (j<k&&k<=r)
          return k_st(a,j+1,r,k);
     else
          return p;
}
```

Рассмотрим работу функции на примере массива a[7]={8, 9, 9, 1, 7, 5, 5}. k=5.

8 9 9 1 7 5 5 l=0 r=6 p=1
1 9 9 8 7 5 5 l=1 r=6 p=8
1 5 5 7 8 9 9 l=4 r=6 p=9
1 5 5 7 8 9 9 l=4 r=5 p=8

k-я порядковая статистика равна 8. Заметим, что алгоритм нахождения k-й порядковой статистики ставит k-й элемент упорядоченного массива на свое место. Также с помощью этого алгоритма можно найти k наименьших чисел массива – они будут находиться в интервале [0, k], но не будут упорядочены.

Сортировка слиянием

Идея алгоритма заключается в разделении массива на два подмассива. Каждый подмассив

сортируется (к ним рекурсивно применяется сортировка слиянием). Полученные упорядоченные подмассивы объединяются в один отсортированный массив. Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы – в массиве будет находиться один элемент (массив из одного элемента упорядочен).

```
void Merge(vector<int>&tm,int l1,int m2,int r)
{
     int k1=l1, k2=m2+1;
     vector<int>temp;
```

```
     while (k1<=m2&&k2<=r){
          if (tm[k1]>tm[k2])
               temp.push_back(tm[k2++]);
          else
               temp.push_back(tm[k1++]);
     }
     while (k1<=m2) temp.push_back(tm[k1++]);
     while (k2<=r) temp.push_back(tm[k2++]);
     for (int i=l1;i<=r;++i) tm[i]=temp[i-l1];
}

void MergeSort(vector<int>&t,int l, int r)
{
     if(l>=r) return;
     int m=(l+r)/2;
     MergeSort(t,l,m);
     MergeSort(t,m+1,r);
     Merge(t,l,m,r);
}
```

Вызов сортировки

```
     vector<int>a(n);
     for (int i=0;i<n;++i)
     {
          a[i]=rand()%11;
          printf("%d",a[i]);
     }
     MergeSort(a,0,a.size()-1);
```

Алгоритм имеет сложность $O(Nlog\ N)$.

Сортировка подсчётом

Сортировка QuickSort использовала операции сравнения элементов. Сортировка подсчетом (Counting Sort) не использует операции сравнения, и применяется для массива **A** дискретных данных (например, целых чисел, символов), которые принимают значения из небольшого диапазона $a_i = [0,\ K - 1]$. Для данного алгоритма применяют вспомогательный массив **C**, элементы которого $c[i] = count(A,\ i)$, где $count(A,\ i)$ - количество элементов в массиве A, равных значению ***i***. Размер массива C равен максимальному элементу массива A.

Приведем пример исходного массива A и вспомогательного массива C.

a[i]	2	5	3	0	2	3	0	3

i	0	1	2	3	4	5	6	7
c[i]	2	1	3	2	2	3	2	3

По массиву **C** легко получить упорядоченный массив **A**. Делая проход по массиву C, мы столько раз последовательно включаем в массив **A** элемент ***i***, сколько раз он встречался в **A,** а именно c[*i*] раз.

```
for (int i=0;i<n;++i)
     c[a[i]]++;
int l=0;
for (int i=0;i<k;++i){
          for (int j=1;j<=c[i];++j){
               a[l]=i;
               l++;
          }
}
```

Массив использует O(K) дополнительной памяти и имеет сложность O(N+K). Сортировка подсчетом может быть реализована так, чтобы она обладала свойством стабильности.

Известны и другие сортировки, не использующие операции сравнения, например поразрядная сортировка.

Поразрядная сортировка

Рассмотрим идею поразрядной сортировки массива чисел. Например, необходимо выполнить сортировку массива трехразрядных чисел a[5]=(523, 153, 088, 554, 235). k=3 – количество разрядов. Числа даны в десятичной системе счисления – m=10. Дополнительно потребуются 10 векторов, отвечающие за цифру обрабатываемого разряда. Записываем числа исходного массива в векторы, отвечающие за цифры: 0, 1, 2, …, 9 в младшем разряде.

523, 153 – числа в векторе, отвечающем за цифру 3 .

554 - число в векторе, отвечающем за цифру 3.

235 – число в векторе, отвечающем за цифру 5.

088 -число в векторе, отвечающем за цифру 8.

Помещаем числа из вспомогательных векторов в исходный вектор.

Получаем: 523, 153, 554, 235, 088.

Производим распределение чисел во вспомогательные векторы по второму разряду.

523

235

153

554

088

Помещаем числа из вспомогательных векторов в исходный вектор.

Получаем: 523, 235, 153, 554, 088.

Производим распределение чисел во вспомогательные векторы по старшему разряду.

088

153

235

523

554

Помещаем числа из вспомогательных векторов в исходный вектор.

Получаем: 088, 153, 235, 523, 554. Получили отсортированный массив. Сложность сортировки O(kn+km). Используется дополнительная память O(n+m).

Квадратичные сортировки. Справочная информация

В качестве справочной информации, приведем программные реализации некоторых квадратичных сортировок, то есть сортировок, которые имеют сложность - $O(N^2)$. Приведенные ниже сортировки используют только сравнения и перестановки соседних элементов. Пузырьковая сортировка, сортировка вставками и выборочная сортировка не используют дополнительной памяти и являются стабильными.

Название сортировки	**Пример реализации**

Пузырьковая сортировка
BubbleSort
Сложность - $O(N^2)$.
	`for (int i=0;i<n-1;++i){`
	`  for (int j=0;j<n-i-1;++j){`
	`  if(a[j+1]<a[j])`
	`  swap (a[j],a[j+1]);`
	`  }`
	`}`

Сортировка вставками
InsertSort
Сложность в среднем $O(N^2)$, в лучшем случае $O(N)$.
Может сортировать элементы в процессе их ввода
	`  int n,x,j; int a[100];`
	`  cin>>n;`
	`  cin>>x;`
	`  a[0]=x;`
	`for (int i=1; i<n;++i){`
	`  cin>>x;`
	`  for (j=i-1; j>=0&&a[j] > x; j--)`
	`  a[j+1]=a[j];`
	`  a[j+1]=x;`

Выборочная сортировка
SelectSort
Сложность - $O(N^2)$
	`for(int i =0 ; i<n; ++i){`
	`  min=a[i];  min_i=i;`
	`  for(int j = i+1; j<n; ++j)`
	`  if(a[j]<min){`
	`  min = a[j]; min_i = j;`
	`  }`
	`  swap(a[min_i],a[i]);`
	`  }`

Для файлов небольших размеров сортировка вставками и сортировка выбором работают примерно в два раза быстрее пузырьковой сортировки, однако время работы любого из этих алгоритмов квадратично зависит от размера файла. Поэтому ни один из этих методов не нужно использовать для сортировки больших случайно упорядоченных файлов.

Лекция 8

Введение в динамическое программирование: одномер-ная и двумерная динамика.

Рекуррентные соотношения. Динамическое программирование: основные определения. Одномерная динамика: подсчет количества вариантов решения, поиск оптимального решения. Восстановление ответа. Двумерная динамика. Задача о рюкзаке

Общее представление

Задачи, в которых при наличии нескольких возможных вариантов решения, предполагается выбор в качестве ответа одного наилучшего (оптимального) решения, теоретически можно решать перебором всевозможных вариантов и выбором среди них наилучшего. При больших размерностях входных данных перебор всех вариантов (так называемый метод «грубой силы») не будет эффективен по времени. В этом случае можно попробовать решить задачу большей размерности при помощи решения подзадач меньшей размерности. Такой метод решения *называется динамическим программированием*.

Для того, чтобы задача могла быть решена методом динамического программирования должны быть выполнены следующие условия: *в задаче можно выделить подзадачи аналогичной структуры меньшего размера*; среди выделенных подзадач *есть тривиальные*, то есть имеющие «малый размер» и очевидное решение; *оптимальное решение* подзадачи большего размера может быть построено *из оптимальных решений подзадач*; *решения подзадач запоминаются в таблицы*, имеющие разумные размеры.

Задачи динамического программирования делятся на два типа. *Подсчет количества вариантов решения*. В этом случае находят суммарное количество решений подзадач. *Поиск оптимального решения (оптимизация целевой функции)*. В этом случае выбирают лучшее среди всех решений подзадач.

Одномерная динамика. Решение задач

Рассмотрим примеры двух указанных выше типов задач на примере задач, решаемых с помощью одномерной динамики, то есть динамики, в которой используется один параметр.

Подсчет количества вариантов решения. Последовательность из 0 и 1.

Требуется подсчитать количество последовательностей длины ***N*** , состоящих из 0 и 1, в которых никакие две единицы не стоят рядом.

Если воспользоваться полным перебором, то, например, для N=64 потребуется сгенерировать 2^{64} последовательностей из нулей и единиц и проверить каждую из них на соответствие условию задачи, что неэффективно по времени. В общем случае получаем сложность такого ния - $O(N2^N)$. Рассмотрим решение задачи с помощью динамического программирования.

Приведем примеры последовательностей, для некоторых небольших чисел N.

N=1	N=2	N=3	N=4
0	00 01	000, 001 010	0000, 0001, 0010 0100, 0101
1	10	100, 101	1000, 1001, 1010
2варианта	3 варианта	5 вариантов	8 вариантов

Из таблицы видим, что к каждой последовательности длиной ***(N-1)*** можно приписать ***0*** в любом случае и ***1*** только в том случае, если она заканчивается на ***0*** (по условию задачи). То есть, каждая последовательность длины ***(N-1)*** может быть продолжена либо одним, либо двумя способами.

Обозначим за ***i*** длину последовательности. *F(i)* – количество последовательностей длины ***i***, удовлетворяющих условию задачи.

Присвоим начальные значения динамики: F[1]=2; F[2]=3. Это будет база динамики.

Правило перехода динамики – F(i+1)=F(i)+F(i-1) позволяет получить ответ для большей подзадачи на основе ответов к предыдущим подзадачам. Заметим, что мы получили правило динамики такое же, как и для вычисления чисел Фибоначчи.

Ответом к задаче будет являться число F(N).

Сложность такого решения O(N).

Подсчет количества вариантов решения. Оптимизация линейной динамики

Задача вычисления чисел Фибоначчи и ей подобные решаются при помощи линейной динамики. Если параметр N является слишком большим (например, порядка 10^{18}), то даже линейная динамика не будет давать оптимальное по времени решение. Последовательность чисел Фибоначчи является возрастающей. Приведем оценку верхней границы N-го числа Фибоначчи.

Оценка скорости роста чисел Фибоначчи: $F_N \geq 2^{\frac{N}{2}}$, для $N \geq 6$.

Оценку можно получить, применив метод математической индукции.

База индукции: $F_6 = 8 = 2^{\frac{6}{2}}$, $F_7 = 13 > 2^{\frac{7}{2}} = 8\sqrt{2}$.

$n-1$ $n-2$ $n-2$

Переход индукции. При $n \geq 8, F_n = F_{n-1} + F_{n-2} \geq 2^{\frac{}{2}} + 2^{\frac{}{2}} >= 2 \cdot 2^{\frac{}{2}} = 2^{n/2}$.

Таким образом, $F_N \geq 2^{\frac{N}{2}}$

Рассмотрим на примере простейшей задачи вычисления чисел Фибоначчи *оптимизацию линейной динамики.* По условию задачи F[0]=0; F[1]=1. F(i+1)=F(i)+F(i-1). Для вычисления следующего элемента достаточно знать значение двух предыдущих элементов. Два предыдущих значения образуют вектор (F(i-1), F(i)). На следующем шаге мы получаем вектор (F(i), F(i+1)). Чтобы из первого вектора получить второй мы можем умножить его на матрицу $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. Проверим корректность перехода. Действительно, $(\mathrm{F(i-1)}, \mathrm{F(i)}) \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (\mathrm{F(i)}, \mathrm{F(i+1)})$. Получаем:

$$(\mathrm{F(N-1)}, \mathrm{F(N)}) = (\mathrm{F(N-2)}, \mathrm{F(N-1)}) \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \ldots = (\mathrm{F(0)}, \mathrm{F(1)}) \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{N-1} = (\mathrm{F(0)}, \mathrm{F(1)}) A^{N-1}.$$

Возведение матрицы в степень, то есть, вычисление A^{N-1} вычисляется при помощи бинарного возведения в степень. Затем вектор $(\mathrm{F(0)},\ \mathrm{F(1)})$ просто умножается на полученную матрицу. Такое решение будет работать за время $O(\log(N)\ K^3)$, где N – показатель степени матрицы (номер числа Фибоначчи), К – размерность матрицы. В рассмотренной задаче случае К=2. Таким образом, получаем лучшую асимптотику вычисления N-го числа Фибоначчи. Заметим, что при таком подходе мы можем вычислить только N-е число, а не все числа.

Определение оптимального решения. Задача о кузнечике, который собирает монеты

Задачи, в которых требуется определить наилучшее решение среди возможных (производится поиск оптимального решения), также удобно решать при помощи динамического программирования. Проиллюстрируем применение метода динамического программирования для поиска оптимального решения на конкретной задаче.

Задача. Кузнечик собирает монеты

Кузнечик прыгает по столбикам, расположенным на одной линии на равных расстояниях друг от друга. Столбики имеют порядковые номера от 1 до N. Вначале Кузнечик сидит на столбике с номером 1. Он может прыгнуть вперед на расстояние от 1 до K столбиков, считая от текущего.

На каждом столбике Кузнечик может получить или потерять несколько золотых монет (для каждого столбика это число известно). Определите, как нужно прыгать Кузнечику, чтобы собрать наибольшее количество золотых монет. Учитывайте, что Кузнечик не может прыгать назад.

Входные данные

В первой строке вводятся два натуральных числа: N и K ($2 \leq N$, $K \leq 10000$), разделённые пробелом. Во второй строке записаны через пробел N - 2 целых числа – количество монет, которое Кузнечик получает на каждом столбике, от 2-го до N - 1 -го. Если это число отрицательное, Кузнечик теряет монеты. Гарантируется, что все числа по модулю не превосходят 10000.

Выходные данные

В первой строке программа должна вывести наибольшее количество монет, которое может собрать Кузнечик. Во второй строке выводится число прыжков Кузнечика, а в третьей строке – номера всех столбиков, на которых побывал Кузнечик (через пробел в порядке возрастания).

Если правильных ответов несколько, выведите любой из них.

Примеры

Входные данные	Выходные данные
5 3 2 -3 5	7 3 1 2 4 5

Для решения задачи будем использовать массив динамики d[], где d[i] – наибольшее количество золотых монет, которое может собрать кузнечик, находясь в данный момент на столбике i.

База динамики. Начальное заполнение массива динамики d[1]=0 – если кузнечик находится на первом столбике, то он собрал 0 монет.

Правило перехода динамики d[i]=d[num_max]+a[i], где num_max – номер столбика с мак-

симальным количеством собранных кузнечиком монет, который предшествовал данному. То есть, кузнечик, находясь на столбике с номером *i*, может собрать такое максимальное количество монет, которое равно сумме предыдущего максимального количества монет и количеству монет на текущем столбике.

Ответом к задаче будет являться число d[n], где n – номер последнего столбика.

```
     d[1]=0; a[1]=0; a[n]=0;
     for (int i=2;i<=n;++i){
          int num_max = i - 1;
//поиск предыдущего столбика с максимальным количеством монет
          for(int j=max(1,i-k);j<=i-1;++j)
               if (d[j]>d[num_max]) {
                    num_max=j;
               }
          d[i]=d[num_max]+a[i]; //Текущее максимальное значение
          p[i] = num_max;
     }
     printf("%d\n", d[n]);
```

Дополнительно в программе используется массив p[], где p[i] – номер столбика с максимальным количеством монет, предшествующего i-му столбику. Этот массив будет использоваться для восстановления ответа – перечисления всех номером столбиков, на которых побывал кузнечик.

Восстановление ответа

В задаче про кузнечик и многих других нужно не только дать ответ в виде количества способов решения или наилучшего решения, но и указать всевозможные способы или способ достижения этого наилучшего решения. Например, в задаче про кузнечика требовалось указать номера столбиков, по которым перемещался кузнечик. В программе был использован массив предков p[], который для каждого i столбика хранил значение p[i] – номер того столбика, с которого кузнечик прыгнул на i столбик. Теперь мы просто можем переходить к предыдущему столбику с конца решения вот таким образом - i=p[i]. Это позволит восстановить ответ – указать цепочку столбиков, на которых был кузнечик. Получаемые на каждом шаге цикла номера столбиков будем записывать в массив ans. Поскольку заполнение массива происходит, начиная с последнего столбика и, заканчивая первым, то вывод вектора ans нужно осуществлять в обратном порядке. Приведем фрагмент программы.

```
     int i=n;i
     vector<int> ans;
     ans.push_back(i);
     do   {
          i=p[i];
          ans.push_back(i);
     }
     while (i>1);
     printf("%d\n",ans.size()-1);
     for(auto i=ans.rbegin();i!=ans.rend();++i)
          printf("%d ",*i);
```

Для теста, указанного в задаче проиллюстрируем заполнение массива динамики d и массива предков p. Входные данные:5 – количество столбиков, 3 – количество прыжков, которые может делать кузнечик (от 1 до 3), 2 -3 5 – количество монет, которые лежат на столбиках.

i - номер столбика	**1**	**2**	**3**	**4**	**5**
a[i] - количество монет	0	2	-3	5	0
d[i]	0	2	-1	7	7
p[i]	-1	1	2	2	4

Из заполненной таблицы следует, что d[5]=7 – максимальное количество собранных монет. Восстановление ответа (столбиков, на которых был кузнечик) проходит следующим образом: 5 столбик – последний, на котором он был; его предком является столбик 4; в свою очередь его

предком является столбик 2, а его предком, столбик 1. То есть получается последовательность элементов вектора ans: 5, 4, 2, 1. Выведем ее в обратном порядке и получим: 1, 2, 4, 5.

Задача о наибольшей возрастающей подпоследовательности

Рассмотрим классическую задачу динамического программирования о наибольшей возрастающей подпоследовательности (НВП). В ней требуется найти оптимальное решение и восстановить ответ.

Дана последовательность, требуется найти длину её наибольшей возрастающей подпоследовательности. Подпоследовательностью последовательности называется некоторый набор её элементов, не обязательно стоящих подряд.

Входные данные

В первой строке входных данных задано число N - длина последовательности ($1 \le N \le 1000$). Во второй строке задается сама последовательность (разделитель - пробел). Элементы последовательности - целые числа, не превосходящие 10000 по модулю.

Выходные данные

Требуется вывести длину наибольшей строго возрастающей подпоследовательности и саму наибольшую возрастающую подпоследовательность.

Разбор решения задачи. Определим для каждого i – номера элемента в последовательности ту наибольшую возрастающую подпоследовательность, которая построена из элементов промежутка $[a_0,..,a_i]$ и заканчивается элементом a_i. В массиве d[] будем в качестве d[i] хранить количество элементов в соответствующей НВП, заканчивающейся элементом a[i].

База динамики. d[0]=1. Длина НВП для одного первого символа равна 1. В качестве НВП в данном случае выступает сам элемент.

Правило перехода динамики. Предположим, что каждое d[i] – уже вычисленное для каждого элемента i оптимальное значение длины НВП, состоящей из подходящих элементов последовательности $[a_0,..,a_i]$ и заканчивающейся элементом a[i]. Чтобы определить $d[i+1]$ надо посмотреть все предыдущие элементы $d[0..i]$, для которых выполняется a[i]<a[i+1], выбрать наибольший такой элемент d[k] и прибавить к нему 1. То есть, найти в предыдущей последовательности наибольшую длину НВП, заканчивающуюся элементом, меньшим, чем текущий элемент a[i] последовательности. $d[i+1] = max_{0 \le k \le i}(d[k]|a[k]<a[i])+1$.

Ответом к задаче будет наибольшее число массива d[].

Проиллюстрируем решение примером. Дана последовательность a[]={1 4 2 5 6 3 7}. Рассмотрим изменение массива d[].

i	0	1	2	3	4	5	6
a[i] Массив	1	4	2	5	6	3	7
d[i] Динамика	1	2	2	3	4	3	5
НВП. Пример конкретной НВП	1	1, 4	1, 2	1, 2, 5	1, 2, 5, 6	1, 2, 3	1, 2, 5, 6, 7

Ответом для задачи будет длина НВП, равная 5. Это максимальное значение среди элементов массива d[]. Для восстановления ответа, начиная с $max_{i=0..n}d[i]$ (ответа к задаче), будем двигаться к началу массива динамики. Каждый раз, осуществляя поиск элемента массива динамики, на единицу меньшего, чем текущий элемент, перемещаемся к началу массива d[] до тех пор, пока не дойдем до какого-либо d[i]=1. Второй способ восстановления ответа заключается в том, чтобы хранить в массиве last[i] значение индекса предпоследнего элемента в НВП, заканчивающейся i-м элементом. Вначале значение элементов массива last[] заполняются значениями (– 1) – барьерами, указывающими предварительно на то, в НВП нет элементов, предшествующих данному элементу. В приведенном примере last={0,2,3,4}. По такому вспомогательному массиву легко восстанавливается ответ: a[0]=1, a[2]=2, a[3]=5, a[4]=6, a[6]=7.

Фрагмент кода программы, соответствующий поиску оптимального решения (поиску длины НВП) приведен ниже.

```
d[0]=1; last[0]=-1; //База динамики
for (int i=1;i<n;++i){ // Правило перехода динамики
     max=0; last[i]=-1;
     for (int k=i-1; k>=0; --k){
          if (d[k]>max&&a[k]<a[i]) {
               max=d[k];
               last[i]=k;  //Запоминаем номер предпоследнего
          }                //элемента НВП
     }
          d[i]=max+1;
}
```

Для вывода ответа – длины НВП, необходимо вывести максимальное значение динамики.

```
max=d[1];
for (int i=0;i<n;++i)
     if (d[i]>max) {
               max=d[i]; l=i; //Индекс элемента,
     }                        //на который заканчивается НВП
```

Восстановление ответа при помощи массива last[]. В ответе ans[] помещаются элементы в обратном порядке.

```
ans.push_back(a[l]); //добавили последний элемент НВП
while(last[l]!=-1){ //Пока не дошли до барьерного элемента
   ans.push_back(a[last[l]]);//добавляем в ans предыд.эл.НВП
     l=last[l];              //переход на индекс пред.эл.НВП
}
for (auto i=ans.rbegin();i!=ans.rend(); ++i)
     cout<<*i<<" ";  //Вывод ответа в обратном порядке
```

Сложность решения $O(N^2)$.

Двумерная динамика. Задача о рюкзаке

В одномерной динамике для правил перехода динамики используется одномерный массив динамики. Двумерная динамика использует два параметра динамики и соответственно двумерные массивы динамики. Рассмотрим классическую задачу о рюкзаке, на примере которой проиллюстрируем двумерное динамическое программирование.

Дано N предметов массой m_1, …, m_N и стоимостью c_1, …, c_N соответственно.

Ими наполняют рюкзак, который выдерживает вес не более M. Какую наибольшую стоимость могут иметь предметы в рюкзаке?

Входные данные

В первой строке вводится натуральное число N, не превышающее 100 и натуральное число M, не превышающее 10000.

Во второй строке вводятся N натуральных чисел m_i, не превышающих 100.

Во третьей строке вводятся N натуральных чисел c_i, не превышающих 100.

Выходные данные

Выведите одно целое число: наибольшую возможную стоимость рюкзака.

Примеры

Входные данные	Выходные данные
4 6 2 4 1 2 7 2 5 1	13
Комментарии. Дано 4 предмета, имеющих параметры (масса, стоимость): {(2,7), (4,2), (1,5), (2,1)}. Рюкзак выдерживает вес не более 6 кг. Чтобы набрать в рюкзак предметы наибольшей стоимости оптимальнее всего взять предметы {(2,7), (1,5), (2,1)}. В этом случае рюкзак будет иметь массу 5 кг и стоимость 13.	

Разбор решения задачи

Информацию о предметах будем хранить в векторе пар s(n), каждый элемент которого – пара (масса, стоимость): Двумерный массив динамики d состоит из элементов d[i][j], где d[i][j] - максимальное значение стоимости рюкзака в случае возможности использования первых i предметов, которыми наполняется рюкзак вместимостью j. 0<=i<=N, 0<=j<=m. Размер таблицы динамики *n x m.*

База динамики. Начальное заполнение массива d[i][j]=0. Первоначальные стоимости рюкзака равны нулю.

Правило перехода динамики. Каждый предмет либо будет помещен в рюкзак, либо не будет помещен в него. Тип данной задачи – рюкзак без повторений. В случае типа задачи – рюкзак с повторениями, каждый предмет может быть использован необходимое количество раз. Оптимальное значение d[i][j] выбирается как максимум из двух возможных случаев:

- предмет i не помещают в рюкзак. В этом случае стоимость рюкзака вместимостью j остается такой же, какой она была без данного предмета, то есть d[i-1][j]. Разумеется, предмет не помещается в рюкзак если масса предмета больше вместимости рюкзака.
- предмет i помещается в рюкзак. В этом случае стоимость рюкзака вычисляется как сумма стоимости рюкзака без данного предмета и стоимости этого предмета d[i-1][j-m_i]+c_i.

Таким образом, получаем правило перехода динамики. d[i][j]=max {d[i − 1][j], d[i − 1][j − mi] + ci}.
Ответом к задаче будет являться элемент d[n][m].

Приведем фрагмент программы, иллюстрирующий правило перехода динамики.

```
  for(int i = 1; i<=n; ++i){//перебор по предметам
    for(int j = 1; j<=m; ++j){//перебор по массе рюкзака
//если предмет имеет допустимую массу для помещения его в рюкзак
//и если предмет выгоднее поместить в рюкзак, то помещаем его
     if((j>=s[i].first)&&(d[i-1][j]<(d[i-1][j-
s[i].first]+s[i].second)))
                d[i][j]=d[i-1][j-s[i].first]+s[i].second;
            else
//если не выгодно добавлять предмет в рюкзак, то не добавляем его
                d[i][j] = d[i-1][j];
        }
    }
```

Сложность алгоритма $O(n \times m)$.

Рассмотрим иллюстрацию решения задачи о рюкзаке на примере.

W=10 – вместимость рюкзака. N=4 - количество предметов.

1 предмет. m1=6 c1=30

2 предмет. m2=3 c2=14

3 предмет. m3=4 c3=16

4 предмет. m4=2 c4=9

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0	0	9	14	16	23	30	30	39	44	46

Максимальная стоимость рюкзака – 46.

В таблице цветом отображено восстановление ответа – выделены ячейки, которые представляют собой оптимальную промежуточную стоимость рюкзака d[i][j].

Получим оптимальный набор вещей: в рюкзак помещают 1 и 3 предмет.

LR динамика

В случае обработки строк и поиска в таких задачах оптимального решения удобно использовать подход LR динамики (динамики по подстрокам). L – левая граница рассматриваемой подстроки, R – правая граница рассматриваемой подстроки. Решение подзадач представляет собой поиск оптимального решения для подстрок строки. Рассмотрим применение LR динамики на примере конкретной задачи.

Задача. Удаление скобок

Дана строка, составленная из круглых, квадратных и фигурных скобок. Определите, какое наименьшее количество символов необходимо удалить из этой строки, чтобы оставшиеся символы образовывали правильную скобочную последовательность.

Входные данные

Строка из круглых, квадратных и фигурных скобок. Длина строки не превосходит 100 символов.

Выходные данные

Выведите строку максимальной длины, являющуюся правильной скобочной последовательностью, которую можно получить из исходной строки удалением некоторых символов. Если возможных ответов несколько, выведите любой из них.

Примеры

Входные данные	Выходные данные
([)]	[]
{{[(]{)})]	[({})]

Разбор решения задачи

Считываем скобочную последовательность в строку s. Двумерный массив динамики состоит из элементов dp[i][j] – наименьшего количества символов, которые надо удалить из подстроки s[i..j], левый символ которой имеет позицию *i*, правый символ – *j*. Размер таблицы $n \times n$, где n – длина строки *s*.

База индукции - dp[i][i]=1. Если строка состоит из одной скобки (левая граница подстроки совпадает с правой границей), то нужно удалить эту единственную скобку. Заметим, что в данной задаче начальное заполнение двумерной таблицы динамики проходит по диагонали. Значения dp[i][j] при i>j заполняются нулями (левая граница строки не может быть больше правой границы).

Правило перехода динамики. Рассмотрим сначала отдельно случай, когда в строке s[l..r] на позициях l и r стоят соответствующие друг другу по типу открывающаяся и закрывающаяся скобки. В этом случае количество удаляемых скобок будет равно этому количеству для подстроки s[l+1..r-1]. Например, для строки ({{}}) количество удаляемых скобок равно количеству удаляемых скобок в последовательности {{}} – 2 скобки. Исключение составляют ситуации совокупности изначально правильных последовательностей, например (){}(). Если переходить к подстроке s[l+1..r-1] =")}{(", то получим ответ 2. В то же время, ответ к строке (){}() – 0 (ноль) удаляемых скобок. Поэтому, нужно учесть такого рода ситуации для строк s[l..r] с соответствующими друг другу по типу открывающейся и закрывающейся скобками, и выбрать наименьшее значение – или из таблицы динамики, или при переходе к подстроке s[l+1..r-1].

Рассмотрим общий случай. Чтобы вычислить d[l][r] – наименьшее количество скобок, которое нужно удалить из подстроки s[l..r] разобьем всевозможными способами строку s[l..r] на две подстроки s[l,k] и s[k+1,r], где k=l…r-1. Очевидно, что если подстроки s[l,k] и s[k+1,r] сделать правильными скобочными последовательностями, удалив лишние строки в них, то и строка s[l..r] станет правильной скобочной последовательностью. Поэтому остается найти минимальное значение суммарного количества удаляемых скобок для всевозможных разбиений строки на две подстроки

– $min_{l \le k \le r-1}(s[l,k] + s[k+1,r])$. Это значение и будет давать наименьшее количество удаляемых скобок в строке s[l..r].

Рассмотрим заполнение таблицы динамики для конкретного примера.

Входные данные. ({ {) } } – скобочная последовательность.						
L\R	0	1	2	3	4	5
0	1	2	3	2	3	2
1	0	1	2	3	2	1
2	0	0	1	2	1	2
3	0	0	0	1	2	3
4	0	0	0	0	1	2
5	0	0	0	0	0	1

Значение динамики	Значение динамики
Пример вычисления значения d[0][5]	Min({0,0}+{1,5},{0,1}+{2,5},{0,2}+{3,5},{0,3}+{4,5}, {0,4}+{5,5}) = 2

Приведем ниже фрагмент программы решения задачи. Заметим, что матрица динамики заполняется по столбцам, начиная с диагонали каждого столбца (в противном случае придется обращаться к невычисленным еще значениям). В программе используется вспомогательный массив, элементы которого ep[l][r] хранят индексы k, указывающие на оптимальное разбиение строк $s[l,k]$ на две подстроки $s[l,k]$ и $s[k+1,r]$.

```
int n = s.size();
     for (int r = 0; r < n; ++r)
     for (int l = r; l >= 0; --l){
          if (l == r)
               dp[l][r] = 1; // База динамики
          else{
               int best = 1000000; int mk = -1;
               if (s[l] == '(' && s[r] == ')' || s[l] == '[' &&
s[r] == ']' || s[l] == '{' && s[r] == '}')
               //Случай соответствующих скобок
               best = dp[l + 1][r - 1];
               //Общий случай правила перехода динамики
               for (int k = l; k < r; ++k)
               if (dp[l][k] + dp[k + 1][r] < best){
                    best = dp[l][k] + dp[k + 1][r];
                    mk = k; //поиск оптимального разбиения строки
               }
               dp[l][r] = best; ep[l][r] = mk;
          }
     }
```

Восстановление ответа для задачи про удаление скобок удобно реализовать при помощи рекурсивной процедуры rec(int l, int r). Выход из рекурсии происходит в том случае, если из скобочной последовательности нужно удалить все скобки. Если из последовательности s[l..r] ни одной скобки удалить не нужно, то это правильная скобочная последовательность – в этом случае печатается полностью подстрока s[l..r] и осуществляется выход из рекурсии.

```
void rec(int l, int r){
     if (dp[l][r] == r - l + 1)
          return;
```

```
    if (dp[l][r] == 0){
        cout << s.substr(l, r - l + 1);
        return;
    }
    if (ep[l][r] == -1) {//Если подстрока имеет в начале и конце
                  //соответствующего типа правильные скобки,
        cout << s[l]; //то печатаем левую скобку
        rec(l + 1, r - 1);//вызов рекурсию вложенной подстроки
        cout << s[r];// печатаем правую скобку
        return;
    }
    rec(l, ep[l][r]); //вызов рекурсии от левой подстроки
    rec(ep[l][r] + 1, r);//вызов рекурсии от правой подстроки
}
```

В основном тексте программы процедура rec() вызывается для всей строки.

```
    rec(0, n - 1); // Вызов рекурсии в основном тексте программы
```

По такому же принципу, как и рассмотренная задача, решается задача добавления наименьшего количества скобок в скобочную последовательность для того, чтобы она стала правильной.

Комбинаторный перебор и рекурсия, алгоритмы STL для организации перебора.

Переборные задачи. Классы сложности задач P и NP. Рекурсивные алгоритмы. Дерево рекурсивных вызовов на примере рекурсивного вычисления чисел Фибоначчи. Пример нерекурсивного и рекурсивного алгоритма вычисления a^n со сложностью O(n) и O(log n). Комбинаторные объекты: перестановки, сочетания, размещения, сочетания с повторениями, размещения с повторениями. Перебор перестановок: рекурсивный и нерекурсивный алгоритмы. Генерация m-размещений без повторений и с повторениями. Генерация всех m-элементных подмножеств n-элементного множества (генерация m-сочетаний без повторений и с повторениями). Треугольник Паскаля. Правильные скобочные последовательности. Числа Каталана. Перебор разложений числа в сумму. Алгоритмы STL для организации перебора. Примеры олимпиадных задач.

Общее представление

С содержательной точки зрения комбинаторный анализ представляет собой совокупность задач, связанных с нахождением числа объектов, обладающих определенным перечнем свойств. Например, всем известные задачи о разложении числа в сумму всевозможных слагаемых, о генерации всех m-элементных подмножеств n-элементного множества, о раскраске вершин графа, о «счастливых билетах», являются комбинаторными задачами. К основным комбинаторным объектам относятся: перестановки, размещения без повторений и с повторениями, сочетания. Математические понятия о данных объектах, алгоритмы их генерации будут рассмотрены в этой лекции.

При решении комбинаторных переборных задач удобно использовать рекурсивные алгоритмы. Поэтому рассмотрение темы будет начато с изложения логики работы рекурсивных алгоритмов на конкретных примерах. Кроме того, в Visual Studio представлены алгоритмы STL для организации перебора, которые могут помочь в быстром решении несложных олимпиадных задач.

Переборные задачи. Классы сложности P и NP

Задачи, которые можно решать при помощи алгоритмов, имеющих полиномиальную сложность $O(n^k)$, относятся к задачам *класса сложности P* (так называемым, быстро разрешимым

задачам). В обозначениях полиномиальной сложности $O(n^k)$, n – это размер задачи, определяемый входными данными, k- некоторое положительное целое число. Задачи, которые решаются при помощи полного перебора, относятся к классу задач класса NP. Для этих задач не удается найти полиномиальный алгоритм решения, поэтому их также называют *трудноразрешимыми*. В большинстве своем это переборные задачи, которые характеризуются экспоненциальным множеством вариантов, среди которых нужно найти решение. Такие задачи могут решаться при помощи полного перебора, что возможно только для небольших размеров задачи, в противном же случае - с ростом размера задачи число вариантов быстро растет, и задача становится практически неразрешимой методом полного перебора.

Примерами задач класса P являются задачи: умножения матриц, например, со сложностью $O(n^3)$; сортировки массивов; нахождения эйлерова цикла в графе из m ребер (обход графа, начиная с какой-то вершины и заканчивая в ней самой, таким образом, чтобы каждое ребро было посещено по одному разу). *Примерами задач класса NP* являются задачи: коммивояжера (коммивояжер хочет объехать все города, побывав в каждом ровно по одному разу, и вернуться в город, из которого начато путешествие. Известно, что переезд из города i в город j стоит c(i,j) рублей, требуется найти путь минимальной стоимости.

Одна из проблем теоретической информатики заключается в том, верно ли равенство P=NP? Неформально говоря, постановка задачи заключается в следующем: можно ли быстро решить всякую переборную задачу, или, иначе говоря, можно ли найти полиномиальный алгоритм для задач класса NP. В настоящее время таких алгоритмов указать не удалось. Если для некоторой задачи удалось доказать ее NP-полноту, то есть основания считать ее практически неразрешимой. В этом случае лучше построить приближенный алгоритм решения задачи.

Рекурсивные алгоритмы

Рассмотрим рекурсивные алгоритмы на примере задачи вычисления a^n - степени числа. Первый способ вычисления основан на однопроходном итерационном алгоритме:

```
long long  p=1;
for (int i=1; i<=n;++i)
          p*=a;
```

Сложность работы алгоритма $O(n)$.

Второй способ вычисления степени основан на рекурсии. Рекурсия базируется на следующем рекуррентном соотношении: $a^0 = 1,\ a^n = a * a^{n-1}$. То есть, каждый раз функция рекурсивно вызывает саму себя с уменьшенным на единицу значением показателя степени. Таким образом, для вычисления a^n потребуется *n* рекурсивных вызовов.

```
long long power(int a,int n){
     if(n==0)
          return 1;  //Выход из рекурсии в случае нулевой степени
     else
          return a*power(a,n-1); //Рекурсивный вызов
}
```

Оказывается, существует и более оптимальный алгоритм, основанный на рекуррентном соотношении

a^n

$$= \begin{cases} a * a^{n-1}, \text{если } n - \text{нечетное} \\ (a^{n/2})^2, \text{если } n - \text{четное} \end{cases}$$

```
int power(int a,int n){
     if(n==0) return 1;  //Выход из рекурсии
     else
          if (n&1)       //Степень нечетная
               return a*power(a,n-1);
          else           //Степень четная
               return power(a*a, n/2);
}
```

Оценим количество рекурсивных вызовов при вычислении a^{100}. В дальнейших рассуждениях потребуется двоичное представление показателя степени - $100_{10}=1100100_2=64+32+4$.

$a^{100} = (a^2)^{50} = (a^4)^{25} = (a^4)^{24} * a^4 = (a^8)^{12} * a^4 = (a^{16})^6 * a^4 = (a^{32})^3 * a^4 =$
$= (a^{32})^2 * a^{32} * a^4 = a^{64} * a^{32} * a^4.$

Как видим, потребовалось 8 рекурсивных вызовов, а в двоичной записи показателя 100 семь разрядов: $\lceil log 100 \rceil = 7$. Всего же, в общем случае будет не более $2 \log n$ рекурсивных вызовов при вычислении степени. Таким образом, сложность работы алгоритма $O(\log n)$.

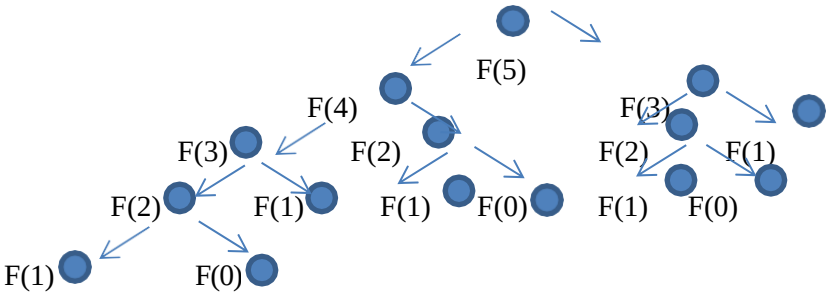
Использование стека в работе рекурсивных функций. Рекурсивные функции при каждом рекурсивном вызове используют *рекурсивный стек* – область памяти, в которую заносятся значения всех локальных переменных функции в момент рекурсивного обращения. Каждое такое обращение формирует один слой данных стека. При завершении вычислений по конкретному об ращению, из стека считывается соответствующий ему слой данных, и локальные переменные восстанавливаются, снова принимая значения, которые они имели в момент обращения. Максимальное количество слоев рекурсивного стека, заполняемых при конкретном вычислении значения рекурсивной функции, называют *глубиной рекурсивных вызовов*. Количество элементов полной рекурсивной траектории всегда не меньше глубины рекурсивных вызовов. Эта величина не должна превосходить максимального размера стека используемой вычислительной среды.

Вычисление чисел Фибоначчи

В разделе динамического программирования мы рассматривали вычисление чисел Фибоначчи при помощи рекуррентных соотношений, которые служили основой для базы и перехода динамики: F[0]=0, F[1]=1, F(i)=F(i-1)+F(i-2). Вычисление чисел Фибоначчи можно реализовать и рекурсивно.

```
int fib(int n){
     if (n<=1)
          return n;
     else
          return fib(n-1)+fib(n-2);
}
```

Для данной задачи рекурсивные вызовы образуют бинарное дерево, по которому можно оценить количество рекурсивных вызовов. На рисунке изображены рекурсивные вызовы для f(5), всего потребуется 14 рекурсивных вызовов. При вычислении нерекурсивным алгоритмом f(5) будет получено линейно за 5 итераций. Рекурсия в данном случае неэффективна, так как многие числа Фибоначчи вычисляются несколько раз, например на рисунке видно, что f(3) вычисляется дважды, f(2) трижды.



Обозначим за T(n) - количество строк (операций), выполняемых рекурсивной процедурой fib(). T(n)=2 для n<=1. T(n)=T(n-1)+T(n-2)+3, при n>1. То есть, T(n)>=F(n). К примеру, T(100)>3*10^20. Тактовая частота процессора 1GHz позволяет компьютеру производить 10^9 операций в секунду. Легко проверить, что рекурсивное вычисление числа F(100) в этом случае потребует более десяти тысяч лет.

Можем сделать вывод о необходимости правильного выбора алгоритмов для эффективного решения той или иной задачи.

Комбинаторные объекты. Основные математические определения

Перестановкой из n элементов (например, чисел $1, 2, 3, \ldots, n$) называется каждый упорядоченный набор из этих элементов. Для примера рассмотрим все перестановки при $n = 3$: $\{1, 2, 3\}, \{1, 3, 2\}, \{2, 1, 3\}, \{2, 3, 1\}, \{3, 1, 2\}, \{3, 2, 1\}$, получаем 6 вариантов. Количество перестановок из n элементов - $P_n = n!$ Действительно, пусть имеется множество $A = \{1, 2, 3, \ldots, n-1, n\}$, где $|A| = n$. Первый элемент множества можно выбрать n способами для включения его в перестановку, второй элемент - $n - 1$ способом, и так далее до последнего элемента, который можно выбрать одним способом. Получим по правилу комбинаторного произведения формулу вычисления количества перестановок.

$$P_n = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 = n!$$

Размещением из n элементов по m называется упорядоченный набор из m различных элементов некоторого n – элементного множества. Различают размещения без повторений и размещения с повторениями. В размещениях с повторениями элементы исходного множества могут включаться в набор несколько раз. Заметим, что перестановками являются размещения без повторений из n элементов по n.

Так, при при $n = 4$ и $m = 2$ получим следующие 12 размещений без повторений:
$\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 1\}, \{2, 3\}, \{2, 4\}, \{3, 1\}, \{3, 2\}, \{3, 4\}, \{4, 1\}, \{4, 2\}, \{4, 3\}$.

Приведем размещения с повторениями для $n = 3, m = 2$ (9 вариантов):
$\{1, 1\}, \{1, 2\}, \{1, 3\}, \{2, 1\}, \{2, 2\}, \{2, 3\}, \{3, 1\}, \{3, 2\}, \{3, 3\}$.

Число размещений без повторений из n по m обозначается как A_n^m, и равно: $A_n^m = \frac{n!}{(n-m)!}$.

Действительно, первый объект можно выбрать n способами, второй - $(n - 1)$ способом, последний m-й объект – $(n - m + 1)$ способом. Получим по правилу комбинаторного произведения формулу размещений без повторений.

$$A_n^m = n \times (n-1) \times (n-2) \times (n-m+1) = \frac{n!}{(n-m)!}.$$

Число размещений с повторениями из n по m обозначается как $\bar{A}_n^m$ и равно $\bar{A}_n^m = n^m$.

Действительно, каждый элемент подмножества можно выбрать ровно n способами, а всего элементов - m.

Сочетанием из n элементов по m, называется набор из m различных элементов некоторого n – элементного множества, причем их порядок следования в наборе не важен. Различают сочетания без повторений и сочетания с повторениями. В сочетаниях с повторениями элементы исходного множества могут включаться в набор несколько раз

Рассмотрим сочетания без повторений для $n = 4, m = 2$: $\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}$. Получили 6 сочетаний.

Числом сочетаний из n по m называется число всех m -элементных подмножеств - элементного множества и обозначается так: C_n^m, и равно $C_n^m = \frac{n!}{m!(n-m)!}$. Эта формула выводится из формулы размещений: $A_n^m = \frac{n!}{(n-m)!}$, но поскольку в сочетаниях не важен порядок следования элементов, то мы делим количество размещений на $m!$ – количество всевозможных способов, сколькими можно изменить порядок следования элементов набора из m элементов.

Число сочетаний без повторений из n по m равно биномиальному коэффициенту $\binom{n}{m}$. Напомним, что биномиальными коэффициентами называются коэффициенты в разложении бинома Ньютона: $(1+x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k$ или $(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$.

В качестве справочной информации приведем некоторые свойства сочетаний.

$$\binom{n}{0} = \binom{n}{n} = 1$$

$$\binom{n}{m} = \binom{n}{n-m}$$ – свойство симметрии

$$\binom{n}{m} = \binom{n-1}{m-1} + \binom{n-1}{m}.$$

$$\sum_{i=0}^{n} \binom{n}{i} = 2^n.$$

$$\binom{n+m}{k} = \sum_{s=0}^{k} \binom{m}{s} \binom{n}{k-s}$$ – тождество Коши.

Для вычисления биномиальных коэффициентов можно использовать треугольник Паскаля, по бокам и на вершине которого стоят единицы, а для построения треугольника используется тождество: $\binom{n}{m} = \binom{n-1}{m-1} + \binom{n-1}{m}$. Приведём первые строки треугольника Паскаля:

1 1
1 2 1
1 3 3 1
1 4 6 4 1

Каждая ячейка задаётся двумя координатами – номером строки – n и номером столбца в этой строке – m. Таким образом, число, стоящее в ячейке с координатами $(n; m)$ равно C_n^m. Нумерация строк и столбцов начинается с нуля.

Приведем пример для сочетаний с повторениями для $n = 3, m = 2$: {1, 1}, {1, 2}, {1, 3}, {2, 2}, {2, 3}, {3, 3}. Количество сочетаний с повторениями можно вычислить по формуле $\bar{C}_n^m = C_{n+m-1}^m$. Формула выводится на основе следующих соображений. Возьмем множество X={1, 2, 3, 4, 5,…}. Составим из элементов множества *X* мультимножество из m элементов (элементы в мультимножестве могут повторяться). Пусть в мультимножество входят элементы: $a_1 \le a_2 \le \ldots \le a_m$. Прибавим к каждому элементу, начиная со второго, числа 1, 2, 3, …, m-1. Получим строгое соотношение: $a_1 < a_2 + 1 < a_3 + 2 < \ldots \le a_m + m - 1$. Этот набор чисел есть набор из более широкого множества чисел, которое включает в себя (n+m-1) чисел. Из этого множества мы выбираем m чисел для составления сочетаний - C_{n+m-1}^m. Таким образом, получаем формулу $\bar{C}_n^m = C_{n+m-1}^m$.

Комбинаторные объекты. Алгоритмы генерации

На практике часто возникают задачи генерации комбинаторных объектов, а также задачи нахождения объекта по его номеру и задачи определения номера комбинаторного объекта. Рекурсивные программы являются удобным способом порождения комбинаторных объектов, и позволяют прозрачно и сжато записать алгоритм решения. В то же время, любой рекурсивный алгоритм тратит время на запоминание данных в незавершенных вызовах (используется рекурсивный стек) и поэтому не всегда является эффективным. Каждый рекурсивный алгоритм может быть представлен нерекурсивным итерационным образом, в этом разделе будут рассмотрены и рекурсивные, и нерекурсивные реализации алгоритмов генерации комбинаторных объектов. Для перестановок рассмотрены также задачи получения перестановки по ее номеру и номера для заданной перестановки.

Перестановки. Генерация всех перестановок. Рекурсивный алгоритм

Дано множество чисел {1, 2, 3,..,n}. Необходимо вывести на экран все перестановки этого множества. Очередная перестановка чисел (в общем виде каких-либо объектов) хранится в массиве a[]. Индексация элементов начинается с нуля. t – индекс элемента, который включается в очередную перестановку. Рекурсивная процедура печатает очередную полученную перестановку, когда t станет равным n-1 (набрали n элементов). Для того, чтобы сгенерировать очередную перестановку, необходимо обменять местами текущий включенный элемент в перестановку со всеми последующими элементами массива (цикл перебора элементов для обмена начинается с индекса t, чтобы не терять первоначальные перестановки после добавления нового элемента). Сложность алгоритма оценивается как *O(n!)*. Используется дополнительная память $O(n)$.

```
void generate (int t){
     if (t==n-1){ //Вывод очередной перестановки
          for (int i=0;i<n;++i)
               cout<<a[i]<< " ";
               cout<<endl;
     }
     else{
          for (int j=t;j<n;++j){   //Запускаем процесс обмена
```

```
                swap(a[t],a[j]);     //a[t] со всеми последующими
                t++;
                generate(t);         //Рекурсивный вызов
                t--;
                swap(a[t],a[j]);
            }
        }
}
```

В основной программе первый рекурсивный вызов делаем от нуля (добавляем нулевой элемент).

```
generate(0);
```

Программа выведет всевозможные n! перестановок. Пример вывода для n=3.

1 2 3, 1 3 2, 2 1 3, 2 3 1, 3 2 1, 3 1 2.

Заметим, что программа генерирует перестановки не в лексикографическом порядке. Для различного рода задач необходим именно лексикографический порядок генерации. Две перестановки находятся в лексикографическом порядке, если первые их элементы могут совпадать, а начиная с некоторого индекса, элемент во второй перестановке больше, чем элемент в первой перестановке. Например, перестановки 12453 и 12534 находятся в лексикографическом порядке.

Перестановки. Генерация всех перестановок. Нерекурсивный алгоритм

Рассмотрим нерекурсивный алгоритм генерации перестановок в лексикографическом порядке.

1. Последовательность элементов просматривается с конца до тех пор, пока не будет встречен первый элемент, такой что *a[i]<a[i+1]*.
2. В «хвосте» последовательности, состоящем из элементов, расположенных за найденным элементом, производим поиск минимального элемента min, большего, чем a[i].
3. Меняем местами a[i] и найденный элемент min.
4. Сортируем хвост последовательности.

Такой алгоритм позволяет получить все перестановки в лексикографическом порядке. Проиллюстрируем на примере *n=5* рассмотренный алгоритм.

i=0	i=1	i=2	i=3	i=4	Комментарии
1	2	3	4	5	Определили, что a[3]<a[4]. Меняем его места с минимальным, большим a[3] в «хвосте» последовательности. То есть, поменяли 4 и 5
1	2	3	5	4	a[2]<a[3]. Min=4. Меняем 3 и 4. Сортируем «хвост» последовательности, то есть элементы 3 и 5.
1	2	4	3	5	
1	2	4	5	3	Найден первый элемент с конца, меньший последующего – это a[2]=4. В «хвосте» последовательности {5, 3} ищем минимальный min>a[2]. Min=5. Меняем 4 и 5. Получаем новый «хвост» {4, 3}.Сортируем «хвост». Получаем перестановку {1, 2, 5, 3, 4}.
1	2	5	3	4	

Генерация очередной перестановки реализована в виде функции NextPermutation(). Функция возвращает true, если была сгенерирована очередная перестановка и false, если очередной перестановки в лексикографическом порядке сгенерировать невозможно (конец работы программы).

```
bool NextPermutation(){
     for (int i = n - 2; i >= 0; i--){
               if (a[i] < a[i + 1]){
               int min_val = a[i + 1],  min_id = i + 1;
               for (int j = i + 2; j < n; j++)
                    if (a[j] > a[i] && a[j] < min_val){
                         min_val = a[j];
                         min_id = j;
                    }
               swap(a[i], a[min_id]);
               sort(a.begin() + i+1, a.end());
               return 1;
               }
     }
          return 0;
}
```

В основном тексте программы для генерации всех перестановок можно использовать цикл while(), работающий до тех пор, пока не удастся сгенерировать очередную перестановку.

```
while (NextPermutation()){
          for (int i=0;i<n;++i) cout<<a[i]<< " ";
          cout<<endl;
```

Определение перестановки по номеру и номера перестановки

Задача определения перестановки по ее номеру. Рассмотрим всевозможные перестановки из n первых натуральных чисел. Таких перестановок n!. Требуется по номеру перестановки вывести ее на экран. Договоримся, что нумерация перестановок начинается с 1 (единицы).

Например, при n=5 получаем 120 перестановок с номерами от 1 до 120. Допустим, нужно вывести на экран перестановку с номером num=110. Проведем следующие рассуждения. Все перестановки можно разбить на группы. Выделим n=5 групп перестановок по их первой цифре – от 1 до 5.

1 в начале. Перестановки 1****	4!=24 перестановки с 1 в начале. Номера 1-24
2 в начале. Перестановки 2****	24 перестановки с 2 в начале. Номера 25-48
3 в начале. Перестановки 3****	24 перестановки с 3 в начале. Номера 49-72
4 в начале. Перестановки 4****	24 перестановки с 4 в начале. Номера 73-96
5 в начале. Перестановки 5****	24 перестановки с 5 в начале. Номера 97-120

Таким образом, можно найти номер группы, к которой относится перестановка – 110/24=4. Значит, *первая цифра* в перестановке – dig=5. Номер перестановки в группе (а также и новое число для поиска второй цифры) Np=110%24=14. Перестановки вида 5**** в свою очередь разбивается на 4 группы с вторыми цифрами из множества {1,2,3,4}. Количество перестановок в каждой из четырех групп - (n-1)!=3!=6. Продолжаем алгоритм и далее, используя информацию о группе перестановки и номеру перестановки в группе, однозначно задающую текущую цифру перестановки. Заведем вспомогательный массив digit[], значения которого digit[i]=1, если цифра i уже была использована в перестановке. В перестановку будем брать dig-ю по счету свободную цифру.

i - номер цифры в искомой перестановке dig – номер свободной цифры, которую будем ставить на позицию i в перестановку Np – номер перестановки в группе	Шаблон P=***** до определения цифры на позиции i и после ее определения	Массив digit[], хранящий информацию об использовании цифр в перестановке

i=1;
dig=110/24+1=5;
Np=110%24=14
	P=***** (4!=24)
	P=5****
		{1,2,3,4,~~5~~} – множество цифр для второй *

i=2;
dig=14/6+1=3;
Np=14%6=2
	P=5**** (3!=6)
	P=53***
		{1,2,~~3~~,4,~~5~~} – множество цифр для третьей *

i=3; dig=2/2=1; Np=2%2=0; Np=6 (если остаток равен нулю, то единицу к dig не добавляем, номер перестановки в группе делаем последним)	P=53*** (2!=2) P=531**	{~~1~~,2,~~3~~,4,~~5~~} – множество цифр для четвертой *
i=4; dig=6/1=6; Np=6%1=0; Np=6	P=531** (1!=1) P=5314*	{~~1~~, 2,~~3,4,5~~} – множество цифр для пятой *
На пятом шаге алгоритма получим P=53142		

В программе удобно завести вспомогательный массив факториалов, например

```
int fact[13] = {1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880,
3628800, 39916800, 479001600};
```

Сама программа выглядит следующим образом.

```
for (int i=n; i>0;--i){
     Np=num%fact[i-1]; //Определение номера группы перест.
     d=num/fact[i-1]; //Определение индекса текущей цифры
     if (Np) d++; else Np+=fact[i-1];
     num=Np;
     int pos=0;
     for (int j=1;j<=n;++j){//Определение текущей цифры
          if (!digit[j]) pos++;
          if (pos==d) {
               digit[j]=1;
               res.push_back(j); //Формирование перестановки
               break;
          }
     }
```

Ответ программы – перестановка по ее номеру, хранится в массиве res. Сложность алгоритма - $O(n^2)$.

Задача определения номера перестановки по заданной перестановке. Требуется по заданной перестановке P вывести на экран ее номер. Нумерация перестановок начинается с 1 (единицы). Количество цифр в перестановке – n.

Рассмотрим идею решения на конкретном примере. n=5, p=53142. Номер данной перестановки сформируем следующим образом

p=53142. Первый элемент перестановки, данной в условии – 5. Перестановки, начинающиеся на цифры {1, 2, 3, 4} находятся перед данной нам в условии. Их количество 4*4!=96. Значит, номер перестановки, заданной в условии > 96. Следующий элемент перестановки – 3. Значит, до этого идут перестановки, у которых вторая цифра может быть 1 или 2. Следовательно, минимально возможный номер перестановки, заданной в условии > 4*4!+2*3!. Следующий элемент перестановки – 1 (минимально возможный). То есть, номер перестановки не меняется – минимальный номер заданной в условии перестановки остается прежним > 4!*4+3!*2+2!*0. Следующий элемент перестановки – 4. Напомним, что это на этом месте могли быть не занятые цифры– {2,4}. В перестановке используется 4 – второй не занятый элемент. Следовательно, минимально возможный номер перестановки, заданной в условии > 4!*4+3!*2+2!*0+1!*1. С учетом последнего элемента получаем - 4!*4+3!*2+2!*0+1!*1+0!*1=110 – номер перестановки.

Цифра i	i=2	i=3	i=4	i=5	Итог
Количество перестановок до i цифры	4!*4	4!*4+3!*2	4!*4+3!*2+2!*0	4!*4+3!*2+2!*0++1!*1	4!*4+3!*2+2!*0++1!*1+0!*1=110

Группа перестановки (с учетом занятых цифр)	5-я	3-я	1-я	4-я	2-я
Занятые цифры после вычисления	12345	12345	12345	12345	12345

Итак, получаем ответ – 110. Это номер перестановки p=53142. Сложность ма - $O(n^2)$.

Алгоритмы STL для генерации перестановок

Удобным преимуществом библиотеки STL является наличие в ней реализованных алгоритмов, в том числе и относящихся к комбинаторному перебору. Для использования алгоритмов требуется наличие заголовка <algorithm>. Приведем некоторые из них.

Алгоритм	Описание. Пример (msdn.microsoft.com)

next_permutation(it1, it2)	Генерирует следующую перестановку в указанном подмножестве контейнера, которое задается итераторами it1, it2.
	`Вектор v1 (-3 -2 -1 0 1 2 3).`
	`После первого next_permutation, вектор v1 выглядит следующим образом: v1 = (-3 -2 -1 0 1 3 2).`
	Функция next_permutation() возвращает true, если следующая лексикографическая перестановка сгенерирована, в противном случае – false (следующей лексикографической перестановки не существует).

prev_permutation(it1, it2)	Генерирует предыдущую перестановку в указанном подмножестве контейнера, которое задается итераторами it1, it2.
	`Вектор v1 = (-3 -2 0 3 -1 2 1).`
	`После prev_permutation вектора v1, вектор v1 выглядит следующим образом: v1 = (-3 -2 0 3 -1 1 2).`

is_permutation(it11, it12, it21, it22)	Функция возвращает true, когда указанные подмножества можно переупорядочить так, чтобы они стали совпадающими. В противном случае функция возвращает false.

В нерекурсивном алгоритме генерации всех m-размещений из n элементов потребуется использование алгоритма next_permutation().

Генерация всех m-размещений из n элементов. Рекурсивный и нерекурсивный алгоритмы

Рекурсивный алгоритм. В массиве a[] будет хранится очередное размещение. Вспомогательный массив used[] используется для пометки элемента, как уже взятого в размещение. Суть алгоритма в том, что в очередное размещение добавляются по очереди всевозможные элементы, которые еще не были использованы в размещении. Пример реализации функции generate, получающий на вход номер очередного элемента, включаемого в размещение.

```
void generate(int num){
     if (num == m){ //Если размещение готово, то печатаем его
          for (int i = 0; i < m; i++)
               cout<<a[i]<<" ";
               cout<<endl;
          }
     for (int i = 1; i <= n; i++)
          if (!used[i]){ //Добавляем еще не взятый элемент
               used[i] = 1, a.push_back(i);
               generate(num + 1);
               used[i] = 0, a.pop_back();
          }
}
```

Вызов функции из основного текста

```
     generate(0); //Начинаем добавлять элемент на нулевое место
```

Программа выведет для n=3, m=2 следующие перестановки: 1 2, 1 3, 2 1, 2 3, 3 1, 3 2.

Нерекурсивный алгоритм генерации следующего размещения. Работа алгоритма основыва-

ется на генерации всевозможных перестановок первых натуральных чисел от 1 до n и умении «взять» новое размещение из перестановки, «удалив» лишние элементы из «хвоста» перестановки. Рассмотрим пример для n=4 всех перестановок и всех размещений при m=3.

Перестановки. n=4	Размещения. n=4, m=3
1234	123
1243	124
1324	132
1342	134
1423	142
1432	143

По таблице хорошо видно, что размещения можно получать из перестановок, «удаляя» хвост новой перестановки и оставив только первые m элементов в ней. Новое размещение получается из той перестановки, в которой будет меняться элемент, стоящий на m –месте. Это происходит тогда, когда за m-м элементом следуют элементы в порядке убывания. Получаем способ генерации нового размещения: добавим к очередному размещению неиспользованные элементы в порядке убывания, сгенерируем новую перестановку, возьмем первые m элементов и получим новое размещение. Приведем фрагмент программы генерации размещений.

```
bool generate(){
     int k=m;
     for (int i=n-1;i>=0;--i){
          if(!used[i]){
               a[k]=i+1;
               k++;
          }
     }
     if (next_permutation(a.begin(),a.end())){
          for (int i=m;i<n;++i) used[a[i]-1]=0;
          for (int i=0;i<m;++i) used[a[i]-1]=1;
          return 1;
     }
     else
          return 0;
}
```

Начальные присваивания в основном тексте программы

```
for (int i=0;i<n;++i)a[i]=i+1; //Берем последовательные эл.
     for (int i=0;i<m;++i)used[i]=1; //Помечаем первые m
     while (generate()){//Если функция генерирует перестановку
          for (int j=0;j<m;++j)cout<<a[j]; //печатаем ее
               cout<<endl;
     }
```

Сложность алгоритма генерации всех размещений оценивается как O(n!/(n-k)!).

Генерация всех m-сочетаний n элементов. Рекурсивный и нерекурсивный алгоритмы

Рекурсивная функция генерации сочетаний. Сочетания из n=4 элементов по m=3 будут представлять собой следующий набор подмножеств n-элементного множества: 123, 124, 134, 234. Реализация в виде рекурсивной функции generate() использует переменные: last – последний добавляемый элемент в сочетание и num – количество элементов в сочетании. Если сочетание сформировано (num=m), то выводим его на экран. В противном случае добавляем новый элемент в сочетание, следующий за последним добавленным элементом, и снова вызываем рекурсивно функцию generate().

```
            for (int i = 0; i < m; i++)
                cout<<a[i];
                cout<<endl;
        }
        for (int i = last + 1; i <= n; i++){
                a.push_back(i);
                generate(num + 1, i);
                a.pop_back();
        }
}
```

Основной текст программы вызывает функцию с начальными параметрами 0,0.

```
generate(0 , 0);
```

На экране будут выведены для n=5, m=3 следующие сочетания: 123, 124, 125, 134, 135, 145, 234, 235, 245, 345.

Нерекурсивный алгоритм генерации сочетаний. В основе алгоритма используется следующая идея: просматривая сочетания справа налево, находим первый элемент, который можно увеличить. Этот элемент увеличиваем (берем следующий возможный элемент), а все элементы после него, заменяем на натуральные числа, следующие за новым элементом. Зададимся вопросом – какой элемент можно увеличить в сочетании?. Если n=5, m=3, то максимальное лексикографическое сочетание 345. Значит, на месте m может стоять максимальный элемент n, на месте (m-1) может стоять максимальный элемент (n-1). То есть, на месте i может стоять максимальный элемент (n - m + i). Нумерация элементов в массиве a[] - с единицы.

```
bool Next(){
        for (int i = m; i >= 1; i--)
            if (a[i] < n - m + i)    {
            a[i]++;
            for (int j = i + 1; j <= n; j++)
                    a[j] = a[j - 1] + 1;
                return 1;
                                }
        return 0;
}
```

Сложность алгоритма генерации всех сочетаний оценивается как O(n!/(k!*(n-k)!)).

Представление числа в виде суммы целых положительных слагаемых

Необходимо перечислить все представления целого положительного числа n в виде суммы последовательности невозрастающих целых положительных слагаемых.

Для n=5 программа выведет на экран следующие представления числа пять в виде сумммы.
1+1+1+1+1=5, 2+1+1+1=5, 2+2+1=5, 3+1+1=5, 3+2=5, 4+1=5.

Программа решения задачи будет оперировать массивом слагаемых a[], состоящим из n элементов, где n – максимальное количество слагаемых. Максимальное количество слагаемых достигается в том случае, когда все слагаемые – единицы. Пусть a[1], a[2], a[3],..,a[last] – невозрастающая последовательность целых чисел, сумма которых не превосходит числа n. Сумма элементов последовательности хранится в переменной s= a[1]+ a[2]+ a[3]+… +a[last], где last – индекс последнего взятого элемента. Если сумма стала равна заданному числу n, то выводим последовательность чисел. В противном случае, пробуем подобрать следующее слагаемое j, начиная с единицы, которое не больше последнего слагаемого j<= a[last], и которое не увеличит сумму, так чтобы она не стала больше n, т.е. j<= (n-sum). Взяв очередное слагаемое, комбинируем последовательность дальше. Функция представления числа в виде суммы приведена ниже.

```
void generate(int last,int sum){
     if (sum==n) {
          for (int i=0;i<last;++i)
               cout<<a[i]<<"+";
          cout<<a[last]<<"="<<n<<endl;
     }
     else
          for (int j=1; j<=min(a[last],n-sum);++j){
//Берем следующие слагаемые, начиная с 1, но не больше a[last]
               last++;
               a[last]=j;
               generate(last,sum+j);
               last--;
          }
}
```

В основном тексте программы необходимо вызвать функцию для различных первых слагаемых из представления числа в виде суммы:

```
for (int j=1; j<n;++j){
     a[0]=j;
     generate(0,j); //Индекс последнего элемента, сумма
}
```

Аналогично можно составить алгоритм представления числа в виде суммы неубывающих слагаемых.

Правильные скобочные последовательности. Числа Каталана

Проверка скобочной последовательности с целью определения ее правильности была проделана нами в лекции 1 при изучении темы «Стек». В этом разделе мы хотим решить следующую задачу: *сгенерировать все правильные скобочные последовательности, состоящие из n пар скобок*. Напомним, что скобочная последовательность правильная, если для каждой позиции скобочной последовательности выполняется правило - число открывающихся скобок в последовательности больше или равно числу закрывающихся скобок, а общее число открывающихся скобок для всей скобочной последовательности равно числу закрывающихся.

Нерекурсивный алгоритм генерации правильных скобочных последовательностей

Пусть n=3 пар скобок. Рассмотрим все соответствующие правильные скобочные последовательности из 6 скобок. ()()(), ()(()), (())(), ((())), (()()).

Генерацию очередной скобочной последовательности будем делать следующим образом:

- просматриваем последовательность справа налево до первой комбинации скобок)(. Такая комбинация будет в любой последовательности, кроме последней;
- найденную комбинацию надо заменить на ();
- подсчитываем с начала строки, на сколько число открывающихся скобок больше числа закрывающихся и дописываем это количество закрывающихся скобок;
- если длина строки еще не стала равна 2*n, то ее необходимо дополнить нужным количеством пар скобок ().

Рекурсивный алгоритм генерации правильных скобочных последовательностей

Лексикографическим порядком правильных скобочных последовательностей будем называть следующий их порядок: ((())), (()()), (())(), ()(()), ()()() (приведен пример для n=3 – трех пар скобок). Рекурсивная функция получает следующие параметры: n – количество пар скобок, s – строка, представляющая собой скобочную последовательность, op_br – количество открывающихся скобок в текущей последовательности, cl_br – количество закрывающихся скобок в текущей последовательности. Функция печатает очередную скобочную последовательность, если уже использовано n пар скобок. Если можно поставить открывающуюся скобку (не все открывающиеся скобки использованы), то ставим открывающуюся скобку. Закрывающуюся скобку ставим в том случае, если ее можно поставить, то есть количество отрывающихся скобок больше количества закрывающихся.

```
void generate(int n, string s, int op_br, int cl_br){
     if (op_br+cl_br==2*n)
          cout<<s<<endl;
     if (op_br<n)
          generate(n, s+'(',op_br+1,cl_br);
     if(op_br-cl_br>0){generate(n,s+')',op_br,cl_br+1);

}
```

Количество правильных скобочных последовательностей. Числа Каталана

Количество правильных скобочных последовательностей определяется числом Каталана C_n, применяющегося во многих комбинаторных задачах. Для вывода формулы будем рассматривать X – произвольную правильную скобочную последовательность длины 2n. Она начинается с открывающейся скобки, ей соответствует определенная закрывающаяся скобка. Найдем ее и представим исходную последовательность как X=(A)B, где A и B – тоже правильные скобочные последовательности. Пусть длина последовательности A равна 2k, тогда правильных скобочных последовательностей для A - C_k. Тогда длина последовательности B – (2n-2k-2)=2(n-k-1), а правильных скобочных последовательностей для B - C_{n-k-1}. Всевозможные комбинации правильных скобочных последовательностей для k=0..n-1 и дадут общее количество правильных последовательностей. То есть, получаем рекуррентную формулу:

$C_n = C_0C_{n-1} + C_1C_{n-2} + C_2C_{n-3} + \cdots + C_{n-1}C_0$.

Начальные значения чисел Каталана: $C_0 = 1$, $C_1 = 1$ (ноль скобок и одну пару скобок можно расставить одним способом), $C_2 = 2$. Последующие значения приведены в таблице.

n	0	1	2	3	4	5	6	7	8	9
C_n	1	1	2	5	14	42	132	429	1430	4862

Числа Каталана применяются и в других задачах комбинаторики, с их помощью можно вычислить:

- количество бинарных деревьев с заданным количеством листьев,
- количество различных триангуляций выпуклого многоугольника диагоналями,
- количество разбиений на пары четного числа точек на окружности непересекающимися хордами.

Лекция 7. Графы: способы их хранения и обхода (в ширину и в глубину). Проверка графа на двудольность, поиск циклов и топологическая сортировка графа

Понятие графа. Специальные графы. Способы хранения графов в виде матрицы смежности и списка смежности. Способы обхода графа: обход в ширину и обход в глубину. Компоненты связности графа. Ориентированные графы. Поиск циклов в ориентированном и неориентированном графе. Проверка графа на двудольность. Топологическая сортировка графа. Расстояния в графе.

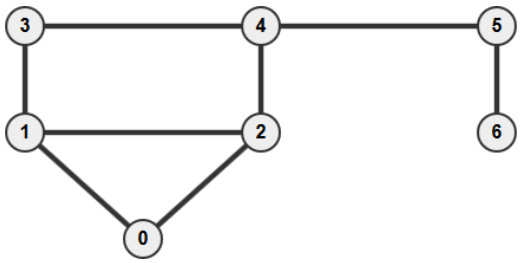
Общее представление

Основные понятие теории графов занимают важное место и в математических олимпиадах, и в олимпиадах по программированию. Почти в каждой олимпиаде по программированию будет встречаться задача на тему «Графы» и для того, чтобы ее решить необходимо не только хорошо знать известные алгоритмы на графах, но и быть математически осведомленным в теории графов. В этой лекции будут освещены самые необходимые понятия и алгоритмы теории графов. Кроме того, графы занимают важное место в прикладных исследованиях по различным разделам знаний: геоинформатике, химии, экономике и теории управления, логистике, биоинформатике. Лишь уверенно освоив базовые алгоритмы, приведенные в этой лекции, можно будет приступать к дальнейшему изучению более сложных алгоритмов теории графов.

Основные определения графов. Способы хранения графов

Графом G называется совокупность множеств $G = (V(G), E(G))$, где $V(G)$ — непустое конечное множество элементов, называемых *вершинами графа*, а $E(G)$ — множество пар элементов из $V(G)$ (необязательно различных), называемых *ребрами графа*. $E(G) = \{(v_i, v_j)\}$ — множество ребер графа G, состоящее из пар вершин (v_i, v_j) . Ребро (v_i, v_j) соединяет вершины v_i и v_j. Две вершины, соединенные ребром, называют смежными вершинами. Количество ребер, исходящее из вершины называют степенью вершины *d(v)*. Если какие-то две вершины соединены более, чем одним ребром, то говорят, что граф содержит *кратные ребра*. Если ребро соединяет вершину саму с собой, то такое ребро называют *петлей*. *Простой граф* не содержит петель и кратных ребер. Граф называют *ориентированным*, если ребра графа имеют направление (в этом случае они называются *дугами*), в противном случае граф – *неориентированный*.

Матрица смежности A для данного графа G содержит элементы $a_{ij} = 1$, если две вершины v_i и v_j являются смежными и $a_{ij} = 0$, если вершины v_i и v_j смежными не являются. Матрица смежности простого графа симметрична. Пример графа G и матрица смежности A для данного графа приведены на рисунке.



Матрица смежностей

	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
2	1	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	1	0
5	0	0	0	0	1	0	1
6	0	0	0	0	0	1	0

V={0, 1, 2, 3, 4, 5, 6} – множество вершин графа. |V|=7 – количество вершин.

E={ (0,1),(0,2),(1,2),(1,3),(2,4),(3,4),(4,5),(5,6)}. |E|=8 – количество ребер

Хранение матрицы смежности требует $O(|V|^2)$ памяти, что во многих случаях неэкономно. В ряде алгоритмов удобнее пользоваться не матрицей смежностей графа, а *списком смежности*. В списке смежности для каждой вершины указываются вершины, смежные с ней. В следующей таблице представлен список смежности вершин для графа G. Хранение списка смежности требует $O(|V| + |E|)$ памяти.

Если входные данные к задаче задают матрицу смежности графа, то по ней легко получить список смежности в виде двумерного массива g, где g[i] – вектор вершин, смежных с вершиной *i*.

Вершина	Смежные вершины		Преобразование матрицы смежности в список смежности

0	1, 2	d(0)=2	`vector<vector<int>> g(n);`
			`for (int i=0; i<n;++i)`
			`for (int j=0;j<n;++j){`
			`cin>>a;`
			`if (a) g[i].push back(j);`
			`}`

1	0, 2, 3	d(1)=3	
2	0, 1, 4	d(2)=3	
3	1, 4	d(3)=2	
4	2, 3, 5	d(4)=3	
5	4, 6	d(5)=2	
6	5	d(6)=1	

Если входные данные к задаче представляют собой список ребер, то по данному списку легко получаем и матрицу смежности, и список смежности. Ниже приведен пример для графа G входных данных в виде списка ребер: на вход программы поступают числа *n* – количество вершин в графе и *m* (1<=*m*<=*n(n-1)*/2) – количество ребер. Затем следует *m* пар чисел – ребра графа.

```
7  8
0  1
0  2
1  2
1  3
2  4
3  4
4  5
5  6
```

Входные данные: n=7, m=8. 8 ребер заданы парой смежных вершин. Считываем список ребер и формируем массив списка смежности g.

```
cin>>n>>m;
vector<vector<int>> g(n);
for (int i=0; i<m;++i){
        cin>>v1>>v2;
        g[v1].push_back(v2);
        g[v2].push_back(v1);
}
```

Плотные графы, имеющие большое количество ребер оптимальнее хранить при помощи матрицы смежности, а вот *разреженные графы*, имеющие небольшое количество ребер, оптимальнее хранить при помощи списка смежности.

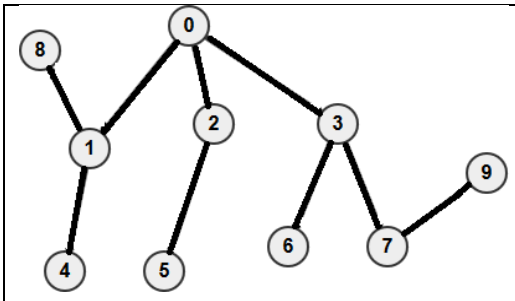
Если граф является ориентированным, то матрица смежности в общем случае не будет являться симметричной. Для *взвешенных графов*, то есть графов, каждому ребру которых сопоставлено число, например, расстояние от одной вершины до другой, соответствует матрица весов. Матрица весов является обобщением матриц смежности, и содержит в качестве элементов веса ребер графа.

Обход графа в глубину

Алгоритм обхода графа глубину (depth-first search)

Обход графа в глубину DFS позволяет определить вершины, достижимые из данной вершины. При обходе графа используется массив used[], хранящий информацию о том, была ли посещена вершина: $used[i] = 1$, если вершина уже была посещена при обходе и $used[i] = 0$, если вершина еще не была посещена. В начале работы алгоритма все вершины являются непосещенными: $used[i] = 0, \forall i = 1..n$. Функция DFS получает на вход очередную вершину и вызывает себя рекурсивно для всех еще непосещенных вершин, достижимых из данной. Из основного текста программы вызов dfs осуществляется от стартовой вершины – той, с которой мы начинаем обход.

Неориентированный граф G	Функция обхода графа в глубину DFS




```
void dfs (int v){
  used[v]=1;
  for (auto i=g[v].begin();i!=g[v].end();++i)
          if(!used[*i])
                dfs(*i);
}
```

Вызов из основного текста программы

```
dfs (0)
```

Программа обойдет вершины графа в следующем порядке: 0, 1, 4, 8, 2, 5, 3, 6, 7, 9.

Оценка сложности алгоритма включает в себя следующие операции:

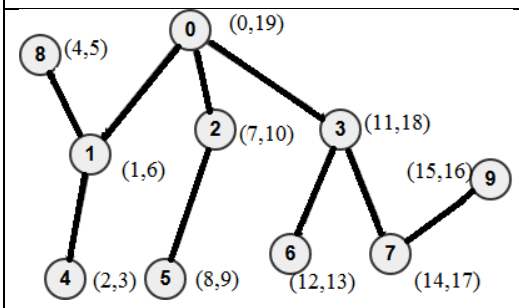
- просмотр всех |V| вершин, для каждой из которых *v* просматриваются ее соседи;
- просмотр всех соседей вершины *v*. При этом алгоритм проходит по ребру {v, u}. Причем, каждое такое ребро {v, u} просматривается дважды: для вершины *u* и для вершины *v*.

Итоговая сложность алгоритма dfs, таким образом $O(|V| + |E|)$.

Время начала и конца обработки вершины

Для ясного понимания алгоритма поиска в глубину рассмотрим время начала обработки и время конца обработки вершин. В счетчике time будет фиксироваться текущее время, каждый раз увеличиваясь на единицу. В вектора time_in[] будет записываться время начала обработки для вершин, а в вектора time_out[] – время конца обработки. После работы алгоритма dfs получим времена начала и конца обработки вершин графа G, указанные в виде пар чисел на графе.

Неориентированный граф G



Функция обхода графа в глубину DFS с фиксацией времени начала и конца обработки вершин

```
void dfs (int v){
  used[v]=1;
  time_in[v]=time++;
  for (auto i=g[v].begin();i!=g[v].end();++i)
     if(!used[*i])
          dfs(*i);
  time_out[v]=time++;
}
```

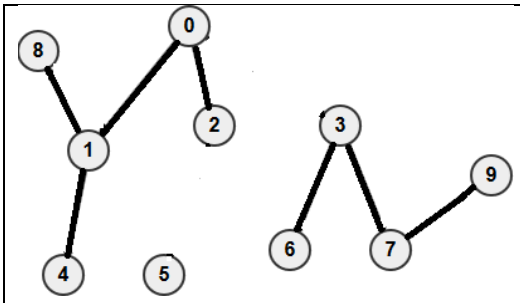
Заметим, что [time_in[v], time_out[v]] – это промежуток времени, в течение которого вершина v была на обработке. Аналогично, [time_in[u], time_out[u]] – это промежуток времени, в течение которого вершина u была на обработке. Указанные отрезки либо не пересекаются, либо содержится один в другом, так как если вершина u, например, начала обрабатываться после вершины v, то и завершение ее обработки будет раньше, чем у вершины v.

При помощи значений векторов time_in[] и time_out[] можно проверить, является ли одна вершина дерева предком другой. Ответ можно найти за O(1): вершина *i* является предком вершины *j* тогда и только тогда, когда time_in[i]<time_in[j] и time_out[i]>time_out[j].

Поиск компонент связанности графа

Путем в графе называется последовательность вершин $v_i \in V, i = 1..k$ таких, что две любые последовательные вершины соединены хотя бы одним ребром, и все ребра различны. Число k вершин в пути называется *длиной пути*. Граф G называется *связным*, если две его любые вершины соединены путем. Если граф не связен, то его можно разбить на непересекающиеся связные подмножества, называющиеся *компонентами связанности*. На рисунке представлен несвязный граф, имеющий три компоненты связности. Поиск глубину dfs будет обходить ту компоненту связности, из вершины которой, он был вызван. Поэтому при помощи dfs легко проверить, является ли граф связным, вызывая во внешнем цикле dfs от всех, еще непосещенных вершин. Можно также найти все компоненты связности и для каждой вершины вывести номер той компоненты связности, ко- торой она принадлежит – номера компонент связности вершин будут храниться в векторе cc_num[].

Несвязный граф G	Вызов DFS из основного текста программы



{0, 1, 2, 4, 8}, {5}, {3, 6, 7, 9} - компоненты связности.

```
for (int i=0;i<n;++i){
     if (!used[i]){
          cc++;
          dfs(i);
     }
}
```

В саму функцию DFS необходимо добавить заполнение вектора номеров компонент связности для вершина графа - `cc_num[v]=cc;`

Обход ориентированного графа в глубину

Рассмотрим особенности обхода графа в глубину для ориентированных графов. Входные данные для ориентированного графа в виде перечня ребер *u v*, при этом подразумевают, что направление ребра *(u,v)* - от вершины u к вершине v. При считывании данных добавляем только вершину v к списку смежности вершины u.

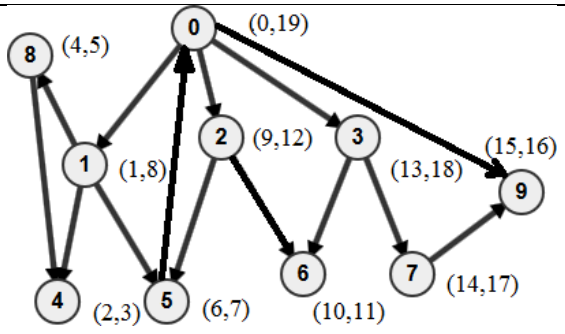
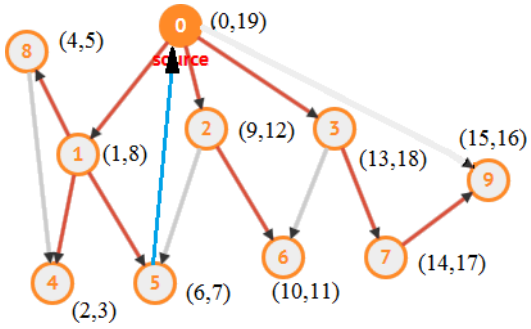
Алгоритм DFS как на неориентированном графе, так и на ориентированном графе построит дерево. *Дерево* – это связный граф без циклов. Корень дерева будет находиться в стартовой вершине (той, с которой начался обход в глубину). Ориентированный граф и соответствующее ему дерево приведены в таблице ниже. На графе справа можно выделить несколько типов ребер.

Древесные ребра. Красным цветом в дереве справа помечены те ребра *(u,v),* которые могут быть изображены в соответствии с временем начала и конца обработки вершин. Это так называемые древесные ребра. Для их вершин выполняется: time_in[u]<time_in[v]<time_out[v]<time_out[u].

Прямые ребра. Ведут от вершины к потомку, не являющемуся ребенком. Для них выполняется то же условие, что и для древесных ребер *(u,v):* На рисунке справа ребро (0, 9) – прямое.

Обратные ребра. Ведут от вершины к ее предку. То есть, для обратного ребра *(u,v):* выполняется условие time_in[v]<time_in[u]<time_out[u]<time_out[v]. Синее ребро на рисунке – обратное.

Перекрестные ребра. Для таких ребер выполняется условие time_in[v]<time_out[v]< time_in[u]<time_out[u]. На рисунке справа можно найти перекрестные ребра, например (8,4).

Ориентированный граф G	Дерево, построенное алгоритмом DFS по ориентированному графу G
	

В случае *неориентированного* графа алгоритм в глубину также будет строить дерево на графе, а все ребра после алгоритма, примененного к неориентированному графу, будут двух типов: древесные и обратные.

Поиск циклов в графе

Циклом в графе G называется путь, ведущий из вершины *v* в саму себя. Граф называют *ациклличным*, если в нем нет циклов. Рассмотрим граф G без петель и кратных ребер, и проверим его на ацикличность. Функция dfs обнаружит цикл в том случае, если пытается пройти в вершину, которая находится в данный момент времени в обработке. Вершины, находящиеся в обработке, алгоритм будет красить в серый цвет, вершины, не находящиеся в обработки предварительно покрашены в белый цвет, а уже обработанные вершины алгоритм будет красить в черный цвет. Для вывода всех вершин цикла удобно использовать массив предков.

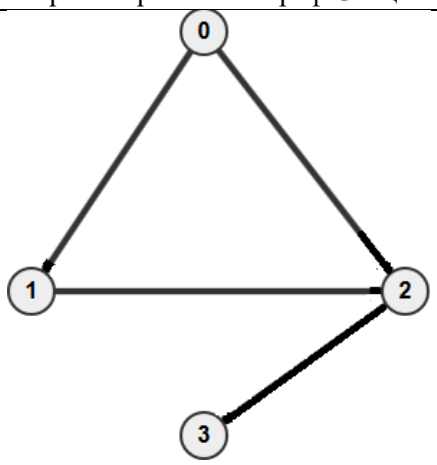
Поиск циклов в неориентированном графе

Для неориентированных графов нужно предусмотреть ситуации, когда dfs из вершины идет в ее предка – такие ситуации не являются признаком цикла. Для хранения массива предков будем использовать массив p[].

Неориентированный граф G. Поиск цикла

```
void dfs (int v) {
  used[v] = 1;
  for (size_t i=0; i<g[v].size(); ++i) {
     int to = g[v][i];
     if (used[to] == 0){
          p[to] = v;
          dfs (to);
     }
     else if (used[to] == 1&&to!=p[v])
          cycle=true;
}
  used[v] = 2;
}
```

Неориентированный граф G с циклом



Цикл 0→1→2

Алгоритм посетит вершину 0: покрасит ее в серый цвет, запишет предком вершины 1 вершину 0;
посетит вершину 1: покрасит ее в серый цвет, запишет предком вершины 2 вершину 1;
попытается пройти в вершину 0, но она уже серая и при этом не была еще отмечена как предок вершины 2 – значит, алгоритм обнаруживает цикл. Переменная cycle становится равной true.

Для вывода вершин цикла необходимо в момент обнаружения цикла (попытка перехода в вершину, находящуюся в обработке) не только изменить значение переменной cycle=true, но и запомнить начало и конец цикла.

```
cycle=true;
cycle_end = v; //Текущая вершина - конец цикла
cycle_st = to; //Вершина, в которую пытаемся перейти  - начало цикла
```

Вывод вершин цикла осуществляется за один проход по массиву предков, начиная с конца цикла.

```
     while (k!=cycle_st){
          cout<<k<<" ";
          k=p[k];
     }
     cout<<cycle_st;
```

Сложность алгоритма оценивается как O(|E|) – достаточно один раз пройти по каждому ребру графа, чтобы обнаружить цикл и закончить алгоритм.

Поиск циклов в ориентированном графе

Для ориентированных графов алгоритм аналогичен, за исключением того, что как только алгоритм пытается посетить вершину, находящуюся в обработке, сразу можно делать вывод о наличии цикла (дополнительных условий проверять не нужно). В основном тексте программы необходимо осуществить серию запусков DFS от каждой непосещенной еще вершины, с целью поиска циклов.

Ориентированный граф G. Поиск цикла	Основной текст программы

`void dfs (int v) {`
`  used[v] = 1;`
`  for (size_t i=0; i<g[v].size(); ++i) {`
`  int to = g[v][i];`
`  if (used[to] == 0) {`
`  p[to] = v;`
`  dfs (to);`
`  }`
`  else if (used[to] == 1)`
`  cycle=true;`
`}`
`  used[v] = 2;`
`}`
	`for (int l=0; l<n;++l){`
	`  if (!used[l]) dfs(l);`
	`  if (cycle) {`
	`  cout<<cycle;`
	`  return 0;`
	`  }`
	`}`
	`cout<<0;`
	Серия запусков DFS в основном тексте программы.

Восстановление вершин, образующих цикл делается, как и в случае неориентированного графа по массиву предков. Сложность алгоритма - O(|E|)

Основные свойства графов. Специальные графы

Для решения олимпиадных задач по теме «Графы» необходимо знать их основные свойства и специальные виды графов.

В неориентированном графе G сумма степеней всех вершин равна удвоенному количеству всех ребер. $\sum_{v_i \in V} d(v_i) = 2|E|$. Доказательство следует из того факта, что каждое ребро учитывается в степенях двух вершин, являющихся его концами.

Количество вершин в графе G, имеющих нечетную степень, четно. Доказательство следует из необходимости четности сумм степеней всех вершин.

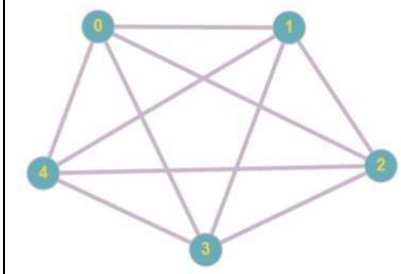
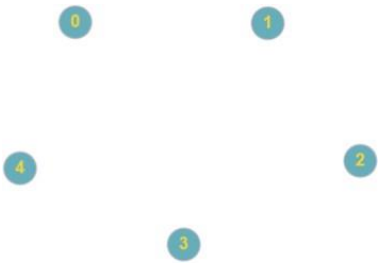
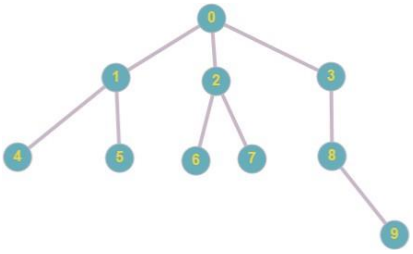
На n вершинах можно построить $2^{C_n^2}$ различных графов. Например, на 3 вершинах можно построить 8 различных графов, включай пустой граф.

Рассмотрим некоторые специальные виды графов.

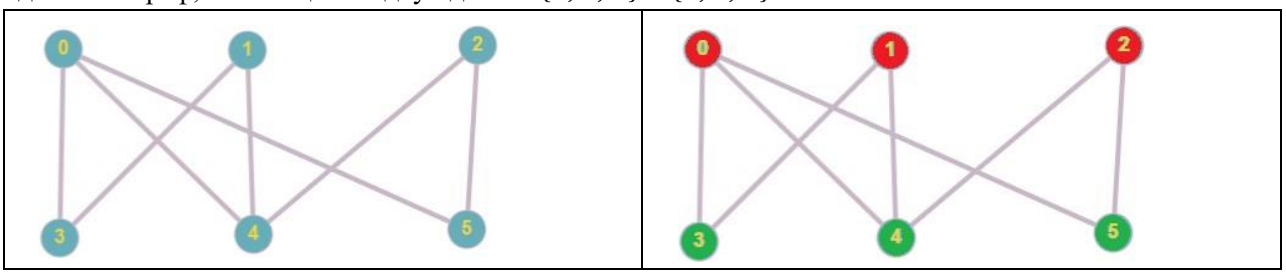
Полный граф K_n – граф, в котором любая его вершина соединена с каждой другой вершиной. У полного графа K_n будет $n(n-1)/2$ ребер.

Пустой граф $\bar{K}_n$ состоит из n изолированных вершин, и является дополнением к полному графу. *Дополнением* $\bar{G}$ к графу G называется граф, у которого множество вершин совпадает с первым графом, а множество ребер графа $\bar{G}$ дополняет множество ребер графа G до полного графа K_n.

Дерево – это простой связный граф без циклов. В дереве, построенном на n вершинах, имеется ровно (n-1) ребро. Можно доказать теорему, полезную для определения того, является ли граф деревом: *Любой простой связный граф, построенный на n вершинах и имеющий (n-1) ребро – дерево*. Примеры графов рассмотренных видов приведены на рисунках.

Полный граф K_n, n=5	Пустой граф $\bar{K}_n$, n=5	Дерево
		

Двудольный граф – граф, вершины которого можно разбить на два множества так, чтобы концы каждого ребра принадлежали различным множествам. На рисунке ниже представлен двудольный граф, состоящий из двух долей: {0, 1, 2} и {3, 4, 5}.



Обратим внимание на тот факт, что каждая доля двудольного графа может быть раскрашена в свой цвет, причем вершины одного цвета не являются смежными.

Полный двудольный граф $K_{m,n}$ – это граф со всеми возможными ребрами между долями.

Проверка графа на двудольность

Алгоритм DFS применяется для проверки графа на двудольность. При обходе графа в глубину делаем раскраску графа в два цвета: {1,2}. Если в процессе работы алгоритма встречаем ребро, концы которого окрашены в один и тот же цвет, то граф является двудольным. Для неориентированного графа требуется проверка, что мы идем не в ту вершину, которая уже помечена как предок рассматриваемой.

Проверка на двудольность неориентированного графа

```
void dfs(int v){
  for(size_t i=0;i<g[v].size();++i){
      int to=g[v][i];
      int try_c=3-used[v];
        if (!used[to]){
            used[to]=try_c;
            p[to]=v;
            dfs(to);
        }
        else if(to!=p[v]&&used[v]==used[to])
                  Two=false;
  }
}
```

Проверка на двудольность ориентированного графа

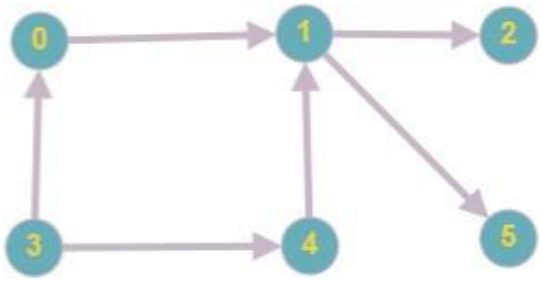
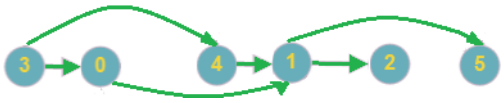
```
void dfs(int v){
  for(size_t i=0;i<g[v].size();++i){
      int to=g[v][i];
      int try_c=3-used[v];
        if (!used[to]){
            used[to]=try_c;
            p[to]=v;
            dfs(to);
        }
        else if (used[v]==used[to])
                  Two=false;
  }
}
```

В основном тексте программы цвет стартовой вершины делаем равным це - used[0]=1 и предполагаем, что граф является двудольным - Two=true. Делаем серию запусков dfs с проверкой на двудольность от каждой непосещенной вершины, начиная с нее как со старто-

вой. Вывод всех вершин каждой доли можно сделать согласно полученной раскраске вершин в цвета. Сложность алгоритма - O(|E|).

Топологическая сортировка графа

Дан ориентированный граф, не содержащий циклов (предварительно необходимо проверить отсутствие циклов). Топологическая сортировка (topologically sort) упорядочивает вершины графа так, что все ребра идут от вершины с меньшим номером к вершине с большим номером в топологическом порядке.

Ориентированный ациклический граф G	Топологический порядок вершин графа
	3 0 4 1 2 5 – один из возможных топологических порядков. 3 – исток графа, 5 – сток графа. 

На рисунке справа вершины графа G расположены в топологическом порядке. Произведена, так называемая, линеаризация графа. В вершину 3 не входит ни одно ребро – это исток графа. Из вершины 5 не выходит ни одно ребро – это сток графа. У каждого ациклического ориентированного графа есть хотя бы один исток и хотя бы один сток.

В ациклическом ориентированном графе нет обратных ребер, поэтому для топологической сортировки следует упорядочить вершины в порядке убывания их времени конца обработки вершины time_out[]. То есть, в процедуре DFS при выходе из вершины ее номер добавляется в конец вектора с ответом. Ответ выводится в противоположном порядке. Из основного текста программы выполняется серия запусков DFS от каждой непосещенной вершины.

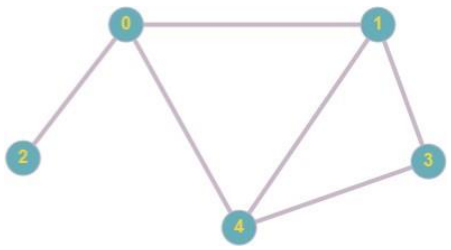
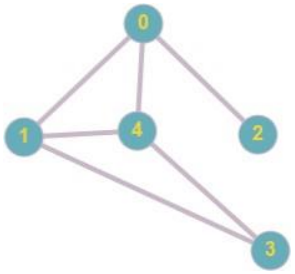
```
void dfs(int v)//Функция DFS
{
     for (size_t i=0; i< g[v].size(); ++i)
          if (used[g[v][i]]==0)
               dfs(g[v][i]);
     used[v]=2;         //По окончании обработки
     ans.push_back(v); //вершины добавляем ее в вектор ответа
}
int main() //Основной текст программы
{
     /*Считывание данных…*/
     for (int i=0;i<n;++i)
          if (!used[i])
               dfs(i); Запуск от каждой непосещенной вершины
     for (auto i=ans.rbegin();i!=ans.rend();++i)
          cout<<(*i)<<" "; //Вывод вершин ans в обратном порядке
     return 0;
}
```

Одна из распространенных задач на топологическую сортировку заключается в проверке неравенств на непротиворечивость. Есть n переменных, значения которых неизвестны. Известно лишь про некоторые пары переменных, что одна переменная меньше другой. Требуется проверить, не противоречивы ли эти неравенства, и если нет, выдать переменные в порядке их возрас-

тания (если решений несколько — выдать любое). Данная задача решается алгоритмом топологической сортировки графа.

Обход графа в ширину

Обход графа в ширину или поиск в ширину BFS (breadth-first search) является еще одним способом обхода графа. В результате поиска в ширину находится путь кратчайшей длины в невзвешенном графе (ребра которого не имеют весов), то есть путь, содержащий наименьшее число рёбер. Сложность работы алгоритма такая же, как и алгоритма обхода в глубину - $O(|V| + |E|)$.

Неориентированный граф G	Граф G, изображенный в виде слоев по увеличению расстояния от вершины 0
	 Расстояние от вершины 0 до вершин 1, 4, 2 равно 1. Расстояние от вершины 0 до вершины 3 равно 2.

Суть алгоритма заключается в том, чтобы просматривать сначала стартовую вершину 0, затем те вершины, которые удалены от нее на расстояние 1 и так далее слоями: затем просматриваем вершины, которые удалены на расстояние d, далее d+1. Для этого в алгоритме используется очередь Q, в которую сначала заносится стартовая вершина 0. Затем повторяем следующие итерации: пока очередь не пуста, достаем из ее головы очередную вершину и просматриваем всех соседей этой вершины, и если какие-то из них еще не помещены в очередь, то помещаем их в конец очереди. При таком обходе, когда очередь станет пуста, все вершины будут просмотрены, причем в порядке увеличения расстояния от стартовой вершины. Длины кратчайших путей считаются в процессе алгоритма при помощи массива расстояний d[] и массива предков вершин p[].

Приведем алгоритм BFS (источник – сайт http://e-maxx.ru/algo/bfs)

```
queue<int> q; // Очередь вершин
q.push (s); //Добавляем стартовую вершину
vector<bool> used (n); //Вектор признака посещенности вершин
vector<int> d (n), p (n); //Вектор расстояний и предков
used[s] = true; //Стартовую вершину считаем посещенной
p[s] = -1; //У стартовой вершины нет предка
while (!q.empty()) { //Пока очередь не пуста
     int v = q.front(); //Извлекаем из головы очереди вершину
     q.pop(); //удаляем извлеченную вершину
     for (size_t i=0; i<g[v].size(); ++i) { //Просмотр всех
          int to = g[v][i];                  //смежных вершин
          if (!used[to]) {         //Если вершина не посещена,
               used[to] = true; //посещаем ее
               q.push (to); //и добавляем к концу очереди
               d[to] = d[v] + 1;//Считаем расстояние до вершины
               p[to] = v; //Запоминаем предка
          }
     }
}
```

В основном тексте программы для вывода пути до какой-то вершины to, делаем это следующим образом: пути нет, если вершина осталась непосещенной.

```
if (!used[to])
     cout << "No path!";
```

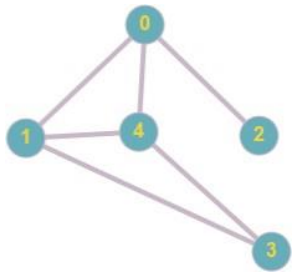
Если путь есть, то его можно восстановить в обратном порядке, используя массив предков и записывая все вершины в массив path[].

```
else {
     vector<int> path;
     for (int v=to; v!=-1; v=p[v])
          path.push_back (v);
```

Разумеется, путь будем выводить, начиная со стартовой вершины. Для этого выводим вектор path[] в обратном порядке.

```
     reverse (path.begin(), path.end());
     cout << "Path: ";
     for (size_t i=0; i<path.size(); ++i)
          cout << path[i] + 1 << " ";
}
```

Для понимания работы алгоритма рассмотрим состояние очереди Q при посещении каждой вершины.

Граф G, изображенный в виде слоев по увеличению расстояния от вершины 0		Порядок посещения вершин	Состояние очереди
	Расстояние от вершины 0 до вершин 1, 4, 2 равно 1. Расстояние от вершины 0 до вершины 3 равно 2.	0 1 4 2 3	0 1 4 2 4 2 3 2 3 3 Очередь пуста

Этот алгоритм можно применить также и к ориентированному графу. Расстоянием между вершинами в этом случае будет также минимальное количество ребер, которое надо пройти от стартовой вершины до данной, в направлении ориентации ребер. При этом расстояние не будет симметрично.

Алгоритм Дейкстры для определения кратчайших расстояний во взвешенном графе

Применяется для нахождения кратчайших путей от одной вершины (стартовой) до всех остальных вершин в неориентированном (или ориентированном) взвешенном графе, при условии, что все ребра в графе имеют неотрицательные веса. Алгоритм назван в честь голландского ученого Эдсгера Дейкстры и был предложен в 1959 году.

Сложность алгоритма оценивается как $O(|V|^2 + |E|)$

Обход в ширину применяется для определения расстояний между вершинами в том случае, когда длина каждого ребра одинакова (невзвешенные графы). На практике более распространена ситуация, когда каждое ребро имеет определенную длину, то есть ребра взвешены. Примером может служить карта местности с расстояниями между городами, такая, как например, на рисунке ниже. Требуется уметь определять наикратчайшее расстояние между любой парой вершин. Рассмотрим подробнее алгоритм Дейкстры, который применяется именно для решения таких задач.

Взвешенный граф удобно хранить в виде списка смежности, где для каждой вершины определен вектор пар – (смежная вершина, расстояние до этой вершины). vector<vector<pair<int,int>>> g – список смежности графа.

Карта Липецкой области

Таблица расстояний между некоторыми городами Липецкой области

Липецк – Елец	85
Липецк – Грязи	31
Липецк – Лебедянь	62
Липецк – Задонск	89
Липецк – Данков	85
Липецк – Усмань	75
Липецк – Хлевное	64

Описание алгоритма

Граф G.

Количество вершин – n.

Количество ребер – m.

s – стартовая вершина

d[] – массив текущих кратчайших расстояний из вершины s в вершины v_i.

d[s]=0, d[v]=∞ - начальные присваивания.

used[] – массив посещения вершин. Изначально все вершины не посещены.

В алгоритме n итераций

1. На очередной итерации выбирается вершина v_i с наименьшей величиной d[v_i] и еще не пройденная ранее. Выбранная вершина v_i отмечается посещенной.
2. Просматриваются все вершины, исходящие из вершины v_i, и для каждой такой вершины to алгоритм пытается улучшить значение d[to] по формуле d[to]=min(d[to],d[v_i]+len), где len – расстояние от вершины v до to
3. Пункты 1-3 алгоритма повторяются

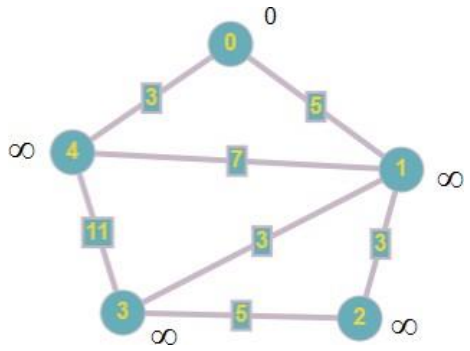
Реализация

```
const int inf=1e9;
int  main(){
  int n;
  cin>>n;
  vector<vector<pair<int,int>>> g(n);
  // чтение графа…
  vector <int> d (n, inf), p(n);
  vector<int> u(n);
  int s; - стартовая вершина
  d[s]=0;
  for (int i=0; i<n;++i) {
     int v=-1;
     for (int j=0; j<n;++j)
     if (!u[j]&&(v==-1||d[j]<d[v]))
          v=j;
          if (d[v]==inf) break;
          u[v]=true;
  for (size_t  j=0; j<g[v].size();++j){
     int to=g[v][j].first;
     int len =g[v][j].second;
     if (d[v]+len<d[to]){
          d[to]=d[v]+len;
          p[to]=v; }
     }
  }
}
```

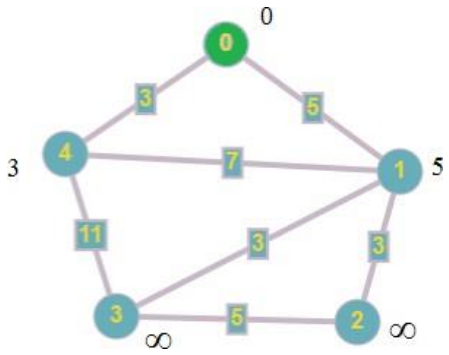
Вам предоставляется возможность проделать практическую работу по приведенному примеру и определить чему равно кратчайшее расстояние между Липецком и всеми другими городами. Разумеется, нужно узнать не только кратчайшее расстояние, но и соответствующие кратчайшим расстояниям пути от Липецка до других городов.

Ниже на рисунке приведена иллюстрация работы алгоритма Дейкстры на примере неориентированного графа. Значения элементов вектора расстояний d[] в виде меток вершин приведены на рисунке каждого шага алгоритма.

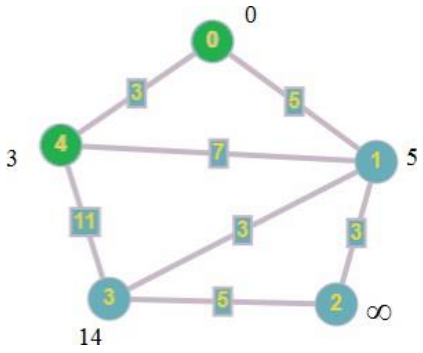
Исходный граф G.
Стартовая вершина – 0.
Начальные значения d[0]=0.
d[v_i]=∞.



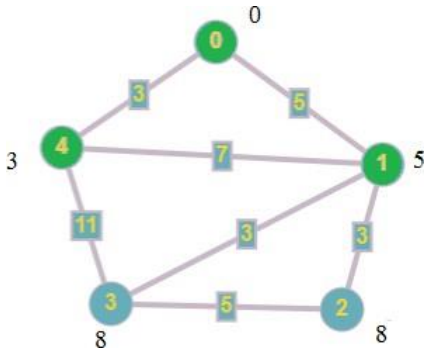
Итерация 1.
0 - вершина с наименьшим значением расстояния $0 = \{v_i | d[v_i] = \min\{d[v_j], j = 1..n\}$. Отмечаем ее посещенной u[0]=1. Для всех смежных с ней вершин пытаемся улучшить расстояние. Для вершин 3 и 5 алгоритм улучшил расстояние.

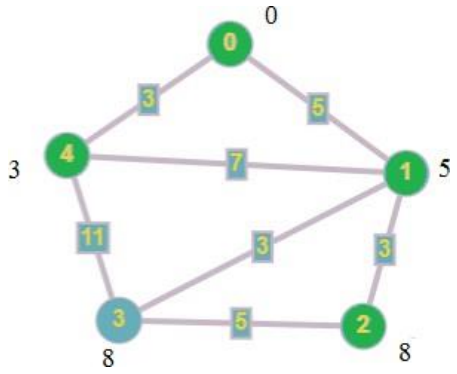
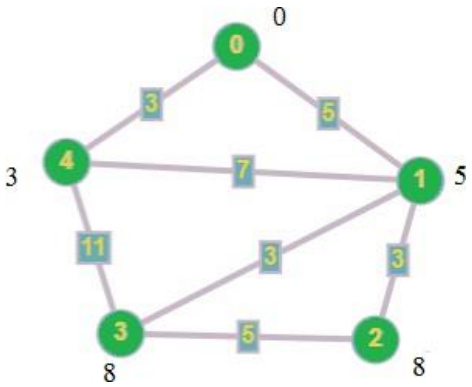


Итерация 2.
4 - вершина с наименьшим значением расстояния $4 = \{v_i | d[v_i] = \min\{d[v_j], j = 1..n\}$. Отмечаем ее посещенной u[4]=1. Для всех смежных с ней вершин, пытаемся улучшить расстояние. Для вершины 1 алгоритм оставил прежнее расстояние, для вершины 3 улучшил.



Итерация 3.
1 - вершина с наименьшим значением расстояния. Отмечаем ее посещенной. Для всех смежных с ней вершин, пытаемся улучшить расстояние. Для вершин 2 и 3 это удалось сделать.



Итерация 4. 2 - вершина с наименьшим значением расстояния. Отмечаем ее посещенной. Улучшений расстояния для смежных с ней вершин произвести не удалось.  	**Итерация 5.** 3 - вершина с наименьшим значением расстояния. Отмечаем ее посещенной. На этом алгоритм заканчивает свою работу. 
В конце работы алгоритма получаем наименьшие расстояния от всех вершин графа G до стартовой вершины: d[]={0, 5 8, 8, 3}. Например, от вершины 0 до вершины 3 кратчайшим образом надо двигаться по ребрам (0,1) - (1,3).	

Алгоритм Флойда-Уоршелла для определения кратчайших расстояний во взвешенном графе

Применяется для нахождения кратчайших путей от каждой вершины графа до всех остальных вершин в неориентированном (или ориентированном) взвешенном графе. Алгоритм назван в честь Роберта Флойда и Стивена Уоршелла и был предложен в 1962 году.

На вход программе подается граф, заданный в виде матрицы смежности d[][] размером *nxn*, в которой каждый элемент d[i][j] - длина ребра между вершинами v_i и v_j. Если i=j, то d[i][i]=0. Если две вершины v_i и v_j не являются смежными, то полагаем расстояние между ними d[i][j]=inf – большому числу (бОльшему, чем длины расстояний между другими вершинами в графе).

Алгоритм изменяет значения расстояний в матрице смежности графа в том случае, если расстояние d[i][j] можно улучшить, пройдя от v_i до v_j через промежуточную вершину v_k. Рассматривая всевозможные промежуточные вершины и оптимизируя каждый раз расстояние с учетом еще одной промежуточной вершины, получим n фаз алгоритма.

```
for (int k=0; k<n;++k)
     for (int i=0; i<n; ++i)
          for (int j=0; j<n; ++j)
               d[i][j]=min(d[i][j], d[i][k]+d[k][j]);
```

Сложность алгоритма $O(|V|^3)$.

Лекция 9

Общие сведения о системе

«1С:Предприятие» является универсальной системой автоматизации экономической и организационной деятельности предприятия. Поскольку такая деятельность может быть довольно разнообразной, система «1С:Предприятие» может приспосабливаться к особенностям конкретной области деятельности, в которой она применяется. Для обозначения такой способности используется термин *конфигурируемость*, то есть возможность настройки системы на особенности конкретного предприятия и класса решаемых задач.

Это достигается благодаря тому, что «1С:Предприятие» – это не просто программа, существующая в виде набора неизменяемых файлов, а совокупность различных программных инструментов, с которыми работают разработчики и пользователи. Логически всю систему можно разделить на две большие части, которые тесно взаимодействуют друг с другом, – *конфигурацию и платформу*, которая управляет работой конфигурации.

Для того чтобы легче понять взаимодействие этих частей системы, сравним ее с проигрывателем компакт-дисков. Как вы хорошо знаете, проигрыватель служит для того, чтобы слушать музыку. На вкус и цвет товарищей нет, поэтому существует множество разнообразных компакт-дисков, на которых записаны музыкальные произведения на любой вкус.

Чтобы прослушать какую-либо композицию, нужно вставить компакт-диск в проигрыватель, и проигрыватель воспроизведет записанное на диске музыкальное произведение. Более того, современный проигрыватель компакт-дисков даже позволит вам записать собственную подборку музыкальных произведений, то есть создать новый компакт-диск.

Сам по себе проигрыватель совершенно бесполезен без компактдиска, точно так же, как компакт-диск не может принести нам никакой пользы (кроме как стать подставкой под чашку кофе), если у нас нет проигрывателя.

Возвращаясь к системе «1С:Предприятие», можно сказать, что платформа является своеобразным «проигрывателем», а конфигурация – «компакт-диском». Платформа обеспечивает работу конфигурации и позволяет вносить в нее изменения или создавать собственную конфигурацию.

Существует одна платформа («1С:Предприятие») и множество конфигураций. Для функционирования какого-либо прикладного решения всегда необходима платформа и какая-либо (одна) конфигурация (рис. 1.1).

Рис. 1.1. Конфигураций много, а платформа одна

Сама по себе платформа не может выполнить никаких задач автоматизации, так как она создана для обеспечения работы какой-либо конфигурации. То же самое с конфигурацией: чтобы выполнить те задачи, для которых она создана, необходимо наличие платформы, управляющей ее работой.

Конфигурация и прикладное решение

Здесь следует сказать о небольшой двойственности терминологии, которая будет использоваться в дальнейшем. Двойственность заключается в употреблении разных терминов для обозначения одного и того же предмета: *конфигурация* и *прикладное решение*.

Эти термины обозначают ту часть системы «1С:Предприятие», которая работает под управлением платформы и которую видят все пользователи. Бывает, конечно, что пользователи работают и с инструментальными средствами платформы, но это продвинутые пользователи. Употребление одного или другого термина зависит от контекста, в котором ведется изложение.

Если речь идет о действиях разработчика, то употребляется термин «конфигурация», поскольку это точный термин «1С:Предприятия». Термин «прикладное решение», напротив, является более общепринятым и понятным для пользователя системы «1С:Предприятие».

Итак, поскольку задачи автоматизации, как было упомянуто выше, могут быть самыми разными, фирма «1С» и ее партнеры выпускают прикладные решения, каждое из которых предназначено для автоматизации одной определенной области человеческой деятельности. В качестве примера существующих прикладных решений можно перечислить следующие типовые решения:

- «1С:Бухгалтерия 8»,
- «1С:Управление небольшой фирмой 8»,
- «1С:Управление торговлей 8»,
- «1С:Зарплата и управление персоналом 8»,
- «1С:Управление производственным предприятием 8»,
- «1С:Налогоплательщик 8»,
- «1С:Документооборот 8», «1С:Консолидация 8».

Существует также множество других типовых прикладных решений. Более подробно о них можно узнать на сайте http://v8.1c.ru/solutions/applied_solutions.htm.

Типовое прикладное решение является, по сути, универсальным и способно удовлетворить потребности самых разных предприятий, работающих в одной области деятельности. И это хорошо.

С другой стороны, такая универсальность неизбежно приведет к тому, что на конкретном предприятии будут использоваться далеко не все возможности прикладного решения, а каких-то возможностей в нем будет недоставать (нельзя угодить всем).

Вот тут и выходит на передний план *конфигурируемость* системы, поскольку платформа, помимо управления работой конфигурации, содержит средства, позволяющие вносить изменения в используемую конфигурацию. Более того, платформа позволяет создать свою собственную конфигурацию с нуля, если по каким-либо причинам использование типовой конфигурации представляется нецелесообразным.

Обратите внимание, как мы в одном абзаце перешли от *прикладного решения* к *конфигурации*. Ничего не поделаешь, для пользователя понятнее так, а для разработчика – по-другому.

Таким образом, если вернуться к сравнению с проигрывателем компакт-дисков, мы можем изменять по своему вкусу мелодии, которые были ранее записаны на компакт-диске, и даже создавать диски со своими собственными музыкальными произведениями. При этом нам не потребуются какие-либо музыкальные инструменты – все необходимое для создания мелодий есть в нашем проигрывателе компакт-дисков.

Режимы работы системы

Для того чтобы обеспечить такие возможности, система «1С:Предприятие» имеет различные режимы работы: *1С:Пред- приятие* и *Конфигуратор*.

Режим 1С:Предприятие является основным и служит для работы пользователей системы. В этом режиме пользователи вносят данные, обрабатывают их и получают итоговые результаты.

Режим Конфигуратор используется разработчиками и администраторами информационных баз. Именно этот режим и предоставляет инструменты, необходимые для модификации существующей или создания новой конфигурации.

Поскольку задача дициплины состоит в том, чтобы научить вас создавать собственные конфигурации и изменять существующие, дальнейшее повествование будет в основном посвящено работе с системой в режиме Конфигуратор. И лишь иногда, чтобы проверить результаты нашей работы, мы будем запускать систему в режиме 1С:Предприятие.

Изучение этой дисциплины предполагает, что у вас уже установлена на компьютере система «1С:Предприятие 8.3». Если это не так, то сейчас самое время это сделать, так как далее будет непосредственно описываться последовательность работы с программой.

Что такое подсистема

Подсистемы – это основные элементы для построения интерфейса «1С:Предприятия». Поэтому первое, с чего следует начинать разработку конфигурации, – это проектирование состава подсистем.

При этом перед разработчиком стоит важная и ответственная задача – тщательно продумать состав подсистем и затем аккуратно и осмысленно привязать к подсистемам те объекты конфигурации, которые он будет создавать.

В простых прикладных решениях можно не использовать подсистемы, но мы рассмотрим общий случай, когда подсистемы используются.

Объекты конфигурации Подсистема позволяют выделить в конфигурации функциональные части, на которые логически разбивается создаваемое прикладное решение.

Эти объекты располагаются в ветке объектов Общие и позволяют строить древовидную структуру, состоящую из подсистем и подчиненных им подсистем (рис. 2.1).

Рис. 2.1. Структура подсистем конфигурации

Подсистемы верхнего уровня являются основными элементами интерфейса, так как образуют разделы прикладного решения

(рис. 2.2).

Рис. 2.2. Разделы прикладного решения

Каждый объект конфигурации может быть включен в одну или сразу несколько подсистем, в составе которых он будет отображаться.

Забегая вперед, скажем, что с помощью подсистем, используя видимость по ролям, можно предоставить пользователю удобный и функциональный интерфейс, не содержащий лишних элементов. Например, кладовщик должен иметь возможность принять и выдать товар, и ему совсем не нужно видеть все, что относится к области бухгалтерского учета и оказанию услуг.

Таким образом, наличие подсистем определяет структуру прикладного решения, организует весь пользовательский интерфейс, позволяет рассортировать различные документы, справочники и отчеты по логически связанным с ними разделам, в которых пользователю будет проще их найти и удобнее с ними работать. При этом каждому конкретному пользователю будут видны лишь те разделы, то есть та функциональность прикладного решения, которые ему нужны в процессе работы.

Даже в такой небольшой конфигурации, как наша, можно выделить несколько функциональных частей, представляющих собой отдельные предметные области.

Так, можно выделить в отдельную подсистему все, что имеет отношение к бухгалтерскому учету.

Кроме этого, отдельной предметной областью является расчет зарплаты сотрудников предприятия.

Всю производственную деятельность нашей фирмы ООО «На все руки мастер» можно разделить на учет материалов и оказание услуг.

А кроме этого, для выполнения специальных административных функций с базой данных нам нужно иметь отдельную подсистему, в которую будет иметь доступ только администратор.

Поэтому сейчас мы создадим в нашей конфигурации пять новых объектов конфигурации Подсистема, которые будут иметь имена:

Бухгалтерия, РасчетЗарплаты, УчетМатериалов, ОказаниеУслуг и Предприятие. Чтобы это сделать, выполним следующие действия.

На первый взгляд окно редактирования объекта и палитра свойств дублируют друг друга. В самом деле в палитре свойств отображены все свойства объекта конфигурации. Зачем было создавать еще и окно редактирования объекта? И если существует окно редактирования объекта, то зачем тогда палитра свойств, которая содержит все то же самое, только в другом виде?

Окно редактирования объекта конфигурации предназначено в первую очередь для быстрого создания новых объектов. Быстрое создание подразумевает ввод исчерпывающей

информации об объекте. Значит, нужно очень хорошо знать структуру объекта, а на это требуется время… Выходит, быстро создать объект не получится?

Получится! Окно редактирования объекта имеет в своей основе механизм «мастеров»: разработчику в нужной последовательности предлагается ввести необходимые данные. Последовательность ввода данных разработана таким образом, чтобы предыдущие данные могли служить основой для ввода последующих. Движение управляется кнопками Далее и Назад. На каждом шаге предлагается ввести группу логически связанных между собой данных.

Но, предположим, вы уже освоились со структурой объектов, или вам просто нужно изменить несколько свойств объекта. Чтобы при этом опять не «прокручивать» все с самого начала, окно редактирования объекта содержит закладки, позволяющие вам перейти непосредственно к тому шагу, на котором вводятся интересующие вас данные. Таким образом, окно редактирования объекта помогает быстро создать незнакомый объект конфигурации и в то же время обеспечивает удобный доступ к нужным свойствам при редактировании существующих объектов.

Что же касается палитры свойств, то она предоставляет одну абсолютно незаменимую возможность. Дело в том, что она не привязана по своей структуре к какому-то конкретному виду объектов конфигурации. Ее содержимое меняется в зависимости от того, какой объект является текущим. За счет этого она может запоминать, какое свойство объекта в ней выбрано, и при переходе в дереве к другому объекту будет подсвечивать у себя все то же свойство, но уже другого объекта.

Такая способность палитры свойств абсолютно незаменима, когда, например, среди трех десятков справочников конфигурации вам нужно быстро найти подчиненные какому-нибудь другому. В этом случае вы выбираете мышью в палитре свойств свойство Владелец любого справочника, затем переходите в дерево объектов конфигурации и просто пробегаете его при помощи стрелок или .

Что такое справочник

Объект конфигурации Справочник предназначен для работы со списками данных. Как правило, в работе любой фирмы используются списки сотрудников, списки товаров, списки клиентов, поставщиков и т. д. Свойства и структура этих списков описываются в объектах конфигурации Справочник, на основе которых платформа создает в базе данных таблицы для хранения информации из этих справочников.

Справочник состоит из *элементов*. Например, для справочника сотрудников элементом является сотрудник, для справочника товаров – товар и т. д. Пользователь в процессе работы может самостоятельно добавлять новые элементы в справочник: например, добавить новых сотрудников, создать новый товар или внести нового клиента.

В базе данных каждый элемент справочника представляет собой отдельную запись в основной таблице, хранящей информацию из этого справочника (рис. 3.1).

Каждый элемент справочника, как правило, содержит некоторую дополнительную информацию, которая подробнее описывает этот элемент. Например, все элементы справочника Товары могут содержать дополнительную информацию о производителе, сроке годности и др. Набор такой информации является одинаковым для всех элементов

справочника, и для описания такого набора используются *реквизиты* объекта конфигурации Справочник, которые также, в свою очередь, являются объектами конфигурации (рис. 3.2).

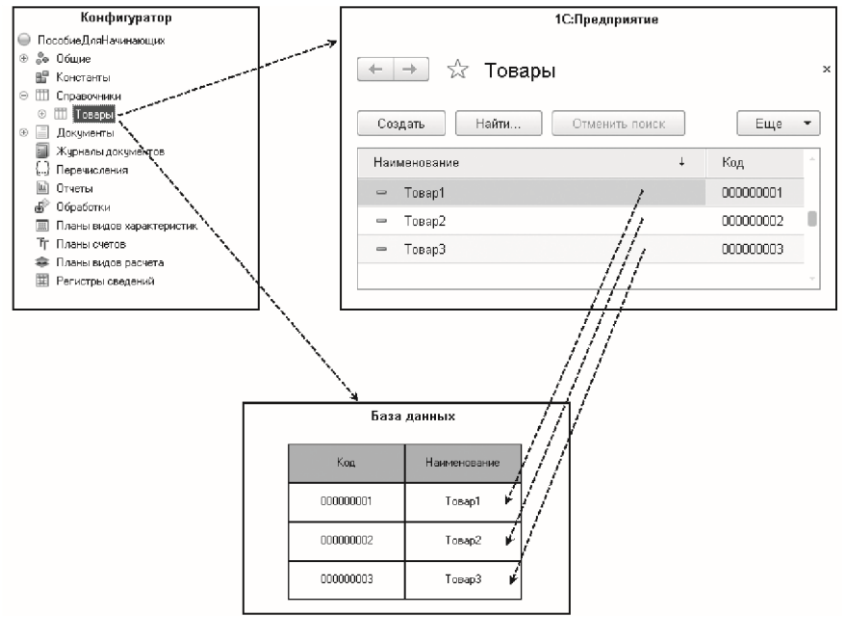


Рис. 3.1. Справочник «Товары» в режиме «Конфигуратор», в режиме «1С:Предприятие» и в базе данных

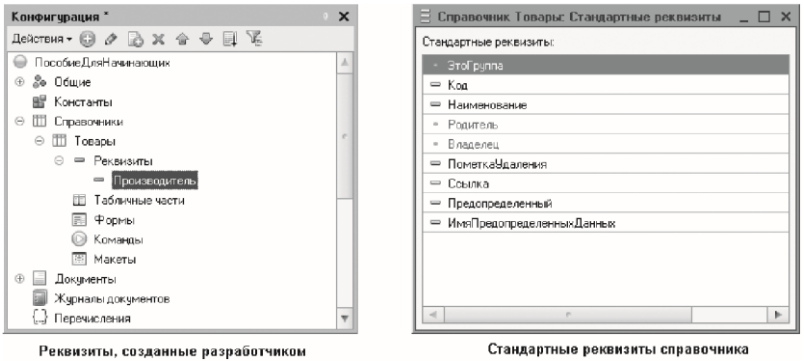


Рис. 3.2. Стандартные реквизиты справочника и реквизиты, созданные разработчиком

Поскольку эти объекты конфигурации логически связаны с объектом Справочник, они называются *подчиненными* этому объекту.

Большинство реквизитов разработчик создает самостоятельно, однако у каждого объекта конфигурации Справочник по умолчанию существует набор стандартных реквизитов: Код и Наименование и пр. (см. рис. 3.2). Причем доступность стандартных реквизитов зависит от свойств справочника.

Например, если справочник иерархический, у него будет доступен стандартный реквизит Родитель. Если справочник подчинен другому объекту конфигурации, у него будет доступен реквизит Владелец. Если установить длину стандартного реквизита Код равной нулю, то у справочника будет недоступен этот реквизит. То же самое относится к реквизиту Наименование. Однако как минимум либо Код, либо Наименование должны присутствовать в реквизитах справочника, иначе такой справочник не имеет смысла.

Таким образом, в базе данных справочник хранится в виде таблицы, в строках которой расположены элементы списка, а каждому реквизиту (стандартному или созданному

разработчиком) в этой таблице соответствует отдельный столбец. Соответственно, в ячейках этой таблицы хранится значение конкретного реквизита для конкретного элемента справочника (рис. 3.3).

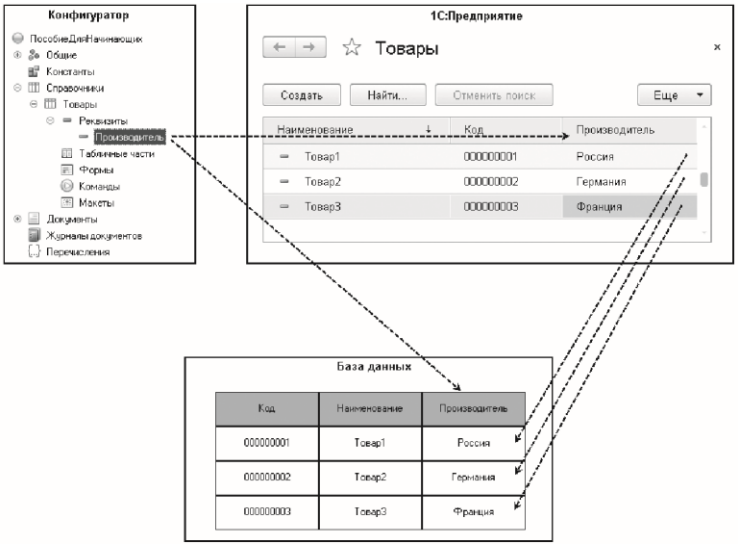


Рис. 3.3. Справочник «Товары» в режиме «Конфигуратор», в режиме «1С:Предприятие» и в базе данных

Кроме этого, каждый элемент справочника может содержать некоторый набор информации, которая одинакова по своей структуре, но различна по количеству и предназначена для разных элементов справочника.

Формы справочника

В зависимости от того, какие действия мы хотим выполнять со справочником, нам требуется изображать справочник в «разном виде». Например, для того чтобы выбрать некоторый элемент справочника, удобнее представить справочник в виде списка, а для того чтобы изменить какой-то элемент справочника, удобнее представить все реквизиты этого элемента справочника в одной форме

(рис. 3.9).

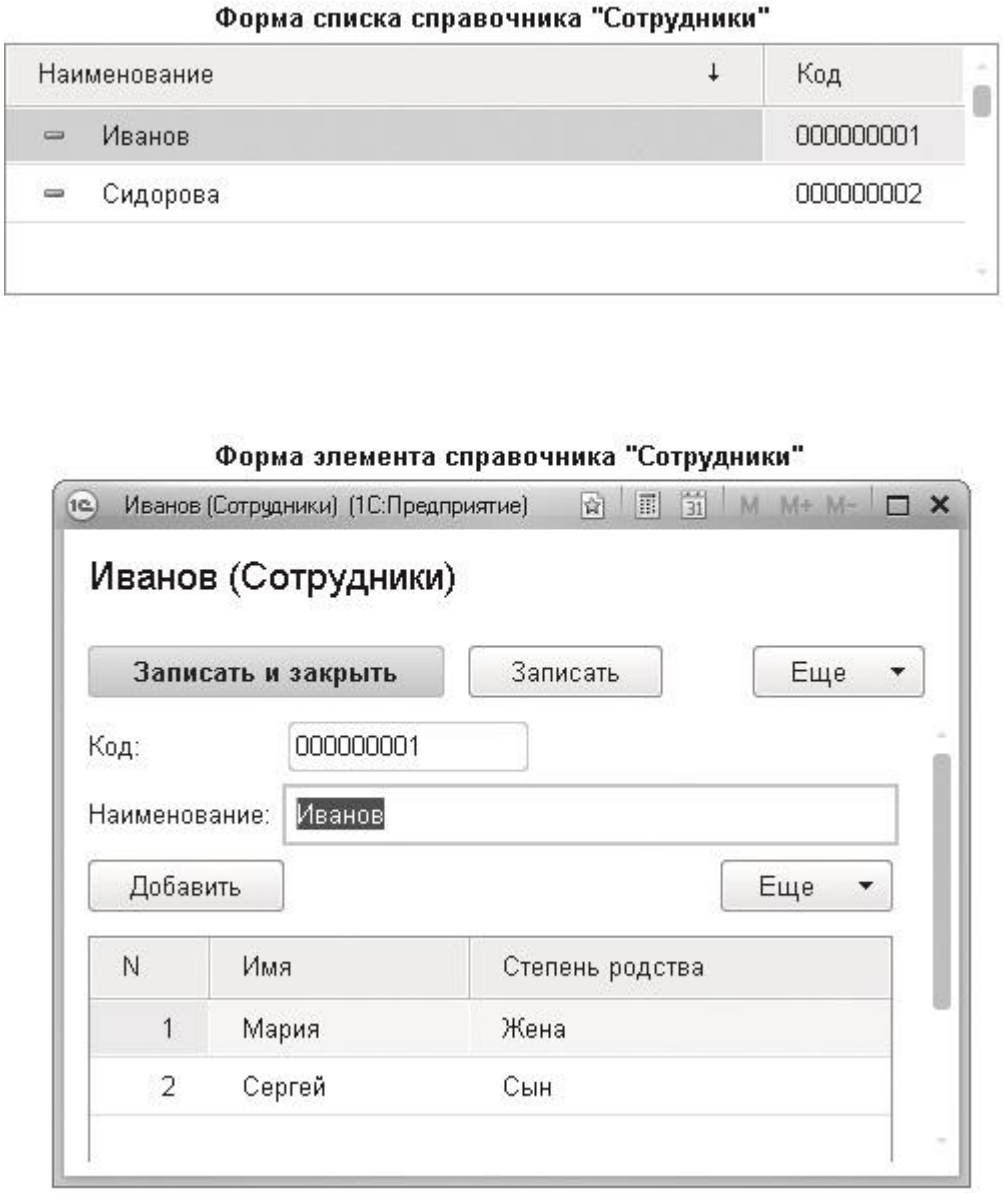


Рис. 3.9. Форма списка и форма редактирования элемента справочника «Сотрудники»

Система может самостоятельно сгенерировать все формы, которые нужны для представления данных, содержащихся в справочнике. Причем система знает, какие именно формы нужно использовать в каких ситуациях.

Вообще говоря, для отображения справочника в различных ситуациях требуется максимум пять форм для справочника.

Обратите внимание, что в различных местах конфигуратора одни и те же формы называются немного по-разному (табл. 3.1). Следующая таблица представляет различные названия форм:

- в контекстном меню справочника (Открыть основную форму…), в дереве конфигурации и в палитре свойств справочника;
- в конструкторе форм;
- на закладке Формы окна редактирования справочника.

таблица 3.1. Формы справочника

В контекстном меню и в палитре свойств (рис. 3.12)	В конструкторе форм (рис. 3.11)	На закладке «Формы» (рис. 3.10)
Основная форма объекта	Форма элемента справочника	Элемента
Основная форма группы	Форма группы справочника	Группы
Основная форма списка	Форма списка справочника	Списка
Основная форма выбора	Форма выбора справочника	Выбора
Основная форма выбора группы	Форма выбора группы справочника	Выбора группы

Предопределенные элементы

Обратите внимание, что система отмечает различными пиктограммами обычный и предопределенный элементы справочника

(см. рис. 3.73).

Несмотря на то, что можно изменить код или наименование у обоих элементов, имя предопределенного элемента, которое мы задали в конфигураторе (Основной), остается неизменным, и в дальнейшем мы сможем обратиться к предопределенному элементу справочника по этому имени из встроенного языка.

Таким образом, на предопределенные элементы могут опираться алгоритмы работы конфигурации.

Из этого видно, в чем заключается принципиальная с точки зрения конфигурации разница между обычными и предопределенными элементами справочника.

Обычные элементы непостоянны для конфигурации. В процессе работы пользователя они могут появиться, исчезнуть. Поэтому конфигурация хоть и может отличить их друг от друга, но рассчитывать на них в выполнении каких-либо алгоритмов она не может в силу их непостоянства.

Предопределенные элементы, напротив, постоянны. В процессе работы пользователя они находятся всегда на своих местах и исчезнуть не могут.

То есть теоретически пользователь может их удалить, но для облегчения задачи мы не даем пользователю прав не только на интерактивное удаление предопределенных элементов, но и на интерактивное удаление объектов вообще.

Поэтому конфигурация может работать с ними вполне уверенно и опираться на них при отработке различных алгоритмов. По этой причине каждый из предопределенных элементов имеет уникальное имя для того, чтобы к нему можно было обратиться средствами встроенного языка.

Основная конфигурация и конфигурация базы данных

Кроме этого, вне информационной базы может существовать хранилище. В нем находится конфигурация, предназначенная для групповой разработки.

Вне информационной базы может существовать также некоторое количество файлов конфигураций, в том числе файлы новой поставки (рис. 3.74).

Конфигурация поставщика, находящаяся в информационной базе, содержит предыдущее состояние поставки. Возможна ситуация, когда конфигурация находится на поддержке одновременно у нескольких поставщиков, каждый из которых поддерживает только свою часть в виде отдельной конфигурации. В этом случае информационная база будет хранить несколько конфигураций поставщиков (состояние предыдущей поставки для каждого поставщика отдельно).

Файлы новой поставки могут существовать в виде файлов конфигураций (*полная поставка*) и файлов обновлений (*поставка обновлений*).

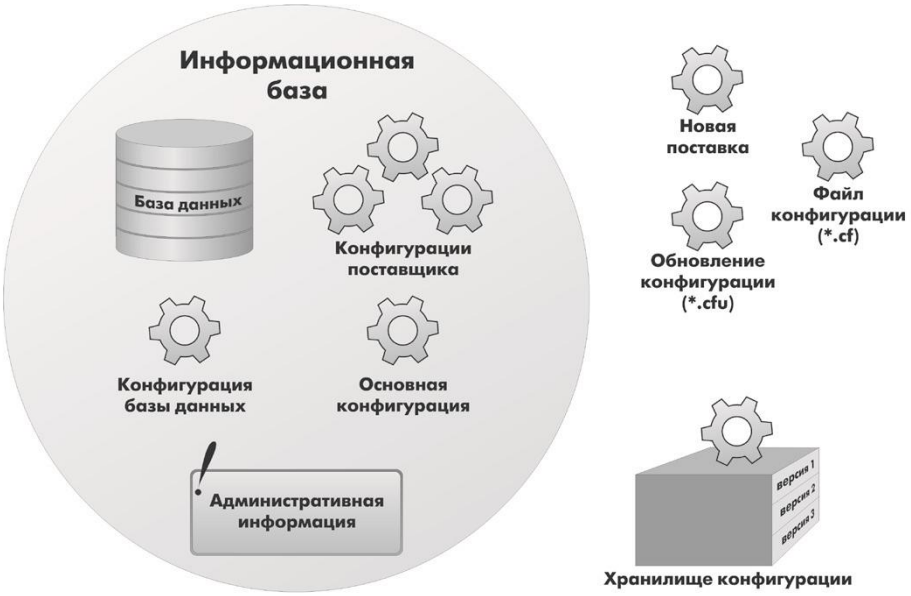


Рис. 3.74. Структура конфигурации

Хранилище конфигурации содержит конфигурацию, предназначенную для групповой разработки. Она хранится не в виде единой конфигурации, а в виде отдельных объектов в разрезе версий конфигурации. Таким образом, мы можем получить из хранилища конфигурацию любой версии – для этого она «собирается» из объектов нужной версии.

Теперь представьте, что между всеми этими видами конфигураций существует возможность сравнения и обновления. В этом случае очень легко запутаться, и название *Основная конфигурация* как нельзя лучше отражает конечную цель всех изменений.

Что такое документ

Объект конфигурации *Документ* предназначен для описания информации о совершенных хозяйственных операциях или о событиях, произошедших в жизни организации вообще. Как правило, в работе любой фирмы используются такие документы, как приходные накладные, приказы о приеме на работу, платежные поручения, счета и т. д. Свойства и структура этих документов описываются в объектах конфигурации Документ, на основе которых платформа создает в базе данных таблицы для хранения информации из этих документов.

Логика работы документов отличается от логики работы других объектов конфигурации. Документ обладает способностью *проведения*. Факт проведения документа означает, что событие, которое он отражает, повлияло на состояние учета.

До тех пор, пока документ не проведен, состояние учета неизменно, и документ не более чем черновик, заготовка. Как только документ будет проведен, изменения, вносимые документом в учет, вступят в силу и состояние учета будет изменено.

Поскольку документ вносит изменения в состояние учета, он всегда «привязан» к конкретному моменту времени. Это позволяет отражать в базе данных фактическую последовательность событий.

Следующим важным фактом, вытекающим из двух предыдущих, является то, что система «1С:Предприятие» имеет механизмы, позволяющие отслеживать правильность состояния учета. Предположим, что мы изменили один из проведенных ранее документов и снова провели его задним числом. В этом случае система «1С:Предприятие» способна отследить, повлияют ли внесенные нами изменения на последующие проведенные документы, и, если это так, система способна перепровести необходимые документы.

В процессе работы пользователь может самостоятельно создавать новые документы – приходные и расходные накладные, счета и т. п.

В базе данных каждый документ представляет собой отдельную запись в основной таблице, хранящей информацию об этом виде документов (рис. 4.1).

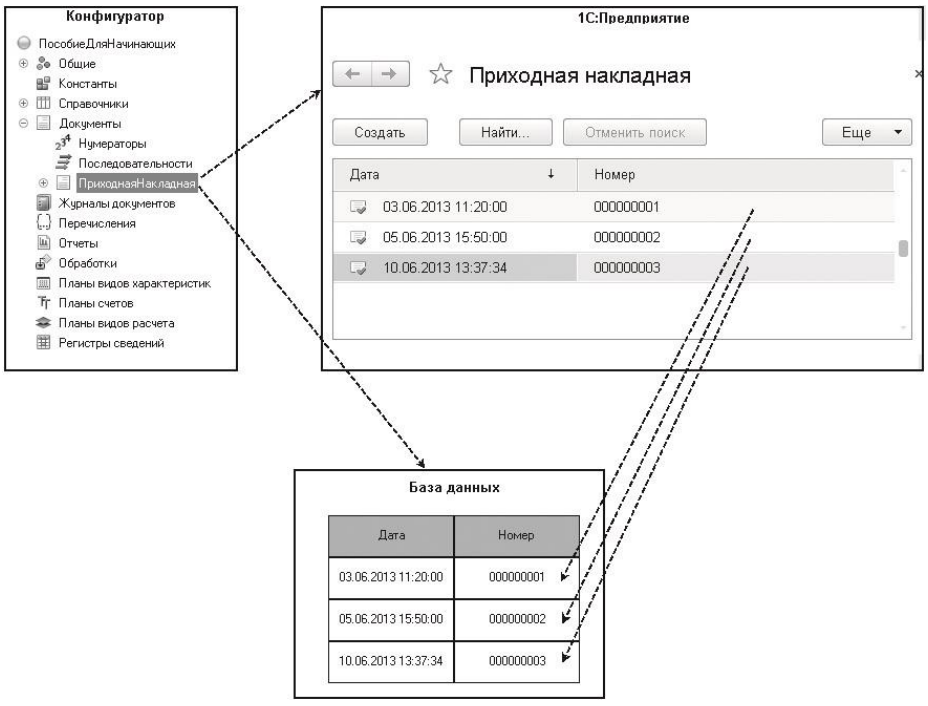


Рис. 4.1. Стандартные реквизиты документа «Приходная накладная» в режиме «Конфигуратор», в режиме «1С:Предприятие» и в базе данных

Формы документа

Для визуализации документа существует несколько основных форм, которые, как мы уже говорили, имеют несколько вариантов названий (табл. 4.1).

таблица 4.1. Основные формы документа

В контекстном меню и в палитре свойств	В конструкторе форм	на закладке формы
Форма объекта	Форма документа	Документа
Форма списка	Форма списка документа	Списка
Форма выбора	Форма выбора документа	Выбора

типы данных, типообразующие объекты конфигурации

Прежде чем мы приступим к практическому созданию документов, необходимо сделать отступление о том, какие типы данных могут использоваться в системе «1С:Предприятие».

На предыдущем занятии, когда мы создавали реквизиты справочников или табличных частей, мы всегда указывали тип значения, которое может принимать этот реквизит. Это были *примитивные* типы данных: Число, Строка, Дата и Булево. Примитивные типы данных изначально определены в системе, и их набор ограничен.

Наряду с такими изначально определенными в любой конфигурации типами могут существовать типы данных, определяемые только конкретной конфигурацией. То есть такие типы, которые не присутствуют в конфигурации постоянно, а появляются в результате того, что добавлены некоторые объекты конфигурации.

Например, после того как мы создали объект конфигурации Справочник Склады, сразу же появилось несколько новых типов данных, связанных с этим справочником. Среди них, например, СправочникСсылка.Склады. И если теперь мы укажем какому-либо реквизиту этот тип данных, то сможем хранить в нем ссылку на конкретный объект справочника Склады.

Объекты конфигурации, которые могут образовывать новые типы данных, называются *типообразующими*.

Например, после создания нового справочника Номенклатура становятся доступны следующие типы данных: СправочникМенеджер.Номенклатура,

- СправочникСсылка.Номенклатура,
- СправочникОбъект.Номенклатура,
- СправочникВыборка.Номенклатура.

Следует еще раз отметить, что эти типы данных не поддерживаются платформой изначально и существуют только в конкретном прикладном решении.

Это небольшое отступление было необходимо потому, что уже при создании первого документа мы столкнемся с использованием типов данных СправочникСсылка.Склады и СправочникСсылка.Номенклатура, которые появились в нашей конфигурации в результате создания объектов конфигурации Справочник Склады и Номенклатура.

Документ «Приходная накладная»

После того как мы познакомились с объектом конфигурации Документ, создадим несколько таких объектов, чтобы иметь возможность фиксировать события, происходящие в нашем ООО «На все руки мастер».

Одними из самых популярных услуг нашего предприятия является ремонт телевизоров и установка стиральных машин. И в том, и в другом случае требуются некоторые материалы, которые расходуются в процессе оказания этих услуг. Поэтому двумя важнейшими событиями в хозяйственной жизни нашей организации будут являться поступление материалов и оказание услуг.

Для отражения этих событий в базе данных мы создадим два документа: Приходная накладная и Оказание услуги.

Документ Приходная накладная будет фиксировать факт поступления в нашу организацию необходимых материалов, а документ Оказание услуги – фиксировать оказание услуг и расход материалов, которые используются при оказании этих услуг.

Механизм основных форм

На предыдущем занятии мы создали форму документа Приходная накладная и назначили эту форму основной. Что это значит?

У всех прикладных объектов конфигурации существует некоторое количество основных форм. Они служат для отображения данных объекта в том или ином виде.

Если разработчик не назначит в качестве основных форм объекта свои собственные, система будет генерировать необходимые формы объекта самостоятельно, в те моменты, когда к ним происходит обращение.

Наличие такого механизма позволяет разработчику не тратить время на создание форм для тестирования своей разработки, а воспользоваться тем, что платформа создаст по умолчанию.

Создание этих форм происходит динамически, в процессе работы системы. Форма создается в тот момент, когда к ней происходит обращение. Причем не важно, интерактивное это обращение или программное.

Так, форма списка для справочника Клиенты будет создана как при интерактивном выборе в меню Все функции Справочники Клиенты, так и при программном вызове глобального метода ПолучитьФорму() (листинг 5.1).

листинг Программный вызов метода «ПолучитьФорму()»

```
ФормаСписка = ПолучитьФорму("Справочник.Клиенты.ФормаСписка");
```

Также примечательным фактом является то, что состав основных форм, определенных для объекта конфигурации, может не совпадать с перечнем тех форм, которые вообще можно создать для данного объекта, используя конструктор формы.

Например, для большинства регистров в конфигураторе можно задать основную форму списка. Однако если открыть конструктор форм для регистра, вы увидите, что кроме формы списка предлагается создать и форму набора записей регистра, которая отсутствует в перечне основных форм.

Дело в том, что состав основных форм определяется исходя из того, какое представление данных может понадобиться в процессе интерактивной работы пользователя. Для этих представлений разработчик может создать свои формы и указать их в качестве основных, а может использовать те формы, которые система создаст автоматически.

Конструктор форм, напротив, исходит из потребностей разработчика. Если разработчик посчитает нужным использовать для какого-либо регистра вместо обычной формы списка форму набора записей, он сможет это сделать, воспользовавшись конструктором и определив ее в качестве основной формы регистра. Но для логики работы системы это не будет иметь принципиального значения.

Обработчики событий

При работе с событиями на платформе «1С:Предприятие» следует различать два типа событий: события, связанные с формой и ее элементами, и все остальные.

Разница заключается в том, что обработчики событий, связанных с формой и ее элементами, – назначаемые, а обработчики всех остальных событий – фиксированные.

Фиксированный обработчик события должен иметь имя, совпадающие с именем события. Только в этом случае он будет вызываться при возникновении соответствующего события.

Назначаемый обработчик может иметь произвольное имя. Если имя процедуры совпадает с именем события формы или ее элемента, этого совсем недостаточно для вызова процедуры обработки события с таким именем. Требуется явное назначение процедуры обработчиком этого события в палитре свойств, в соответствующем событии.

Таким образом, любая процедура, расположенная в модуле формы, может быть назначена обработчиком любого события (или сразу нескольких событий) формы или ее элемента, расположенного в форме. Имя процедуры в этом случае не имеет значения. Важно лишь то, что она назначена обрабатывать какое-либо событие.

Назначение обработчика может выполняться интерактивно, при работе с формой в конфигураторе, или программно, используя методы формы и ее элементов – УстановитьДействие().

Модули

На предыдущем занятии мы рассматривали код обработчиков событий. Мы узнали, что эти процедуры располагаются в модуле формы – неком хранилище текста программы на встроенном языке.

Теперь расскажем о модулях подробнее и внимательнее познакомимся с устройством модуля формы.

Виды модулей

В конфигурации существуют различные виды модулей. Они могут принадлежать некоторым объектам конфигурации (например, формам), а могут существовать сами по себе (принадлежать всей конфигурации в целом).

Текст программы, содержащийся в модулях, будет использоваться платформой в заранее известные моменты работы системы «1С:Предприятие» – *события*, о которых мы рассказывали ранее.

В «1С:Предприятии» существуют следующие виды модулей.

Модуль управляемого приложения. Модуль управляемого приложения выполняется при старте и окончании работы системы «1С:Предприятие» в режимах тонкого клиента и веб-клиента.

В нем возможно объявление переменных, а также объявление и описание процедур и функций, которые будут доступны в любом модуле конфигурации (кроме модуля внешнего соединения). Их доступность также обеспечивается для неглобальных общих модулей с установленным свойством Клиент (управляемое приложение). В контексте модуля управляемого приложения доступны экспортируемые процедуры и функции общих модулей.

Чтобы открыть модуль управляемого приложения, нужно выделить корень дерева объектов конфигурации (строку ПособиеДляНачинающих) и вызвать из контекстного меню команду Открыть модуль управляемого приложения (рис. 5.1).

Общие модули. В общих модулях хранятся процедуры и функции, которые вызываются из других модулей системы. Сам по себе общий модуль не исполняется. Исполняются отдельные его процедуры/ функции в момент их вызова из других модулей.

Чтобы открыть общий модуль, нужно раскрыть ветвь Общие в дереве объектов конфигурации, затем раскрыть ветвь Общие модули и дважды щелкнуть мышью на нужном модуле (рис. 5.2).

Модули объектов. Модули объектов – это, например, модуль элемента справочника или модуль документа.

Эти модули вызываются тогда, когда программно создается этот объект средствами встроенного языка, например, методами СоздатьЭлемент() менеджеров справочников или СоздатьДокумент() менеджеров документов, либо когда пользователь создает новый элемент справочника или документ интерактивно.

При записи измененных данных объекта в базу данных вызываются различные обработчики событий, которые располагаются в модуле объекта.

Чтобы открыть модуль объекта, нужно в окне редактирования объекта конфигурации перейти на закладку Прочее и нажать кнопку Модуль объекта (рис. 5.3). Или, выделив нужный объект в дереве объектов конфигурации, вызвать из контекстного меню команду Открыть модуль объекта.

Модули форм. Каждая форма, определенная в конфигурации, имеет свой собственный модуль. Этот модуль исполняется при создании объекта УправляемаяФорма встроенного языка. Этот объект создается системой в режиме 1С:Предприятие в тот момент, когда мы программно (методами ПолучитьФорму() или ОткрытьФорму()) или интерактивно открываем форму некоторого элемента.

Чтобы открыть модуль формы, нужно открыть подчиненный объект Форма нужного объекта конфигурации и в окне редактора форм перейти на закладку Модуль (рис. 5.4).

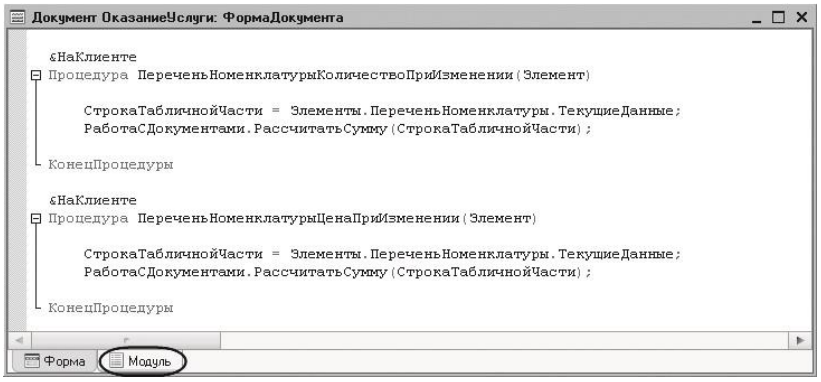


Рис. 5.4. Открытие модуля формы

Модуль сеанса. Модулем сеанса называется модуль, который автоматически выполняется при старте системы «1С:Предприятие» в момент загрузки конфигурации. Модуль сеанса предназначен для инициализации параметров сеанса и отработки действий, связанных с сеансом работы. Модуль сеанса не содержит экспортируемых процедур и функций и может использовать процедуры из общих модулей конфигурации.

Чтобы открыть модуль сеанса, нужно выделить корень дерева объектов конфигурации (строку ПособиеДляНачинающих) и вызвать из контекстного меню команду Открыть модуль сеанса (рис. 5.5).

Модуль внешнего соединения предназначен для размещения в нем текстов функций и процедур, которые могут вызываться в сессии внешнего соединения.

Чтобы открыть модуль сеанса, нужно выделить корень дерева объектов конфигурации (строку ПособиеДляНачинающих) и вызвать из контекстного меню команду Открыть модуль внешнего соединения (см. рис. 5.5).

Модуль менеджера. Для каждого прикладного объекта существует менеджер, предназначенный для управления этим объектом как объектом конфигурации. С помощью менеджера можно создавать объекты, работать с формами и макетами. Модуль менеджера позволяет расширить функциональность менеджеров, предоставляемых системой, за счет написания процедур и функций на встроенном языке.

Рис. 5.15. Описание объектов в синтакс-помощнике

Таким образом, в модуле формы, где основной реквизит содержит данные документа (рис. 5.16), можно обратиться к свойству расширения управляемой формы для документа АвтоВремя (листинг 5.7).

Зачем нужен регистр накопления

Итак, мы с вами подошли к одному из главных моментов разработки любой конфигурации – созданию механизма учета накопления данных.

Казалось бы, все необходимое мы с вами уже создали: у нас есть что расходовать и приходовать (справочники), и у нас есть чем расходовать и приходовать (документы). Осталось только построить несколько отчетов, и автоматизация ООО «На все руки мастер» будет закончена.

Однако это не так.

Во-первых, путем анализа документов можно, конечно, получить требуемые нам выходные данные. Но представьте, что завтра ООО «На все руки мастер» решит немного изменить свои бизнеспроцессы, и нам потребуется ввести в конфигурацию еще один документ (или несколько документов).

Например, сейчас мы полагаем, что товары поступают в ООО «На все руки мастер» и затем расходуются. Руководство захотело усилить материальный контроль и решило приходовать товары на основной склад организации и затем выдавать их материально ответственным лицам. В этом случае нам придется добавить в конфигурацию еще один документ, который будет фиксировать перемещение материалов между основным складом и материально ответственными лицами. И очевидно, нам придется переработать все отчеты, которые были нами созданы к этому моменту, с тем, чтобы они учитывали изменения, вносимые новым документом. А представьте, если в нашей конфигурации не два, а двадцать документов?!

Во-вторых, отчеты, анализирующие документы, будут работать довольно медленно, что будет вызывать раздражение пользователей и недовольство руководителей.

Поэтому в системе «1С:Предприятие» есть несколько объектов конфигурации, которые позволяют создавать в базе данных структуры, предназначенные для накопления информации в удобном для последующего анализа виде. Использование таких хранилищ данных позволяет нам, с одной стороны, накапливать в них данные, поставляемые различными документами (или другими объектами базы данных), а с другой стороны, легко создавать нужные нам отчеты или использовать эти данные в алгоритмах работы конфигурации (рис. 6.1).

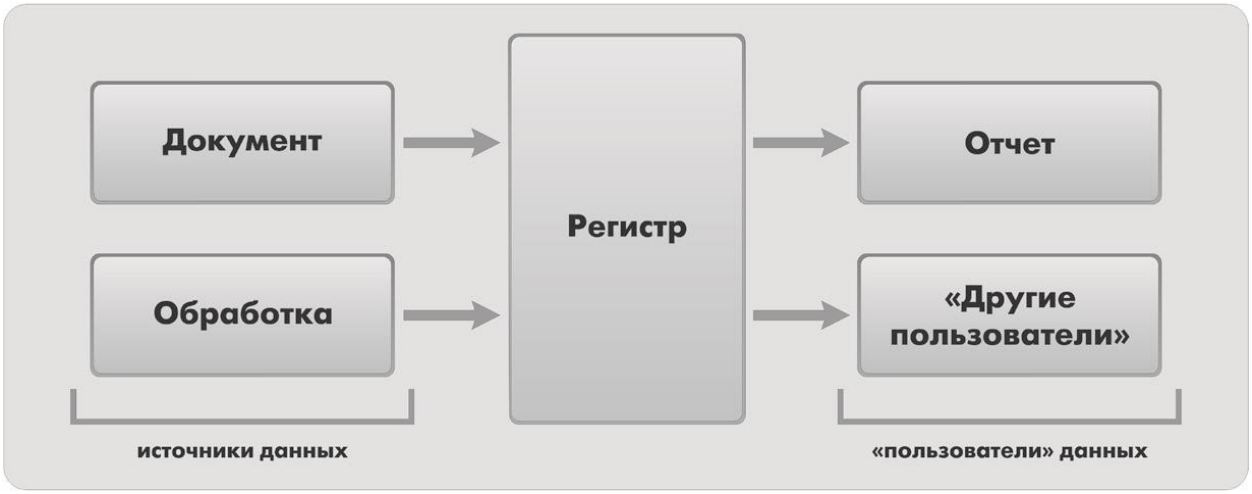


Рис. 6.1. Алгоритм работы конфигурации

В конфигурации существует несколько объектов, называемых *регистрами*, для описания подобных хранилищ. Сейчас мы рассмотрим один из них.

Что такое регистр накопления

Объект конфигурации *Регистр накопления* предназначен для описания структуры накопления данных. На основе объекта конфигурации Регистр накопления платформа создает в базе данных таблицы, в которых будут накапливаться данные, поставляемые различными объектами базы данных.

Эти данные будут храниться в таблицах в виде отдельных записей, каждая из которых имеет одинаковую заданную в конфигураторе структуру (рис. 6.2).

На основании таблицы движений регистра накопления система рассчитывает таблицу итогов регистра, которая хранит в базе данных итоги на момент времени последнего движения (актуальные итоги).

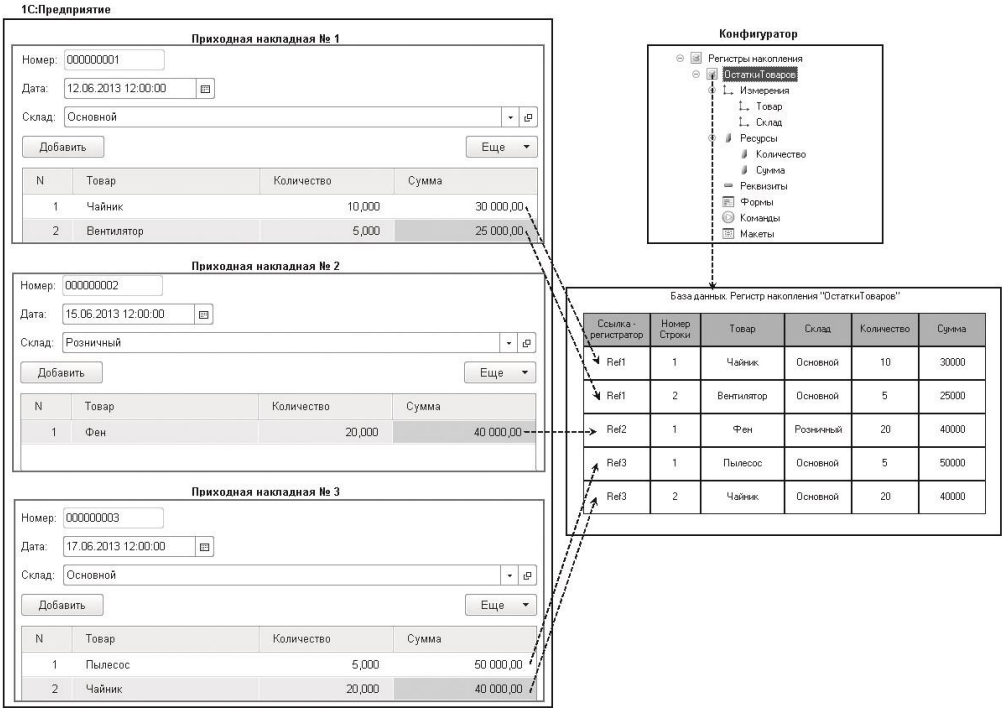


Рис. 6.2. Регистр накопления «Остатки товаров» в конфигураторе и в базе данных

Отличительной особенностью регистра накопления является то, что он не предназначен для интерактивного редактирования пользователем.

Разработчик может при необходимости предоставить пользователю возможность редактировать регистр накопления. Но предназначение регистра накопления заключается в том, чтобы его модификация производилась на основе алгоритмов работы других объектов базы данных, а не в результате непосредственных действий пользователя.

Основным назначением регистра накопления является накопление числовой информации в разрезе нескольких *измерений*, которые описываются разработчиком в соответствующем объекте конфигурации Регистр накопления и являются подчиненными объектами конфигурации.

Виды числовой информации, накапливаемой регистром накопления, называются *ресурсами*, также являются подчиненными объектами и описываются в конфигураторе.

Например, регистр накопления может накапливать информацию о количестве и сумме товаров на складах. В этом случае он будет иметь измерения Товар и Склад и ресурсы Количество и Сумма

(см. рис. 6.2).

Изменение состояния регистра накопления происходит, как правило, при проведении документа и заключается в том, что в регистр добавляется некоторое количество записей.

Каждая запись содержит значения измерений, значения приращений ресурсов, ссылку на документ, который вызвал эти изменения (регистратор), и «направление» приращения (приход или расход). Такой набор записей называется *движениями* регистра накопления. Каждому движению регистра накопления всегда должен соответствовать *регистратор* – объект информационной базы (как правило, документ), который произвел эти движения.

Кроме этого, регистр накопления может хранить дополнительную информацию, описывающую каждое движение. Набор такой дополнительной информации задается разработчиком при помощи *реквизитов* объекта конфигурации Регистр накопления.

способы работы с коллекцией

В процессе формирования движений документов, когда в цикле обходили табличные части документов ПриходнаяНакладная и ОказаниеУслуги, мы столкнулись с одним из объектов встроенного языка, который является коллекцией.

Многие объекты встроенного языка являются коллекциями. Коллекция представляет собой совокупность объектов. Существуют общие принципы работы с любой коллекцией.

Во-первых, доступ к каждому объекту коллекции возможен путем перебора элементов коллекции в цикле. Для этого используется конструкция языка Для Каждого Из … Цикл … (листинг 6.3).

листинг 6.3. Перебор элементов коллекции в цикле

```
Для Каждого СтрокаТабличнойЧасти Из ТабличнаяЧасть Цикл

Сообщить(СтрокаТабличнойЧасти.Услуга);

КонецЦикла;
```

В этом примере ТабличнаяЧасть – это коллекция строк табличной части объекта конфигурации. При каждом проходе цикла в переменной СтрокаТабличнойЧасти будет содержаться очередная строка из этой коллекции.

Во-вторых, существует доступ напрямую к элементу коллекции, без перебора коллекции в цикле. Здесь возможны различные комбинации двух обращений.

1. Во встроенном языке бывают именованные коллекции. То есть коллекции, в которых каждый элемент имеет некоторое уникальное имя. В этом случае обращение к элементу коллекции возможно по этому имени (листинг 6.4). **листинг 6.4.** Обращение к элементу коллекции

```
Справочники.Сотрудники;

Справочники["Сотрудники"];
```

В этом примере Справочники – это коллекция менеджеров всех справочников, содержащихся в конфигурации. Так как каждый справочник конфигурации имеет свое уникальное имя, то к

конкретному элементу этой коллекции (к менеджеру конкретного справочника) можно обратиться, указав имя этого справочника: Справочники[«Сотрудники»].

1. Если нет смысла в «персонификации» элементов коллекции (коллекция неименованная), тогда обращение к элементу коллекции возможно по индексу (индекс первого элемента коллекции – ноль), листинг 6.5.

листинг 6.5. Обращение к элементу коллекции по индексу

ТабличнаяЧасть[0];

В этом примере ТабличнаяЧасть – это коллекция строк табличной части объекта конфигурации. И мы обращаемся к первому элементу этой коллекции, указывая его индекс – 0.

Следует отметить, что существуют коллекции, сочетающие оба вида обращений. Например, к коллекции колонок таблицы значений можно обращаться как по именам колонок, так и по индексу.

Что такое отчет

Объект конфигурации Отчет предназначен для описания алгоритмов, при помощи которых пользователь сможет получать необходимые ему выходные данные. Алгоритм формирования выходных данных описывается при помощи визуальных средств или с использованием встроенного языка. В реальной жизни объектам конфигурации Отчет соответствуют всевозможные таблицы выходных данных, сводных данных, диаграммы и пр.

Добавление отчета

В режиме «Конфигуратор»

Теперь у нас все готово для того, чтобы можно было получать выходные данные. Поэтому приступим к созданию отчета, который будет показывать нам приход, расход и остатки материалов (рис. 7.1).

Рис. 7.1. Результат отчета

На этом примере мы покажем, как быстро и легко разработать отчет с использованием только визуальных средств разработки «без единой строчки кода».

Откроем в конфигураторе нашу учебную конфигурацию и добавим новый объект конфигурации Отчет.

Для этого выделим в дереве объектов конфигурации ветвь Отчеты и нажмем кнопку Добавить в командной панели окна конфигурации

(рис. 7.2).

В открывшемся окне редактирования объекта конфигурации на закладке Основные зададим имя отчета – Материалы.

Больше никаких свойств, определяющих представление объекта в интерфейсе приложения, задавать не будем. Вместо них будет использоваться Синоним объекта, который создается автоматически на основании имени объекта.

Создадим основу для построения любого отчета – *схему компоновки данных*. Для этого нажмем кнопку Открыть схему компоновки данных или кнопку открытия со значком лупы (рис. 7.3).

Макет

Так как у отчета, который мы создаем, еще не существует схемы компоновки данных, платформа предложит создать новую схему. Схема компоновки данных с точки зрения конфигурации является макетом, поэтому будет открыт конструктор макета, предлагающий выбрать единст- венный тип макета – Схема компоновки данных

(рис. 7.4).

Рис. 7.4. Создание схемы компоновки данных отчета

Нажмем кнопку Готово.

Схема компоновки данных

Платформа создаст новый макет, содержащий схему компоновки данных, и сразу же откроет конструктор схемы компоновки данных.

Конструктор обладает большим количеством возможностей для визуального проектирования отчетов, но мы сейчас воспользуемся только самыми простыми его возможностями и определим те данные, которые хотим видеть в результате работы нашего отчета.

Набор данных

Добавим новый набор данных – запрос. Для этого нажмем кнопку Добавить или вызовем контекстное меню ветки Наборы данных

(рис. 7.5).

Рис. 7.5. Добавление набора данных в конструкторе схемы компоновки

Текст запроса

Для того чтобы создать текст запроса, запустим конструктор запроса – нажмем кнопку Конструктор запроса (рис. 7.6).

Конструктор запроса – инструмент, созданный для помощи разработчику, позволяющий визуально конструировать запрос. Даже пользователь, не знакомый с языком запросов, может с помощью конструктора создать синтаксически правильный запрос.

В окне конструктора запроса, в списке База данных представлены таблицы для создания запроса. На основе их данных мы имеем возможность построить отчет.

Если раскрыть ветку РегистрыНакопления, то мы увидим, что кроме таблицы регистра ОстаткиМатериалов в этой ветке присутствуют еще несколько *виртуальных таблиц*, которые формирует система

Что такое макет

Объект конфигурации Макет предназначен для хранения различных форм представления данных, которые могут потребоваться каким-либо объектам конфигурации или всему прикладному решению в целом.

Макет может содержать табличный или текстовый документ, двоичные данные, HTML-документ или Active Document, графическую или географическую схему, схему компоновки данных или макет оформления схемы компоновки данных.

Макеты могут существовать как сами по себе (общие макеты), так и быть подчинены какому-либо объекту конфигурации.

Одно из предназначений макета, подчиненного объекту конфигурации и содержащего табличный документ, – создание печатной формы этого объекта. Создание печатной формы заключается в конструировании ее составных частей – именованных областей, из которых затем «собирается» готовая печатная форма.

Порядок заполнения областей данными и вывода их в итоговую форму описывается при помощи встроенного языка. Печатная форма может включать в себя различные графические объекты: картинки, OLE-объекты, диаграммы и т. д.

Помимо создания макета «вручную» конфигуратор предоставляет разработчику возможность воспользоваться специальным инструментом – *конструктором печати*, который берет на себя большинство рутинной работы по созданию макета.

Макет печатной формы

В режиме «Конфигуратор»

Наша цель будет заключаться в создании печатной формы документа Оказание услуги.

Откроем в конфигураторе окно редактирования объекта конфигурации Документ ОказаниеУслуги.

Перейдем на закладку Макеты, нажмем кнопку Конструкторы и запустим конструктор печати (рис. 8.1).

В открывшемся окне конструктора на первом шаге укажем, что будет создана новая команда Печать для формирования печатной формы документа (рис. 8.2).

Нажмем Далее.

На втором шаге нажатием кнопки ›› определим, что все реквизиты нашего документа будут отображены в шапке печатной формы

(рис. 8.3).

Нажмем Далее.

На третьем шаге точно так же определим, что все реквизиты табличной части документа будут отображены в печатной форме

(рис. 8.4).

Нажмем Далее.

На четвертом шаге конструктор предложит сформировать нам подвал (нижнюю часть) печатной формы. Мы не станем ничего указывать (подвал в данном случае использовать не будем), нажмем Далее и перейдем к пятому шагу (рис. 8.5).

Здесь ничего изменять не будем. Тем самым согласимся с тем, что команда для вызова процедуры формирования печатной формы будет помещена в командную панель формы, в раздел Важное.

Нажмем ОК.

В конфигураторе откроется модуль команды Печать, модуль менеджера документа ОказаниеУслуги и макет этого документа

(рис. 8.6).

Рис. 8.6. Макет документа «Оказание услуги»

Заметим, что разработчик может создать макет печатной формы с нуля и для ее вывода создать соответствующую команду и кнопку в форме документа, но в данном случае всю работу сделал за нас конструктор печати:

- Создан макет печатной формы документа ОказаниеУслуги с именем Печать (см. рис. 8.6).
- Создана команда документа ОказаниеУслуги с именем Печать.

В модуль этой команды помещен обработчик, вызывающий процедуру печати документа, выполняющуюся на сервере. Сама процедура печати помещена в модуль менеджера документа ОказаниеУслуги (рис. 8.7).

Рис. 8.7. Структура документа

«Оказание услуги» в дереве объектов конфигурации

- В командную панель формы документа ОказаниеУслуги помещена команда Печать для формирования печатной формы документа

(рис. 8.8).

Причем поскольку команда Печать принадлежит документу ОказаниеУслуги в целом, а не конкретной его форме, эту команду можно будет помещать в любую форму, созданную для документа.

В будущем мы будем самостоятельно создавать процедуры обработчиков команд и размещать соответствующие им кнопки в форме, но пока воспользуемся результатами работы конструктора печати и проверим макет в работе.

Рис. 8.8. Макет документа «Оказание услуги»

Зачем нужен периодический регистр сведений

Начнем мы с того, что обратим ваше внимание на документ Оказание услуги. Как вы помните, в этом документе мы выбираем услугу, которая оказывается, и затем указываем цену.

Очевидно, что в ООО «На все руки мастер» существует перечень услуг, который определяет стоимость каждой услуги. Казалось бы, стоимость услуги является неотъемлемым свойством самой услуги, и поэтому ее следует добавить в качестве реквизита справочника Номенклатура.

Однако стоимость услуг имеет особенность меняться со временем. И может сложиться такая ситуация, когда нам потребуется внести изменения или уточнения в один из ранее проведенных документов Оказание услуги. В этом случае мы не сможем получить правильную стоимость услуги, поскольку в реквизите справочника будет храниться последнее введенное значение.

Кроме этого, не исключено, что руководство ООО «На все руки мастер» пожелает видеть зависимость прибыли предприятия от изменения стоимости оказываемых услуг. И тогда просто необходимо будет иметь возможность анализировать изменение стоимости услуг во времени.

Поэтому для хранения стоимости услуг мы используем новый пока еще для нас объект – *Регистр сведений*.

Что такое регистр сведений

Объект конфигурации Регистр сведений предназначен для описания структуры хранения данных в разрезе нескольких измерений. На основе объекта конфигурации Регистр сведений платформа создает в базе данных таблицу, в которой может храниться произвольная информация, «привязанная» к набору измерений (рис. 9.1).

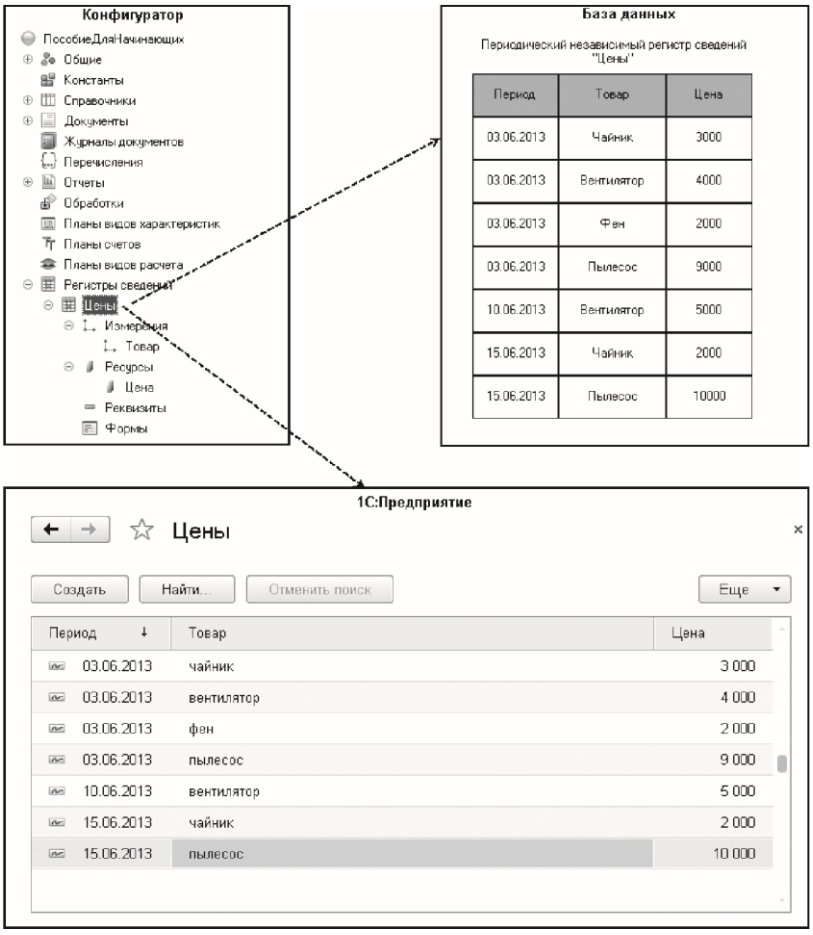


Рис. 9.1. Независимый периодический регистр сведений «Цены» в конфигураторе и в базе данных

Принципиальное отличие регистра сведений от регистра накопления заключается в том, что каждое движение регистра сведений устанавливает новое значение ресурса, в то время как

движение регистра накопления изменяет существующее значение ресурса. По этой причине регистр сведений может хранить любые данные (а не только числовые, как регистр накопления).

Следующей важной особенностью регистра сведений является его способность (при необходимости) хранить данные с привязкой ко времени. Благодаря этому регистр сведений может хранить не только актуальные значения данных, но и историю их изменения во времени. Регистр сведений, использующий привязку ко времени, называют *периодическим регистром сведений*.

Периодичность регистра сведений можно определить одним из следующих значений:

- в пределах секунды;
- в пределах дня;
- в пределах месяца;
- в пределах квартала;
- в пределах года;
- в пределах регистратора (если установлен режим записи Подчинение регистратору).

Периодический регистр сведений всегда содержит служебное поле Период, добавляемое системой автоматически. Оно имеет тип Дата и служит для указания факта принадлежности записи к какому-либо периоду. При записи данных в регистр платформа всегда приводит значение этого поля к началу того периода, в который он попадает.

Например, если в регистр сведений с периодичностью в пределах месяца записать данные, в которых период указан как 08.04.2013, то регистр сохранит эти данные со значением периода, равным 01.04.2013.

Как и для других регистров, система контролирует уникальность записей для регистра сведений. Однако если для прочих регистров уникальным идентификатором записи является регистратор и номер строки, то для регистра сведений применяется другой принцип формирования ключевого значения.

Ключом записи, однозначно идентифицирующим запись, является в данном случае совокупность значений измерений регистра и периода (в случае, если регистр сведений периодический). Например, для периодического регистра сведений с измерением Товар и ресурсом Цена (см. рис. 9.1) ключом записи будет набор значений полей Период и Товар. Регистр сведений не может содержать несколько записей с одинаковыми ключами.

Если продолжать сравнение с регистром накопления, то можно сказать, что регистр сведений предоставляет больше свободы в редактировании хранимых данных. Наряду с возможностью использования в режиме подчинения регистратору (когда записи регистра сведений «привязаны» к документу-регистратору) регистр сведений может применяться и в независимом режиме, в котором пользователю предоставляется полная свобода интерактивной работы с данными регистра. Регистр сведений, не использующий подчинение регистратору, называют *независимым регистром сведений*.

Что такое перечисление

Объект конфигурации *Перечисление* предназначен для описания структуры хранения постоянных наборов значений, не изменяемых в процессе работы конфигурации. На основе

объекта конфигурации Перечисление платформа создает в базе данных таблицу, в которой может храниться набор некоторых постоянных значений.

В реальной жизни этому объекту может соответствовать, например, перечисление вариантов указания цены («включая НДС», «без НДС»). Набор всех возможных значений, которые содержит перечисление, задается при конфигурировании системы, и пользователь не может изменять их, удалять или добавлять новые.

Из этого следует важная особенность перечисления: значения перечисления не «обезличены» для конфигурации, на них могут опираться алгоритмы работы программы.

Добавление перечисления

В режиме «Конфигуратор»

Откроем конфигуратор и создадим сначала новый объект конфигурации Перечисление с именем ВидыНоменклатуры.

На закладке Данные добавим два значения перечисления: Материал и Услуга.

Для этого нажмем кнопку Добавить над списком значений перечисления (рис. 10.1).

Привязка номенклатуры к значениям перечисления «ВидНоменклатуры»

Чтобы привязать номенклатуру к значениям перечисления, мы сделаем следующее:

- в режиме Конфигуратор создадим у справочника Номенклатура реквизит, который будет хранить значение перечисления;
- в режиме 1С:Предприятие проставим нужные значения этого рек-

визита для всех элементов справочника Номенклатура.

В режиме «Конфигуратор»

Добавим в справочник Номенклатура новый реквизит ВидНоменклатуры с типом ПеречислениеСсылка.ВидыНоменклатуры.

Для этого откроем окно редактирования объекта конфигурации Справочник Номенклатура и на закладке Данные нажмем кнопку Добавить над списком реквизитов справочника (рис. 10.2).

Рис. 10.2. Данные справочника «Номенклатура»

В режиме «1С:Предприятие»

После этого запустим «1С:Предприятие» в режиме отладки.

В режиме 1С:Предприятие зададим для каждого элемента справочника Номенклатура соответствующее значение реквизита Вид номенклатуры (рис. 10.3).

Теперь посмотрим, как можно применить новые данные, полученные благодаря использованию перечисления ВидыНоменклатуры.

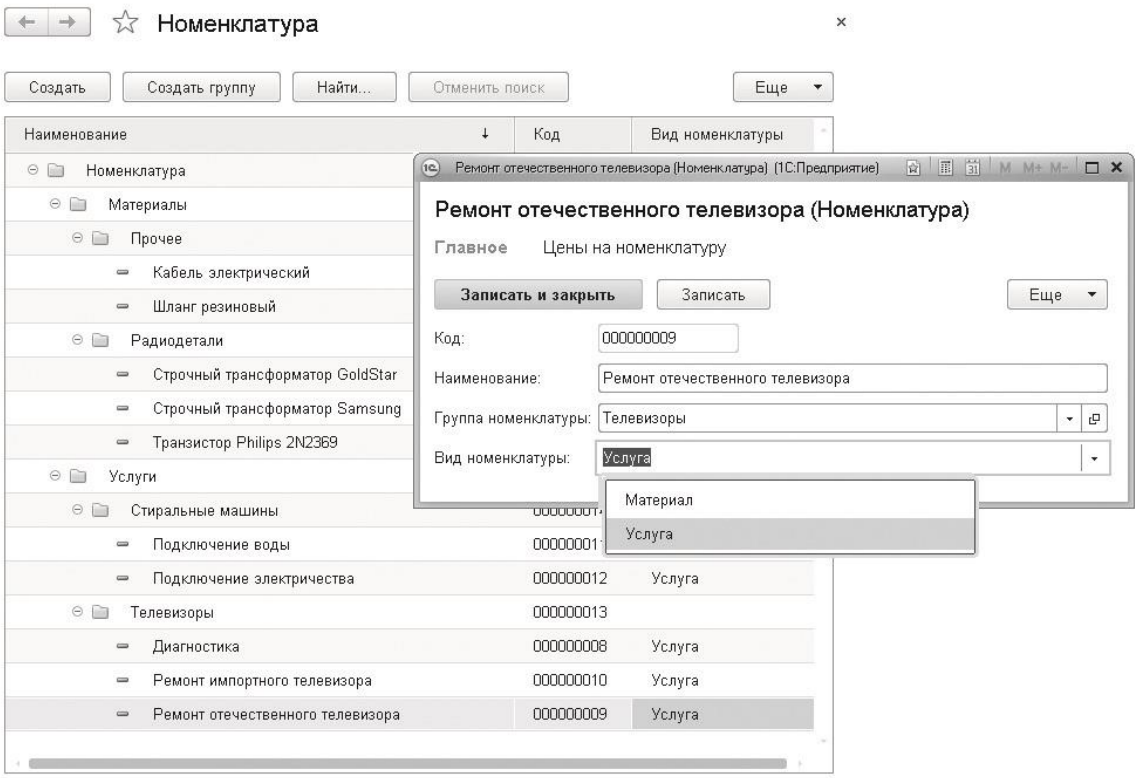


Рис. 10.3. Данные справочника «Номенклатура»

Произвольное представление номенклатуры

Теперь, используя реквизит Вид номенклатуры, зададим произвольное представление номенклатуры в интерфейсе «1С:Предприятия».

Представление номенклатуры используется везде, где отображаются поля, ссылающиеся на элементы справочника Номенклатура. Такие поля мы видим в табличной части наших документов, в регистре сведений, регистре накопления и т. д.

Стандартное представление номенклатуры (как и любого другого элемента справочника) определяется свойством справочника Основное представление. По умолчанию это свойство установлено в значение В виде наименования (рис. 10.4).

Поэтому, например, в табличной части документов в колонке Номенклатура мы видим не ссылку на номенклатуру, а ее наименование

(рис. 10.5).

Рис. 10.5. Документ «Оказание услуги»

Было бы удобно, чтобы при отображении ссылок на номенклатуру в интерфейсе «1С:Предприятия» рядом с наименованием номенклатуры показывался бы и ее вид (материал или услуга).

Зачем нужно проведение документа по нескольким регистрам

До сих пор мы с вами учитывали только количественное движение материалов в ООО «На все руки мастер». Для этих целей мы создали регистр накопления ОстаткиМатериалов.

Однако как вы, наверное, догадываетесь, одного только количественного учета совершенно недостаточно для нужд нашего предприятия. Очевидно, что необходимо также знать, какие денежные средства были затрачены на приобретение тех или иных материалов и каковы материальные запасы ООО «На все руки мастер» в денежном выражении.

После того как мы начали автоматизировать наше предприятие, руководство ООО «На все руки мастер» высказало пожелание, чтобы весь суммовой учет материалов велся бы теперь по средней стоимости.

То есть при закупке материалов они должны учитываться в ценах приобретения, а при расходе – по средней стоимости, которая рассчитывается исходя из общей суммы закупок данного материала и общего количества этого материала, находящегося в ООО «На все руки мастер».

Поскольку подобная информация имеет совершенно другую структуру, нежели количественный учет, для хранения данных об общей стоимости тех или иных материалов мы будем использовать еще один регистр накопления СтоимостьМатериалов.

Таким образом, документы ПриходнаяНакладная и ОказаниеУслуги должны будут создавать движения не только в регистре ОстаткиМатериалов, но одновременно и в регистре СтоимостьМатериалов, отражая изменения суммового учета.

Добавление еще одного регистра накопления

В режиме «Конфигуратор»

Регистр СтоимостьМатериалов совсем не сложен, поэтому мы не будем подробно останавливаться на его создании.

Создадим новый объект конфигурации Регистр накопления с именем СтоимостьМатериалов.

Расширенное представление списка зададим как Движения по регистру Стоимость материалов. Этот заголовок будет отображаться в окне списка записей регистра.

На закладке Подсистемы отметим, что этот регистр будет отображаться в подсистемах Бухгалтерия, Учет материалов и Оказание услуг.

На закладке Данные создадим для регистра одно измерение – Материал типа СправочникСсылка.Номенклатура и один ресурс – Стоимость типа Число длиной 15 и точностью 2.

После создания регистр СтоимостьМатериалов должен выглядеть в дереве конфигурации следующим образом

Теперь отредактируем командный интерфейс, чтобы в разделах Бухгалтерия, Оказание услуг и Учет материалов была доступна команда для просмотра нашего регистра накопления.

В дереве объектов конфигурации выделим ветвь Подсистемы, вызовем ее контекстное меню и выберем пункт Все подсистемы.

В открывшемся окне слева в списке Подсистемы выделим подсистему Бухгалтерия.

Справа в списке Командный интерфейс отразятся все команды выбранной подсистемы.

В группе Панель навигации.Обычное включим видимость у команды

Стоимость материалов и мышью перетащим ее в группу Панель

Рис. 11.2. Настройка командного интерфейса подсистем

Аналогично, выделив подсистемы ОказаниеУслуг и УчетМатериалов, в группе Панель навигации.Обычное включим видимость у команды Стоимость материалов и перенесем ее в группу Панель навигации.См. также.

Теперь мы можем приступить к внесению изменений в процедуры проведения наших документов.

Начнем с самого простого – документа Приходная накладная.

Проведение приходной накладной по двум регистрам

В режиме «Конфигуратор»

Изменение процедуры проведения

Откроем в конфигураторе окно редактирования объекта конфигурации Документ ПриходнаяНакладная и перейдем на закладку Движения.

В списке регистров отметим, что документ будет создавать теперь движения и по регистру СтоимостьМатериалов (рис. 11.3).

в регистре «Стоимость материалов»

Нажмем кнопку Конструктор движений. На вопрос системы о замещении процедуры проведения документа на новую, сформированную конструктором, ответим утвердительно. Мы ничего не потеряем, так как конструктор в обработке проведения сформирует движения уже по двум регистрам, а в прежнюю процедуру проведения мы никаких изменений не вносили.

В открывшемся окне конструктора движений мы увидим, что для регистра ОстаткиМатериалов все поля конструктора уже содержат информацию, которую мы задавали ранее при формировании движений для этого регистра. Над списком Регистры нажмем кнопку Добавить и добавим еще один регистр СтоимостьМатериалов (рис. 11.4).

Рис. 11.4. Конструктор движений регистров

Зачем нужно создавать еще один регистр

Продолжим рассматривать работу нашего документа ОказаниеУслуги.

До сих пор мы создавали в регистрах накопления движения только для строк документа, которые содержат материалы. Услуги, содержащиеся в документе, мы никак не учитывали.

Дело в том, что при учете услуг важны совершенно другие критерии, нежели при учете материалов.

Прежде всего, бессмысленно говорить о том, сколько услуг было и сколько их осталось, важна только сумма и количество услуг, которые были оказаны за определенный промежуток времени.

Кроме этого, интересны следующие моменты:

- какие именно услуги были оказаны (чтобы составить рейтинг услуг);

- какому именно клиенту оказывались услуги (чтобы, например, предоставить ему скидку от объема оплаченных ранее услуг);
- какой мастер предоставлял услуги (чтобы начислить ему заработную плату).

Очевидно, что существующие регистры накопления совершенно не подходят для решения таких задач.

Поэтому мы создадим еще одно хранилище данных, которое будет использоваться в нашей программе, – оборотный регистр накопления Продажи.

Что такое оборотный регистр накопления

Когда мы создавали регистры ОстаткиМатериалов и СтоимостьМатериалов, мы специально не останавливались на видах регистров накопления, которые существуют в системе «1С:Предприятие». Сейчас пришло время сказать об этом несколько слов.

Регистры накопления могут быть *регистрами остатков* и *регистрами оборотов*.

Существующие в нашей учебной конфигурации регистры ОстаткиМатериалов и СтоимостьМатериалов являются регистрами остатков.

Если вы помните, при создании отчета Материалы в конструкторе запроса мы видели, что для таких регистров система создает три виртуальные таблицы: таблицу остатков, таблицу оборотов и совокупную таблицу остатков и оборотов.

Оборотный регистр накопления очень похож на знакомый уже нам регистр остатков, но для него понятие «остаток» не имеет смысла. Оборотный регистр накапливает только обороты, остатки ему безразличны. Поэтому единственной виртуальной таблицей, которую будет создавать система для такого регистра, будет таблица оборотов. В остальном оборотный регистр ничем не отличается от регистра остатков.

Следует сказать об одной особенности конструирования регистров накопления, напрямую связанной с возможностью получения остатков. При создании оборотного регистра накопления нет особой сложности в определении того, какие именно данные должны являться измерениями регистра – мы можем назначить в качестве его измерений любые нужные нам данные.

Совсем иная ситуация в случае регистра накопления, поддерживающего накопление остатков. Для него выбор измерений должен выполняться исходя из того, что движения регистра могут быть осуществлены в две стороны: приход и расход. Таким образом, в качестве измерений нужно выбирать те данные, по которым движения точно будут осуществляться как в одну, так и в другую сторону.

Например, если ведется учет материалов в разрезах номенклатуры и склада, очевидно, что и номенклатура, и склад могут быть измерениями, поскольку как приход, так и расход материалов всегда будут осуществляться с указанием конкретной номенклатуры и конкретного склада. Если же в этой ситуации появляется желание отразить учет материалов еще и в разрезе поставщика, то здесь уже нужно исходить из конкретной схемы учета, принятой на предприятии.

Скорее всего, при поступлении материалов поставщик будет указан, а вот при расходовании материалов с большой долей вероятности поставщик указываться не будет. В большинстве случаев это совершенно лишняя информация.

Значит, поставщика следует добавить не как измерение, а как реквизит регистра накопления.

Если же при расходе материалов поставщик будет указываться наверняка, имеет смысл добавить поставщика в измерения регистра.

Иными словами, по каждому из измерений регистра накопления остатков ресурсы обязательно должны изменяться в обе стороны: приход и расход. Не должно существовать таких измерений, по которым осуществляется только приход или только расход.

Нарушение этого принципа построения регистров накопления будет вести к непроизводительному использованию ресурсов системы и как следствие к замедлению работы и падению производительности.

Для реквизитов же регистра этот принцип не важен. По реквизитам регистра ресурсы могут только приходоваться или только расходоваться.

Добавление оборотного регистра накопления

В режиме «Конфигуратор»

Теперь, когда мы знаем практически все о регистрах накопления, откроем конфигуратор и создадим новый объект конфигурации Регистр накопления.

Назовем его Продажи и определим вид регистра – Обороты.

Кроме этого, зададим Расширенное представление списка как Движения по регистру Продажи. Этот заголовок будет отображаться в окне списка записей регистра (рис. 12.1).

На закладке Подсистемы отметим, что этот регистр будет отображаться в подсистемах Бухгалтерия, Учет материалов и Оказание услуг.

На закладке Данные создадим измерения регистра:

- Номенклатура, тип СправочникСсылка.Номенклатура;
- Клиент, тип СправочникСсылка.Клиенты;
- Мастер, тип СправочникСсылка.Сотрудники.

Рис. 12.1. Создание оборотного регистра накопления

У регистра будет три ресурса:

- Количество, тип Число, длина 15, точность 3;
- Выручка, тип Число, длина 15, точность 2;
- Стоимость, тип Число, длина 15, точность 2.

После создания регистр Продажи должен выглядеть в дереве конфи-

гурации следующим образом

(рис. 12.2).

Рис. 12.2. Оборотный регистр накопления «Продажи»

Теперь отредактируем командный интерфейс, чтобы в подсистемах Бухгалтерия, Оказание услуг и Учет материалов была доступна команда для просмотра нашего оборотного регистра накопления.

В дереве объектов конфигурации выделим ветвь Подсистемы, вызовем ее контекстное меню и выберем пункт Все подсистемы.

В открывшемся окне слева в списке Подсистемы выделим подсистему Бухгалтерия.

Справа в списке Командный интерфейс отразятся все команды выбранной подсистемы.

В группе Панель навигации.Обычное включим видимость у команды Продажи и мышью перетащим ее в группу Панель навигации.См. также.

Аналогично, выделив подсистемы ОказаниеУслуг и УчетМатериалов, в группе Панель навигации.Обычное включим видимость у команды Продажи и перенесем ее в группу Панель навигации.См. также.

Настало время, чтобы познакомиться с одним важным инструментом платформы «1С:Предприятие» – *системой компоновки данных*. На этом занятии мы рассмотрим построение нескольких отчетов, которые будут использоваться в нашей конфигурации, и на их примере объясним основные возможности системы компоновки данных.

Любой отчет, как правило, подразумевает получение сложной выборки данных, сгруппированных и отсортированных определенным образом. Система компоновки данных представляет собой мощный и гибкий механизм, позволяющий выполнить все необходимые действия – от получения данных из различных источников до представления этих данных в виде, удобном для пользователя.

Чаще всего исходные данные, необходимые для отчета, находятся в базе данных. Для того чтобы указать системе компоновки данных, какая информация и откуда должна быть получена, используется язык запросов системы «1С:Предприятие».

На этапе разработки отчета можно задать стандартные настройки отчета для того, чтобы пользователь мог сразу же запустить отчет на выполнение. В то же время пользователь может самостоятельно изменить настройки отчета и выполнить его. При этом система компоновки данных сгенерирует другой запрос и другим образом представит конечные данные – в соответствии с новыми настройками, заданными пользователем.

В начале этого занятия мы познакомимся с общими сведениями о языке запросов системы «1С:Предприятие» и о системе компоновки данных.

Затем на примерах создания конкретных отчетов мы научимся использовать систему компоновки данных для решения различных практических задач.

Способы доступа к данным

Система «1С:Предприятие» поддерживает два способа доступа к данным, хранящимся в базе данных:

объектный (для чтения и записи); табличный (для чтения).

Объектный способ доступа к данным реализован посредством использования объектов встроенного языка.

С некоторыми из этих объектов мы уже познакомились на предыдущих занятиях.

Важной особенностью объектного способа доступа к данным является то, что, обращаясь к какому-либо объекту встроенного языка, мы обращаемся к некоторой совокупности данных, находящихся в базе данных, как к единому целому.

Например, объект ДокументОбъект.ОказаниеУслуги будет содержать значения всех реквизитов документа Оказание услуги и всех его табличных частей.

Объектная техника обеспечивает сохранение целостности объектов, кеширование объектов, вызов соответствующих обработчиков событий и т. д.

Табличный доступ к данным в «1С:Предприятии» реализован с помощью запросов к базе данных, которые составляются на *языке запросов*.

В этой технике разработчик получает возможность оперировать отдельными полями таблиц базы данных, в которых хранятся те или иные данные.

Табличная техника предназначена для получения информации из базы данных по некоторым условиям (отбор, группировка, сортировка, объединение нескольких выборок, расчет итогов и т. д.). Табличная техника оптимизирована для обработки больших объемов информации, расположенной в базе данных, и получения данных, отвечающих заданным критериям.

Работа с запросами

Для работы с запросами используется объект встроенного языка Запрос. Он позволяет получать информацию, хранящуюся в полях базы данных, в виде выборки, сформированной по заданным правилам. **Источники данных запросов**

Исходную информацию запрос получает из набора таблиц. Эти таблицы представляют разработчику данные реальных таблиц базы данных в удобном для анализа виде.

Все таблицы, которыми оперирует язык запросов, можно разделить на две большие группы: реальные таблицы и виртуальные таблицы

(рис. 13.1).

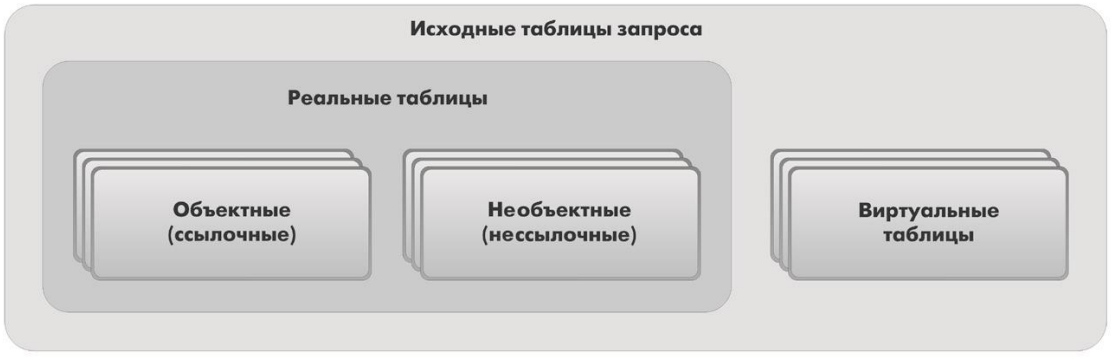


Рис. 13.1. Таблицы запросов

Посмотреть состав таблиц, доступных для запроса, и их описание можно в синтакс-помощнике в разделе Работа с запросами Таблицы запросов.

Отличительной особенностью реальных таблиц является то, что они содержат данные какой-либо одной реальной таблицы, хранящейся в базе данных.

Например, реальной является таблица Справочник.Клиенты, соответствующая справочнику Клиенты, или таблица РегистрНакопления.ОстаткиМатериалов, соответствующая регистру накопления ОстаткиМатериалов.

Виртуальные таблицы формируются в основном из данных нескольких таблиц базы данных.

Например, виртуальной является таблица РегистрНакопления.ОстаткиМатериалов.ОстаткиИОбороты, формируемая из нескольких таблиц регистра накопления Остатки Материалов.

Иногда виртуальные таблицы могут формироваться и из одной реальной таблицы (например, виртуальная таблица Цены.СрезПоследних формируется на основе таблицы регистра сведений Цены).

Однако общим для всех виртуальных таблиц является то, что им можно задать ряд параметров, определяющих, какие данные будут включены в эти виртуальные таблицы. Набор таких параметров может быть различным для разных виртуальных таблиц и определяется данными, хранящимися в исходных таблицах базы данных.

Реальные таблицы подразделяются на *объектные* (ссылочные) и *необъектные* (нессылочные).

В объектных (ссылочных) таблицах представлена информация ссылочных типов данных (справочники, документы, планы видов характеристик и т. д.). А в необъектных (нессылочных) – всех остальных типов данных (константы, регистры и т. д.).

Отличительной особенностью объектных (ссылочных) таблиц является то, что они включают в себя поле Ссылка, содержащее ссылку на текущую запись. Кроме этого, для таких таблиц возможно получение пользовательского представления объекта. Эти таблицы могут быть иерархическими, и поля таких таблиц могут содержать вложенные таблицы (табличные части).

Лекция 10

Язык запросов

Алгоритм, по которому данные будут выбраны из исходных таблиц запроса, описывается на специальном языке – *языке запросов*.

Текст запроса может состоять из нескольких частей:

- описание запроса,
- объединение запросов,
- упорядочивание результатов,
- автоупорядочивание, описание итогов.

Обязательной частью запроса является только первая – описание запроса. Все остальные присутствуют по необходимости.

Описание запроса определяет источники данных, поля выборки, группировки и т. д.

Объединение запросов определяет, как будут объединены результаты выполнения нескольких запросов.

Упорядочивание результатов определяет условия упорядочивания строк результата запроса.

Автоупорядочивание позволяет включить режим автоматического упорядочивания строк результата запроса.

Описание итогов определяет, какие итоги необходимо рассчитывать в запросе и каким образом группировать результат.

Следует заметить, что в случае, когда язык запросов используется для описания источников данных в системе компоновки данных, секция описания итогов языка запросов не используется. Это связано с тем, что система компоновки данных самостоятельно рассчитывает итоги на основании тех настроек, которые сделаны разработчиком или пользователем.

Применение различных синтаксических конструкций языка запросов подробно описано во встроенной справке в режиме Конфигуратор: Справка Содержание справки 1С:Предприятие Встроенный язык Работа с запросами, а также в документации «1С:Предприятие 8.3. Руководство разработчика», глава 8 «Работа с запросами».

Детальнее с языком запросов мы познакомимся далее, в процессе создания конкретных отчетов.

Мы не будем писать запросы руками. Для большинства отчетов, разрабатываемых с помощью системы компоновки данных, запрос можно создать при помощи конструктора запросов.

Поэтому наша задача на этом занятии – научиться читать и понимать тексты этих запросов, чтобы в дальнейшем иметь возможность изменять их.

Выбор данных из двух таблиц

Отчет Рейтинг услуг будет содержать информацию о том, выполнение каких услуг принесло ООО «На все руки мастер» наибольшую прибыль в указанном периоде (рис. 13.18).

На примере отчета Рейтинг услуг мы проиллюстрируем, как отбирать данные в некотором периоде, как задавать параметры запроса, как использовать в запросе данные из нескольких таблиц и как включать в результат запроса все данные одного из источников.

Также мы узнаем, как работать с параметрами системы компоновки данных, как использовать стандартные даты, и познакомимся с быстрыми пользовательскими настройками отчетов.

Кроме этого, мы научимся более детально настраивать отбор и условное оформление в отчетах.

Запрос для набора данных

Левое соединение двух таблиц

В качестве источника данных для запроса выберем объектную (ссылочную) таблицу Номенклатура и виртуальную таблицу регистра накопления Продажи.Обороты.

Чтобы исключить неоднозначность имен в запросе, переименуем таблицу Номенклатура в спрНоменклатура.

Для этого выделим ее в списке Таблицы, вызовем ее контекстное меню и выберем пункт Переименовать таблицу (рис. 13.19).

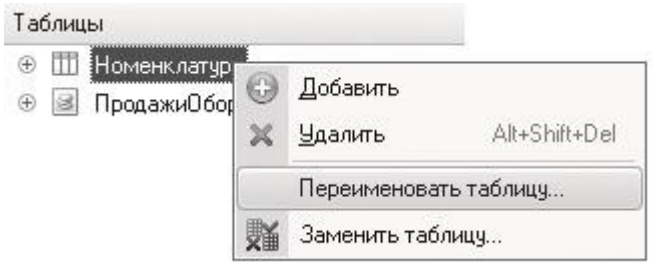


Рис. 13.19. Переименование таблицы в запросе

В список полей перенесем поля СпрНоменклатура.Ссылка и ПродажиОбороты.ВыручкаОборот из этих таблиц (рис. 13.20).

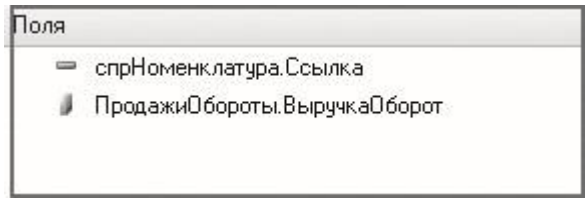


Рис. 13.20. Выбранные поля Перейдем на закладку Связи.

Так как в запросе теперь участвуют несколько таблиц, требуется определить связь между ними.

По умолчанию платформой уже будет создана связь по полю Номенклатура. То есть значение измерения Номенклатура регистра Продажи должно быть равно ссылке на элемент справочника Номенклатура.

Но нам нужно снять флажок Все у таблицы ПродажиОбороты и установить его у таблицы спрНоменклатура.

Тем самым мы задаем тип связи как Левое соединение, то есть в результат запроса будут включены все записи справочника Номенклатура и те записи регистра Продажи, которые удовлетворяют условию связи по полю Номенклатура.

Таким образом, в результате запроса будут присутствовать все услуги, и для некоторых из них будут указаны обороты выручки. Для тех услуг, которые не производились в выбранном периоде, не будет указано ничего.

Описанную связь двух таблиц схематично можно представить следующим примером (рис. 13.21).

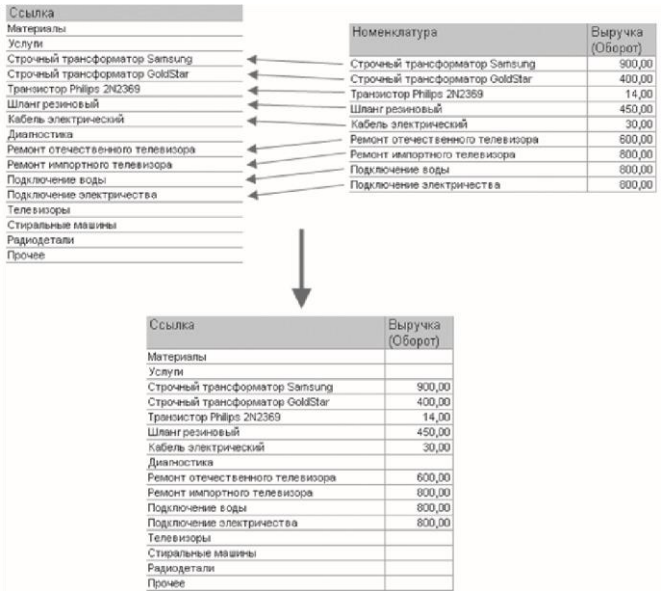


Рис. 13.21. Связь записей таблиц в запросе

В результате описанных выше действий закладка Связи будет иметь следующий вид (рис. 13.22).

Рис. 13.22. Определение связи между таблицами

Условие отбора записей

Перейдем на закладку Условия и установим отбор, чтобы группы справочника Номенклатура не попадали в отчет.

Для этого раскроем таблицу спрНоменклатура, перетащим мышью поле ЭтоГруппа в список условий, установим флажок Произвольное и напишем в поле Условие следующий текст (листинг 13.5).

листинг 13.5. Условие запроса

```
спрНоменклатура.ЭтоГруппа = ЛОЖЬ
```

Тем самым мы указали, что из базы данных нужно выбрать только те записи справочника Номенклатура, которые не являются группами.

Работу этого условия можно проиллюстрировать на следующем примере. Слева – исходная таблица справочника Номенклатура, а справа – записи, которые будут выбраны из этой таблицы (рис. 13.23).

Рис. 13.23. Отбор записей номенклатуры в запросе

Вторым условием должно быть то, что выбранный элемент является услугой. Это Простое условие. Чтобы его создать, перетащим мышью поле ВидНоменклатуры в список условий.

Платформа автоматически сформирует условие, согласно которому вид номенклатуры должен быть равен значению параметра ВидНоменклатуры.

В дальнейшем перед выполнением запроса мы передадим в параметр ВидНоменклатуры значение перечисления – Услуга.

Работу этого условия тоже можно проиллюстрировать на примере. Слева – записи справочника Номенклатура, выбранные согласно первому условию. Справа – только те записи, которые являются услугами (рис. 13.24).

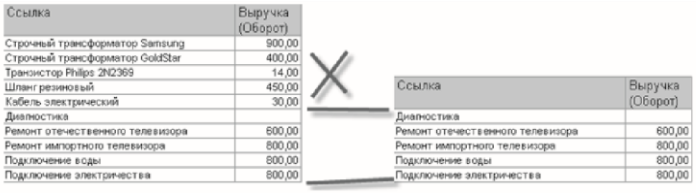


Рис. 13.24. Отбор записей номенклатуры в запросе

В результате закладка Условия примет вид (рис. 13.25).

Рис. 13.25. Создание условия запроса

Вывод данных по всем дням в выбранном периоде

Следующий отчет, который мы добавим, будет называться Выручка мастеров.

Он будет содержать информацию о том, какая выручка была получена ООО «На все руки мастер» благодаря работе каждого из мастеров, с детализацией по всем дням в выбранном периоде и разворотом по клиентам, обслуженным в каждый из дней (рис. 13.57).

На примере этого отчета мы проиллюстрируем, как строить многоуровневые группировки в запросе и как обходить все даты в выбранном периоде.

Также продемонстрируем настройку отдельных элементов структуры отчета, научимся выводить данные в диаграмму и создавать несколько вариантов отчета в конфигураторе.

Добавим новый объект конфигурации Отчет. Назовем его ВыручкаМастеров и запустим конструктор схемы компоновки данных.

Добавим новый Набор данных – запрос и вызовем конструктор запроса.

В качестве источника данных для запроса выберем виртуальную таблицу регистра накопления Продажи.Обороты.

Запрос для набора данных

Параметры виртуальной таблицы

Зададим один из параметров этой виртуальной таблицы – Периодичность.

Для этого перейдем в поле Таблицы, выделим таблицу и нажмем кнопку Параметры виртуальной таблицы (рис. 13.58).

Рис. 13.58. Изменение параметров виртуальной таблицы

В открывшемся окне параметров зададим значение параметра Периодичность – День (рис. 13.59).

Рис. 13.59. Параметры виртуальной таблицы

Нажмем ОК. После этого выберем из таблицы следующие поля

(рис. 13.60):

- ПродажиОбороты.Мастер,
- ПродажиОбороты.Период,
- ПродажиОбороты.Клиент,
- ПродажиОбороты.ВыручкаОборот.

Рис. 13.60. Выбранные поля

Теперь перейдем на закладку Объединения/Псевдонимы и зададим псевдоним Выручка для поля ПродажиОбороты.ВыручкаОборот

Рис. 13.61. Объединения/Псевдонимы

Анализ текста запроса

Нажмем ОК и рассмотрим текст запроса, сформированный конструктором (листинг 13.10).

листинг 13.10. Текст запроса

```
ВЫБРАТЬ

ПродажиОбороты.Мастер,

ПродажиОбороты.Период,

ПродажиОбороты.Клиент,

ПродажиОбороты.ВыручкаОборот КАК Выручка

ИЗ

РегистрНакопления.Продажи.Обороты( , , День, ) КАК ПродажиОбороты
```

В части описания запроса обратите внимание, что у источника данных задана периодичность выбираемых данных – День (листинг 13.11).

листинг 13.11. Задание периодичности виртуальной таблицы

```
ИЗ

РегистрНакопления.Продажи.Обороты( , , День, ) КАК ПродажиОбороты
```

Именно благодаря этому у нас появляется возможность описать среди выбранных полей поле Период. **Ресурсы**

Теперь перейдем к редактированию схемы компоновки данных.

На закладке Ресурсы нажмем кнопку << и убедимся, что конструктор выбрал единственный имеющийся у нас ресурс – Выручка.

Параметры

На закладке Параметры выполним те же действия, что и при создании предыдущего отчета.

Для параметров НачалоПериода и КонецПериода в поле Тип зададим состав даты – Дата.

Для параметра КонецПериода зададим Выражение (листинг 13.12).

листинг 13.12. Выражение для расчета значения параметра «КонецПериода»

```
КонецПериода(&КонецПериода, "День")
```

В результате перечисленных действий параметры компоновки данных будут иметь следующий вид (рис. 13.62).

Рис. 13.62. Параметры компоновки данных

Диаграмма

Диаграмма предназначена для размещения в таблицах и формах диаграмм и графиков различного вида.

Логически диаграмма является совокупностью точек, серий и значений серий в точке (рис. 13.76).

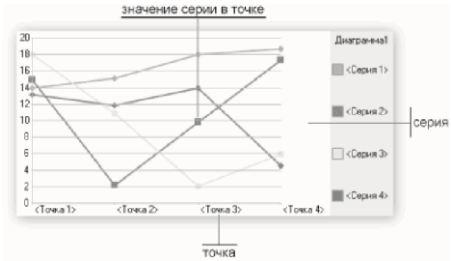


Рис. 13.76. Пример диаграммы

Как правило, в качестве *точек* используются моменты или объекты, для которых мы получаем значения характеристик, а в качестве *серий* – характеристики, значения которых нас интересуют. На пересечении серии и точки находится *значение* диаграммы.

Например, диаграмма продаж видов номенклатуры по месяцам будет состоять из точек – месяцев, серий – видов номенклатуры и значений – оборотов продаж.

Диаграмма как объект встроенного языка имеет три области, которые позволяют управлять оформлением диаграммы: область построения, область заголовка и область легенды (рис. 13.77).

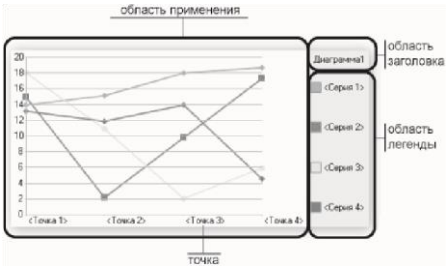


Рис. 13.77. Области диаграммы

Диаграмма может быть вставлена в структуру отчета как отдельный элемент. В следующем варианте настроек отчета ВыручкаМастеров мы будем использовать диаграмму в структуре настроек схемы компоновки данных.

Получение актуальных значений из периодического регистра сведений

Следующий отчет – Перечень услуг – будет содержать информацию о том, какие услуги и по какой цене оказывает ООО «На все руки мастер» (рис. 13.85).

На его примере мы познакомимся с возможностью получения последних значений из периодического регистра сведений и с возможностью вывода иерархических справочников.

Запрос для набора данных

В качестве источника данных для запроса выберем объектную (ссылочную) таблицу справочника Номенклатура и виртуальную таблицу регистра сведений Цены.СрезПоследних.

Для того чтобы исключить неоднозначность имен в запросе, переименуем таблицу Номенклатура в СпрНоменклатура. Для этого выделим ее в списке Таблицы, вызовем ее контекстное меню и выберем пункт Переименовать таблицу.

Параметры виртуальной таблицы

Вызовем диалог ввода параметров виртуальной таблицы

Цены.СрезПоследних и укажем, что период будет передан в параметре ДатаОтчета. Для этого выделим эту таблицу в списке Таблицы и нажмем кнопку Параметры виртуальной таблицы (рис. 13.86).

Затем выберем из таблиц следующие поля (рис. 13.87):

- СпрНоменклатура.Родитель,
- СпрНоменклатура.Ссылка,
- ЦеныСрезПоследних.Цена.

Левое соединение таблиц

Перейдем на закладку Связи. Мы видим, что платформа автоматически добавила условие связи таблиц, при котором значение измерения Номенклатура регистра сведений должно быть равно ссылке на элемент справочника Номенклатура. Нас это устраивает.

Снимем флажок Все у таблицы регистра и установим его у таблицы справочника, тем самым установив вид связи как левое соединение для таблицы справочника (рис. 13.88).

На закладке Условия зададим условие выбора элементов справочника Номенклатура – выбираемые элементы должны соответствовать виду номенклатуры, переданному в параметре запроса ВидНоменклатуры (рис. 13.89).

Псевдонимы полей

На закладке Объединения/Псевдонимы укажем, что поле Родитель будет иметь псевдоним ГруппаУслуг, а поле Ссылка – Услуга (рис. 13.90).

На этом создание запроса завершено, нажмем ОК.

Использование вычисляемого поля в отчете

Следующий отчет – Рейтинг клиентов – будет показывать в графическом виде, каков доход от оказания услуг каждому из клиентов за все время работы ООО «На все руки мастер» (рис. 13.99).

На его примере мы продемонстрируем возможность использования вычисляемого поля и вывод результата в виде круговой диаграммы и в виде гистограммы.

Как понять работу кода на встроенном языке

На предыдущем занятии мы писали код обработчика события МатериалыКоличествоПриИзменении (листинг 4.1) и кратко объясняли смысл написанного.

Теперь мы покажем два способа, как самому разобраться с множеством незнакомых свойств и методов объектов конфигурации, чтобы в будущем самостоятельно изучать фрагменты кода или создавать свои собственные процедуры на встроенном языке.

Синтакс-помощник – инструмент, созданный для помощи разработчику, содержащий описание всех программных объектов, которые использует система, их методов, свойств, событий и пр.

Чтобы открыть синтакс-помощник, нужно нажать соответствующую кнопку на панели инструментов конфигуратора или выполнить команду главного меню Справка Синтакс-помощник (рис. 5.20).

Как и любая другая справочная система, он представляет собой древовидную структуру, состоящую из глав, разделов, подразделов и т. п. Содержание синтакс-помощника полностью дублирует описание встроенного языка в пяти томах, входящих в стандартный комплект поставки «1С:Предприятия». Однако пользоваться синтакспомощником, на наш взгляд, удобнее, так как он находится сразу под рукой и имеет возможность контекстной помощи (Ctrl + F1).

Кроме того, в синтакс-помощнике в конце каждого описания находится ссылка Методическая информация (см. рис. 5.20). По этой ссылке открывается окно браузера, в котором подобраны методические материалы для выбранного раздела. Источниками материалов являются: ИТС, партнерская конференция, база знаний по технологическим вопросам крупных внедрений, сайт «1С:Предприятия», конференция начинающих разработчиков и др. Ссылки на методические материалы постоянно обновляются. Таким образом, разработчики могут быстро, не отрываясь от работы, найти информацию по нужному вопросу на различных ресурсах фирмы «1С».

Анализ кода с помощью синтакс-помощника

Пользоваться синтакс-помощником удобно в тех случаях, когда нужно разобраться в уже написанном незнакомом коде. На примере нашего обработчика события МатериалыКоличествоПриИзменении

(см. листинг 4.1) продемонстрируем, как понять код обработчика, используя синтакс-помощник.

Найти нужный раздел в содержании и спускаться вниз по дереву, раскрывая нужные подразделы, свойства, ссылки и т. п.

Итак, перед нами первая строка нашего обработчика (листинг 5.19).

листинг 5.19. Процедура «МатериалыКоличествоПриИзменении» (первая строка)

СтрокаТабличнойЧасти = Элементы.Материалы.ТекущиеДанные;

Чтобы понять этот код, нужно прежде всего понимать, в каком контексте он исполняется. Программный контекст зависит от того, в каком модуле располагается код. В данном случае процедура обработчика находится в модуле формы, следовательно, мы находимся в контексте модуля формы.

Будем изучать нашу строку последовательно слева направо. Что такое СтрокаТабличнойЧасти? Слева от оператора присваивания

(=) находится либо какое-то свойство, доступное нам непосредственно в этом контексте, либо переменная.

Согласно алгоритму, мы должны проверить:

Объявлена ли в модуле формы переменная СтрокаТабличнойЧасти? Откроем модуль формы. Мы не видим здесь строки описания переменной (Перем СтрокаТабличнойЧасти;), значит это не переменная модуля формы.

- Есть ли у формы реквизит СтрокаТабличнойЧасти? Откроем форму документа ПриходнаяНакладная и перейдем в окно реквизитов формы, расположенное справа вверху редактора форм

(рис. 5.21).

Мы видим, что у формы один основной (он выделен жирным шрифтом) реквизит Объект. Значит, реквизита СтрокаТабличнойЧасти у формы нет.

- Есть ли у объекта УправляемаяФорма свойство СтрокаТабличнойЧасти? Посмотрим в синтакс-помощнике свойства управляемой формы. Откроем синтакс-помощник на закладке Содержание. Управляемая форма – это объект интерфейса управляемого приложения, поэтому раскроем раздел Интерфейс (управляемый) Управляемая форма. Затем раскроем объект УправляемаяФорма и его

Свойства (рис. 5.22).

Рис. 5.22. Список свойств объекта «УправляемаяФорма» в синтакс-помощнике

Свойства расположены в алфавитном порядке. Мы видим, что среди них нет свойства СтрокаТабличнойЧасти.

- Есть ли у расширения формы свойство СтрокаТабличнойЧасти? Мы знаем, что основной реквизит формы содержит данные объекта ДокументОбъект.ПриходнаяНакладная (см. рис. 5.21). Следовательно, в модуле формы становятся доступны свойства, методы объекта встроенного языка Расширение управляемой формы для документа (синтакс-помощник – Интерфейс (управляемый) Управляемая форма Расширение документа). Посмотрим на них (рис. 5.23).

для документа» в синтакс-помощнике

Мы видим, что среди них нет свойства СтрокаТабличнойЧасти.

- Есть ли свойство глобального контекста СтрокаТабличнойЧасти? Откроем в синтакс-помощнике свойства глобального контекста (рис. 5.24).

Мы видим, что среди них нет свойства СтрокаТабличнойЧасти. Выражение СтрокаТабличнойЧасти также не может быть именем неглобального общего модуля, так как к его процедурам следует обращаться через точку (СтрокаТабличнойЧасти.). Также это выражение не может быть экспортируемой процедурой глобального общего модуля, так как в этом случае мы могли бы только вызвать эту процедуру как СтрокаТабличнойЧасти (), а не присваивать ей что-то.

- Есть ли в модуле управляемого приложения экспортная переменная СтрокаТабличнойЧасти? Откроем модуль управляемого приложения. Мы не видим здесь строки описания переменных (Перем СтрокаТабличнойЧасти Экспорт;), значит, это не переменная модуля управляемого приложения.

Таким образом, понятно, что выражение СтрокаТабличнойЧасти – это локальная переменная процедуры МатериалыКоличествоПриИзменении. В процессе выполнения программы ей присваивается какое-то значение. Переменные во встроенном языке не типизированные, поэтому в любой момент ей можно присвоить значение любого типа. Если переменная локальная, то есть используется только в контексте данной процедуры, то не требуется и ее явного объявления. Она объявляется в момент первого ее использования.

Анализ кода с помощью отладчика

Пользоваться отладчиком наиболее удобно в тех случаях, когда нужно написать какой-то собственный код. Потому что в отличие от синтакс-помощника, где нужно, вообще говоря, хорошо представлять контексты исполнения, структуру объектов и пр., с помощью отладчика ничего этого представлять не нужно.

Можно просто остановиться в конкретном месте программы и посмотреть, какие же свойства здесь доступны или какие программные объекты здесь используются.

Отладчик – вспомогательный инструмент, облегчающий разработку и отладку программных модулей системы «1С:Предприятие». Отладчик предоставляет следующие возможности:

- пошаговое выполнение модуля,
- расстановка точек останова,
- прерывание и продолжение выполнения модуля,

- возможность отладки нескольких модулей одновременно,
- вычисление выражений для анализа состояния переменных,
- просмотр стека вызовов процедур и функций, возможность остановки по возникновению ошибки,
- возможность редактирования модуля в процессе отладки.

Но мы пока не будем подробно останавливаться на всех этих возможностях, а рассмотрим использование отладчика для того, чтобы разобраться с обработчиком события МатериалыКоличествоПриИзменении, приведенном в листинге 4.1.

Если в режиме Конфигуратор редактируется текст модуля, то становятся доступными команды пункта главного меню Отладка, позволяющие расставлять и убирать точки останова. *Точки останова* позволяют прерывать выполнение программы в тех местах, где они установлены. Затем разработчик может проанализировать значение и тип выражений и переменных модуля в момент остановки и продолжить выполнение программы до следующей точки останова и т. д.

Итак, откроем форму документа ПриходнаяНакладная, перейдем на закладку Модуль, откроем текст процедуры МатериалыКоличествоПриИзменении. Мы видим, что в пункте главного меню Отладка и на панели инструментов конфигуратора стали доступны команды для работы с точками останова (рис. 5.36).

Рис. 5.36. Панель инструментов «Точки останова»

особенности использования ссылочных данных

В этом разделе мы поговорим об особенностях использования ссылочных данных, так как, используя доступ к этим данным с помощью запросов, мы можем значительно повысить скорость проведения документа и оптимизировать этот процесс.

Термином «ссылочные данные» мы будем обозначать данные, хранящиеся в базе данных, доступ к которым возможен при помощи объектов встроенного языка вида Ссылка: СправочникСсылка.<имя>, ДокументСсылка.<имя> и т. д. Для того чтобы дальнейшее изложение было понятнее, мы построим объяснение на примере получения ссылки на вид номенклатуры при проведении документа ОказаниеУслуги.

Не все данные, хранящиеся в базе данных, являются ссылочными. Это связано с тем, что в модели данных «1С:Предприятия» существует деление на данные, представляющие объектные сущности (справочники, планы счетов, документы и т. д.), и данные, представляющие необъектные сущности (регистры сведений, регистры накопления и т. д.).

С точки зрения платформы некоторая совокупность объектных данных определяется не только значениями своих полей, но и самим фактом своего существования. Другими словами, удалив из базы некоторую совокупность объектных данных, мы не сможем вернуть систему в то же состояние, которое было до удаления. Даже если мы заново создадим ту же самую совокупность объектных данных с теми же самыми значениями полей, с точки зрения системы это будет ДРУГАЯ совокупность объектных данных.

Каждую такую совокупность объектных данных, уникальную с точки зрения системы, называют объектом базы данных.

Для того чтобы система могла отличить один объект базы данных от другого, каждый объект базы данных (совокупность объектных данных) имеет внутренний идентификатор. Различные объекты базы данных всегда будут иметь разные внутренние идентификаторы. Этот идентификатор хранится вместе с остальными данными объекта в специальном поле Ссылка.

Необъектные данные хранятся в виде записей и с точки зрения системы определяются исключительно значениями своих полей.

Таким образом, удалив некоторую запись и записав после этого новую, с точно такими же значениями всех полей, мы получим то же самое состояние базы данных, которое было до удаления.

Таким образом, поскольку мы можем однозначно указать на каждый объект базы данных, у нас появляется возможность хранить такой указатель в полях других таблиц базы данных, выбирать его в поле ввода, указывать в параметрах запроса при поиске по ссылке и т. д.

Во всех этих случаях как раз и будет использоваться объект встроенного языка вида Ссылка. Фактически этот объект хранит только внутренний идентификатор, находящийся в поле Ссылка.

Например, если взять наш документ ОказаниеУслуги, то в поле, хранящем реквизит табличной части Номенклатура, на самом деле находится внутренний идентификатор, указывающий на элемент справочника Номенклатура (рис. 14.1).

Рис. 14.1. Ссылка на элемент справочника «Номенклатура»

Когда в обработчике события ОбработкаПроведения документа

ОказаниеУслуги мы присваиваем значение реквизита табличной части Номенклатура какой-либо переменной, мы имеем дело с объектом встроенного языка ДокументОбъект.ОказаниеУслуги.

Этот объект содержит в себе значения всех реквизитов документа и реквизитов его табличных частей.

Поэтому обращение (листинг 14.1) приводит к тому, что мы просто читаем данные, хранящиеся в оперативной памяти, в этом самом объекте встроенного языка (рис. 14.2).

листинг 14.1. Обращение к реквизиту объекта

Движение.Материал = ТекСтрокаПеречеьНоменклатуры.Номенклатура;

Рис. 14.2. Чтение данных из оперативной памяти

Однако когда мы обращаемся к виду номенклатуры как к реквизиту того элемента справочника, ссылка на который указана в табличной части документа (листинг 14.2), происходит буквально следующее (рис. 14.3).

листинг 14.2. Обращение к реквизиту ссылки

Если ТекСтрокаПеречеьНоменклатуры.Номенклатура.ВидНоменклатуры =

Перечисления.ВидыНоменклатуры.Материал Тогда

Рис. 14.3. Использование кеша объектов

Поскольку в объекте ДокументОбъект.ОказаниеУслуги есть только ссылка на элемент справочника Номенклатура и больше никаких данных об этом элементе нет, платформа возьмет эту ссылку и обратится по ней в кеш объектов в надежде найти там данные того объекта, ссылка на который у нее есть.

Если кеш объектов не будет иметь нужных данных, он обратится к базе данных с тем, чтобы прочитать все данные объекта, ссылкой на который он обладает.

После того как все данные, хранящиеся в реквизитах нужного элемента справочника и в реквизитах его табличных частей, будут считаны в кеш объектов, кеш объектов вернет запрашиваемую ссылку, хранящуюся в реквизите ВидНоменклатуры справочника Номенклатура.

Если же в алгоритме проведения требуется анализировать дополнительные реквизиты объектов, ссылки на которые содержатся в документе, а также использовать результаты расчета итогов регистров, следует использовать запросы для более быстрой выборки данных из базы данных.

То же самое справедливо в отношении выполнения любых участков программы, критичных по производительности. Механизм запросов лучше «читает» информационную базу и может за один раз выбрать только те данные, которые необходимы. Поэтому, например, в типовых

решениях вы практически не увидите использования объекта встроенного языка СправочникВыборка.<имя>. Вместо этого повсеместно используются запросы к базе данных.

Объекты.

Что такое объект в терминах «1С:Предприятия»? Этот вопрос зачастую ставит в тупик не только начинающих разработчиков, но и людей, имеющих определенный опыт разработки на платформе «1С:Предприятие».

Основная трудность заключается в том, что всегда нужно ясно представлять себе, в каком контексте употребляется этот термин.

Как правило, термин *объект* употребляется в одном из трех контекстов:

- конфигурация,
- база данных,
- встроенный язык.

Говоря о конфигурации, термином *объект конфигурации* мы обозначаем некоторую совокупность описания данных и алгоритмов работы с этими данными. Например, в конфигурации может существовать объект Справочник Сотрудники.

На основании каждого объекта конфигурации в базе данных создается информационная структура, в которой будут храниться данные.

Так вот, когда мы говорим о базе данных, термином «объект» мы обозначаем всего лишь некий элемент такой информационной структуры. Характерной особенностью такого элемента является то, что на него (как на совокупность данных) существует ссылка, которая может являться значением какого-либо поля другой информационной структуры.

Например, в базе данных существует справочник Сотрудники, в котором есть сотрудник Иванов. В этом случае элемент справочника, содержащий информацию о сотруднике Иванове, будет являться объектом базы данных. И если в документе ПриходнаяНакладная будет существовать реквизит ОтветственноеЛицо, то тип значения этого реквизита будет ссылкой на объект базы данных, то есть на элемент справочника, содержащий информацию об Иванове.

Если же мы начинаем говорить о встроенном языке и о том, каким образом средствами встроенного языка работать со справочниками, то термином «объект» мы обозначаем тип данных, позволяющий получить доступ к данным и обладающий набором свойств и методов.

Существует целый ряд объектов встроенного языка, позволяющих работать со справочниками (СправочникиМенеджер, СправочникМенеджер.<имя>, СправочникСсылка.<имя> и т. д.).

Среди них есть один объект, который предоставляет доступ к объекту справочника в базе данных, – СправочникОбъект.<имя> (рис. 5.45).

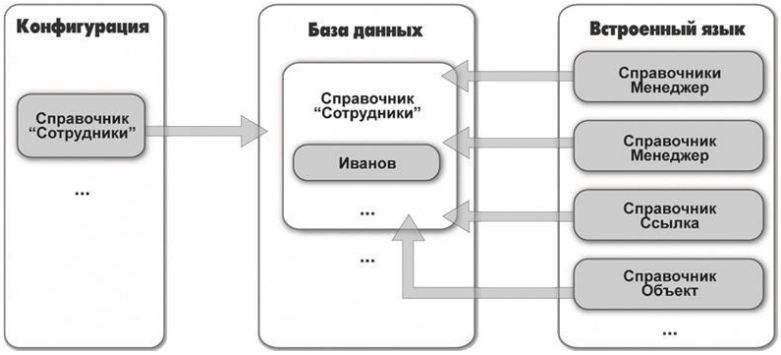


Рис. 5.45. Объект конфигурации, объект базы данных,

объекты встроенного языка

Сервер и клиенты

На предыдущем занятии мы создали одну процедуру для обработки нескольких событий и поместили ее в общий модуль РаботаСДокументами. В свойствах этого общего модуля мы устанавливали флажки Клиент и Сервер. Объясним подробнее, откуда в «1С:Предприятии» взялись вообще какие-то «клиенты» и «серверы».

Система «1С:Предприятие» поддерживает два варианта работы системы: файловый и клиент-серверный.

Файловый вариант работы с информационной базой рассчитан на персональную работу одного пользователя или работу небольшого количества пользователей в локальной сети. В этом варианте все данные информационной базы (конфигурация, база данных, административная информация) располагаются в одном файле (рис. 5.46).

Рис. 5.46. Файловый вариант работы

Основное назначение файлового варианта – быстро и легко установить систему и работать с ней. Например, чтобы что-то посмотреть или доработать дома на ноутбуке. В файловом варианте тоже можно вести реальную учетную работу, но при этом нужно понимать, что он не предоставляет абсолютно всех тех же возможностей по масштабируемости, защите данных, какие имеет клиент-серверный вариант. Поэтому, если вы самостоятельно ведете бухгалтерский учет или у вас небольшой коллектив сотрудников, и вам не требуется гарантированная защита данных от несанкционированного использования сотрудниками, и вы имеете относительно небольшой объем данных, можно работать в файловом варианте. В остальных же случаях нужно использовать клиент-серверный вариант.

Клиент-серверный вариант предназначен для использования в рабочих группах или в масштабе предприятия. Он реализован на основе трехуровневой архитектуры «*клиентсервер*

Клиент-серверный вариант работы – это основной вариант для работы в многопользовательской среде с большим объемом данных. Он предоставляет абсолютно все возможности по масштабируемости, администрированию и защите данных. Однако он требует значительных усилий по установке и администрированию.

При этом физически серверная и клиентские части системы «1С:Предприятие» могут располагаться как на разных компьютерах, так и на одном. Главное, что пользователь не имеет непосредственного доступа к серверу баз данных, и это позволяет обеспечивать

безопасность данных. А в файловом варианте база данных должна находиться на некотором общем сетевом ресурсе, доступном всем пользователям.

Система «1С:Предприятие» изначально рассчитана на клиентсерверный вариант работы. Хотя сейчас вы разрабатываете свою учебную конфигурацию в файловом варианте работы, она будет работать и в клиент-серверном варианте без ваших дополнительных усилий.

Прикладные решения разрабатываются один раз и одинаково работают, что в одном, что в другом варианте. То есть переход с одного варианта на другой не требует переделки конфигурации.

Это достигается за счет того, что конфигурация разрабатывается всегда исходя из клиент-серверной архитектуры. В системе «1С:Предприятие» просто нет возможности разрабатывать ее по-другому. И в том случае, когда используется файловый вариант работы, система при исполнении прикладного решения просто «имитирует» наличие сервера на клиентском компьютере.

Клиент-серверная архитектура разделяет всю работающую систему на три различные части, определенным образом взаимодействующие между собой, – клиент, сервер «1С:Предприятия» и сервер баз данных.

Клиентское приложение – это программа, часть системы «1С:Предприятие». Основное ее назначение – организация пользовательского интерфейса, отображение данных с возможностью их изменения. Кроме этого, клиентское приложение может исполнять код на встроенном языке (то есть какие-то алгоритмы разработчика), но оперирует при этом лишь очень ограниченным пространством типов встроенного языка. Такой подход позволяет клиентскому приложению быть очень «легким», не требовать много ресурсов, «путешествовать» по Интернету и работать даже в среде веб-браузеров.

Клиентское приложение взаимодействует с сервером «1С:Предприятия». *Сервер «1С:Предприятия»* – это тоже программа, часть системы «1С:Предприятие».

Одна из основных задач этой программы – передавать запросы от клиентского приложения к серверу баз данных и возвращать обратно клиенту результаты этих запросов.

Другая задача сервера – исполнение большинства алгоритмов на встроенном языке, подготовка данных для отображения форм, отчетов и т. д. То есть все сложные вычисления, требующие непосредственной работы с данными, исполняются именно на сервере. При этом на сервере доступно практически все пространство типов встроенного языка «1С:Предприятия» (за исключением, естественно, чисто интерфейсных типов, потому что у сервера нет никакой интерфейсной части, так как он общается не с пользователями, а только с другими программами: клиентским приложением и с сервером баз данных).

Сервер баз данных – это тоже программа. Она уже не является частью системы «1С:Предприятие», это специализированная программа, поставляемая сторонними производителями.

Ее основное назначение – это организация и ведение баз данных – структурированных организованных наборов данных, описывающих характеристики каких-либо физических или виртуальных систем.

В настоящее время система «1С:Предприятие» может работать со следующими серверами баз данных:

- Microsoft SQL Server,
- PostgreSQL,
- IBM DB2,
- Oracle Database.

Компиляция общих модулей

На предыдущем занятии мы создали одну процедуру для обработки нескольких событий и поместили ее в общий модуль РаботаСДокументами. У этого модуля, как и у всякого общего модуля конфигурации, существует набор свойств: Клиент (управляемое приложение), Сервер и Внешнее соединение. Значения этих свойств (истина/ложь) определяют, где будут скомпилированы экземпляры этих модулей.

Расскажем подробнее о том, что происходит, когда мы устанавливаем те или иные флажки у общего модуля.

Прежде всего, необходимо понимать, зачем необходима компиляция. Дело в том, что все, что мы разработали и написали в конфигурации, – пока только некая «заготовка». Платформа, когда мы запускаем ее в режиме 1С:Предприятие, превращает все это в программу, которую уже можно исполнить на компьютере, – компилирует. При этом, как мы уже сказали ранее, есть разные части системы, в которых исполняется код – сервер, клиентские приложения. Поэтому для общих модулей мы можем и должны в явном виде указать, где, на какой «стороне» они должны быть скомпилированы – на сервере или клиенте.

Логическая связь объектов

Для реализации этого примера нам понадобятся три новых объекта конфигурации.

Прежде всего, это План видов характеристик. Он будет хранить виды характеристик, которыми в принципе можно описывать материалы.

Кроме этого, нам понадобится специальный справочник, подчиненный справочнику Номенклатура. Элементы этого справочника будут идентифицировать партии материалов с некоторым фиксированным набором значений характеристик.

И третий объект – это регистр сведений, в котором собственно и будет храниться соответствие конкретных значений характеристик некоторому варианту материала (см. рис. 15.4).

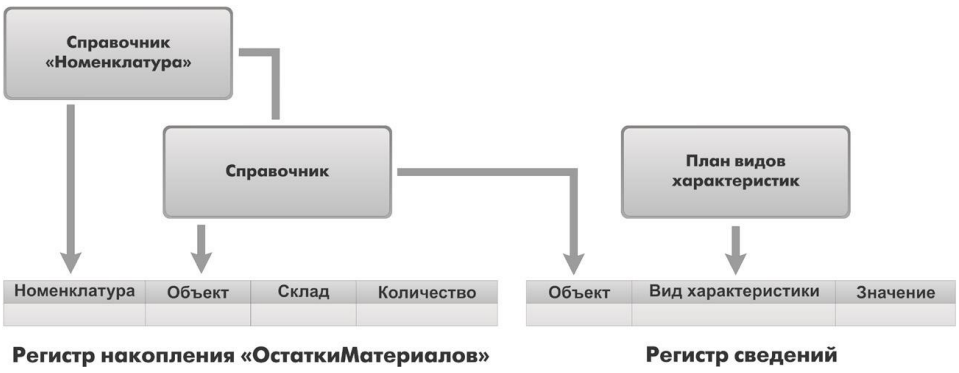


Рис. 15.4. Логическая связь объектов

В результате использования такой логической структуры объектов мы получим возможность описывать каждую поступающую партию материала любым количеством видов характеристик, поскольку это соответствие будет храниться в регистре сведений.

И вместе с этим мы получим возможность вести учет в разрезе видов характеристик, добавив в регистры накопления еще одно измерение для хранения ссылки на элемент справочника, подчиненного справочнику Номенклатура (рис. 15.4).

В результате для того, чтобы узнать остатки материалов, обладающих некоторым значением характеристики, достаточно будет выбрать из регистра сведений все элементы подчиненного справочника с этим значением характеристики и затем по ним и их владельцам получить остатки регистра накопления.

Создание новых объектов конфигурации

Как мы уже говорили, нам понадобится создать несколько новых объектов конфигурации:

- справочник ВариантыНоменклатуры, чтобы описывать партии материалов;
- справочник ДополнительныеСвойстваНоменклатуры, чтобы задавать значения видов характеристик, для которых нет подходящих типов в конфигурации;
- план видов характеристик СвойстваНоменклатуры, чтобы создавать виды характеристик;
- регистр сведений ЗначенияСвойствНоменклатуры, чтобы хранить значения видов характеристик для различных партий материалов.

Сначала создадим объект конфигурации Справочник с именем

ВариантыНоменклатуры и укажем, что он будет подчинен справочнику Номенклатура. Для этого на закладке Владельцы добавим справочник Номенклатура в список владельцев справочника ВариантыНоменклатуры.

Затем создадим еще один объект конфигурации Справочник с именем ДополнительныеСвойстваНоменклатуры.

После этого создадим объект конфигурации План видов характеристик с именем СвойстваНоменклатуры.

Установим его свойство Тип значения характеристик.

Для этого нажмем кнопку выбора ... и зададим составной тип данных следующим образом (рис. 15.5):

- Число, длина 15, точность 3;
- Строка, длина 25;
- Дата;
- Булево;
- СправочникСсылка.ДополнительныеСвойстваНоменклатуры.

Рис. 15.5. Определение составного типа данных для типа значения характеристик плана видов характеристик

Затем справочнику ДополнительныеСвойстваНоменклатуры укажем владельца – план видов характеристик СвойстваНоменклатуры

Описание характеристик вариантов номенклатуры

В заключение для справочника ВариантыНоменклатуры опишем, где хранятся свойства вариантов номенклатуры и как получить значения этих свойств. Это описание платформа будет использовать автоматически при выполнении отчетов и при формировании различных динамических списков, в которых задействуются варианты номенклатуры.

В контекстном меню справочника ВариантыНоменклатуры выберем команду Характеристики (рис. 15.10).

Рис. 15.10. Переход к характеристикам справочника

«ВариантыНоменклатуры»

Откроется диалог описания характеристик. С помощью кнопки Добавить в командной панели добавим в него новую запись. В качестве источника характеристик выберем план видов характеристик СвойстваНоменклатуры. Платформа автоматически определит, что полем ключа будет являться поле Ссылка этого объекта конфигурации (рис. 15.11).

Два оставшихся поля, Поле отбора видов и Значение отбора, оставим пустыми. В нашем случае эти поля не понадобятся.

Перейдем к описанию того, где и как хранятся значения наших свойств. В качестве источника значений характеристик выберем регистр сведений ЗначенияСвойствНоменклатуры. Платформа автоматически определит, что в этом регистре полем объекта является измерение НаборСвойств, а полем вида – измерение ВидСвойства.

Рис. 15.11. Описание источника видов характеристик

Поэтому единственное, что нам останется указать самостоятельно, что значения свойств хранятся в ресурсе Значение. В результате описание характеристик для справочника ВариантыНоменклатуры будет выглядеть следующим образом (рис. 15.12).

Рис. 15.12. Описание характеристик для справочника

«ВариантыНоменклатуры»

На этом занятии мы проиллюстрируем возможность ведения бухгалтерского учета средствами «1С:Предприятия». В рамках этого занятия мы не будем объяснять и рассматривать основы бухгалтерского учета. Поэтому если у вас нет знаний бухгалтерии, то, конечно, лучше сначала прочитать какую-нибудь популярную литературу о том, как вообще устроен бухгалтерский учет в нашей стране.

Для организации бухгалтерского учета мы используем уже знакомый нам объект конфигурации – план видов характеристик и два новых объекта – *План счетов* и *Регистр бухгалтерии*.

Регистр бухгалтерии будет использоваться нами для накопления данных о совершенных хозяйственных операциях.

С помощью плана счетов мы будем описывать счета, в разрезе которых ведется учет, а план видов характеристик будет служить для описания объектов аналитического учета, в разрезе которых должен вестись учет на счетах.

Сразу оговоримся, что план счетов, который мы будем использовать в нашей учебной конфигурации, очень сильно упрощен. Он содержит всего несколько условных счетов, которые, однако, позволят нам познакомиться с основными методами организации бухгалтерского учета средствами «1С:Предприятия».

План видов характеристик в бухгалтерском учете

Объект конфигурации План видов характеристик был подробно рассмотрен нами на предыдущем занятии, поэтому сейчас мы проиллюстрируем только использование этого объекта в контексте бухгалтерского учета.

Бухгалтерский учет, как правило, подразумевает ведение аналитического учета на большинстве счетов. Для обозначения разрезов аналитического учета мы будем использовать термин *виды субконто*. То есть на каждом счете учет может вестись в разрезе нескольких видов субконто.

А для обозначения конкретных объектов аналитического учета мы будем использовать термин *субконто*.

Например, на 41 счете (Товары) учет ведется обычно в разрезе Номенклатуры и Складов, которые являются видами субконто.

А вот конкретная номенклатура Паста шоколадная и конкретный склад Основной, указанные для некоторой проводки по 41 счету, – это субконто.

Так вот, частным случаем использования плана видов характеристик является применение его для описания видов субконто. То есть все разрезы аналитического учета описываются в соответствующем плане видов характеристик, и там же задаются типы значений, которые могут принимать те или иные субконто.

Добавление плана видов характеристик

Приступим к созданию плана видов характеристик, который будет содержать описания разрезов аналитического учета – видов субконто.

Откроем конфигуратор и добавим новый объект конфигурации План видов характеристик. Зададим его имя – ВидыСубконто. На закладке Подсистемы укажем, что план счетов будет отображаться в подсистеме Бухгалтерия.

Поскольку нам понадобится некий вспомогательный справочник, в котором пользователи будут осуществлять «свободное творчество» по созданию значений новых объектов аналитического учета, добавим объект конфигурации Справочник и назовем его Субконто.

Затем на закладке Владельцы укажем, что этот справочник будет подчинен плану видов характеристик ВидыСубконто.

Для этого на закладке Владельцы нажмем кнопку Редактировать элемент списка и выберем в качестве владельца справочника план видов характеристик ВидыСубконто (рис. 16.1).

Закроем окно редактирования справочника и вернемся к нашему плану видов характеристик.

На закладке Основные установим свойство Тип значения характеристик. Нажмем кнопку выбора и зададим составной тип данных следующим образом (рис. 16.2):

Бухгалтерия нашего ООО «На все руки мастер» ведет учет движения денежных средств только в разрезе материалов и клиентов, но не исключено, что в дальнейшем понадобится дополнительная аналитика (поэтому мы и используем справочник Субконто).

Обратите внимание, что тот справочник, который будет использован в качестве дополнительных значений характеристик, тоже должен входить в составной тип данных типа значений характеристик, иначе конфигуратор выдаст сообщение об ошибке.

Затем укажем, что дополнительные значения характеристик будут находиться в справочнике Субконто.

После этого перейдем на закладку Прочее и, нажав кнопку Предопределенные, начнем ввод предопределенных значений плана видов характеристик (рис. 16.3). То есть тех видов аналитического учета, в разрезе которых мы будем вести учет.

Рис. 16.3. Предопределенные виды характеристик

Нажимая кнопку Добавить, создадим предопределенный вид субконто Материалы с кодом 000000001 и типом СправочникСсылка.Номенклатура.

Затем создадим вид субконто Клиенты с кодом 000000002 и типом СправочникСсылка.Клиенты.

На этом создание видов субконто завершено, и мы можем перейти к знакомству со следующим объектом конфигурации, который будет использован нами, – План счетов.

Что такое «План счетов»

Объект конфигурации *План счетов* предназначен для описания структуры хранения информации о совокупности синтетических счетов предприятия, которые созданы для группировки данных о его хозяйственной деятельности.

На основе объекта конфигурации План счетов платформа создает в базе данных таблицы, в которых будет храниться информация о том, какие счета и каким образом будет использовать предприятие.

Это может быть система бухгалтерских счетов, установленная государством, план управленческих счетов или произвольный набор счетов, используемых для анализа тех или иных видов деятельности предприятия.

План счетов в системе «1С:Предприятие» поддерживает иерархию субсчетов: к каждому счету первого уровня может быть открыто несколько субсчетов, которые, в свою очередь, могут иметь свои субсчета, и так далее.

Например, законодательно утвержденный план счетов для ведения бухучета в России имеет следующий вид (рис. 16.4).

Рис. 16.4. Российский план счетов

По любому счету или субсчету может вестись аналитический учет в разрезе субконто, описанных в плане видов характеристик. Связь между планом счетов и планом видов характеристик задается разработчиком на этапе конфигурирования.

Для описания используемых субконто система создает в плане счетов специальную табличную часть ВидыСубконто, которая не видна в конфигураторе (но доступна средствами встроенного языка).

Зачем нужен план видов расчета и регистр расчета?

На этом занятии мы рассмотрим, какие возможности для автоматизации сложных периодических расчетов предоставляет система «1С:Предприятие».

Такие расчеты используются прежде всего при расчете заработной платы. Поэтому дальнейшее их рассмотрение мы будем строить на примере расчета заработной платы сотрудников, которые работают в нашем ООО «На все руки мастер».

В общем случае сумма заработной платы сотрудника складывается из множества частей (например, оплата по окладу, премии, штрафы, оплаты по больничному листу, разовые выплаты и т. д.). Каждая из этих частей рассчитывается по некоторому алгоритму, присущему только этой части.

Например, сумма штрафа может определяться просто фиксированной суммой, сумма премии может рассчитываться как процент от оклада, а сумма оплаты по окладу рассчитывается исходя из количества рабочих дней в месяце и количества дней, отработанных сотрудником. Поэтому для обозначения каждой такой части мы будем использовать термин *вид расчета*.

Алгоритм каждого вида расчета опирается в общем случае на две категории параметров: период, за который нужно получить конечные данные, и набор некоторых исходных данных, используемых при расчете.

Как правило, в реальной жизни различные виды расчета существуют не сами по себе, а оказывают некоторое влияние на другие виды расчета. Исходя из того, что вид расчета опирается на две различные категории параметров, такое влияние тоже имеет двойственный характер.

Зависимость по базовому периоду

Это может быть влияние на исходные данные, используемые при расчете.

В качестве примера можно привести начисление премии в виде процента от оплаты по окладу. При изменении оплаты по окладу размер премии тоже должен быть пересчитан, исходя из новой суммы начисленного оклада. Другими словами, сумма начисленного оклада является базой для расчета премии.

Причем поскольку оклад рассчитывается за некоторый период, при расчете премии нам интересно знать не значение оклада вообще, а сумму, начисленную в том периоде, который влияет на расчет премии. Такой период мы будем называть *базовым*, а подобную зависимость между видами расчета – *зависимостью по базовому периоду*.

В качестве примера рассмотрим начисление премии за апрель. Премия должна начисляться в размере 10 % от суммы, начисленной в качестве оплаты по окладу. Следовательно,

необходимо проанализировать все записи о начислениях оплаты по окладу, которые попадают в интересующий нас базовый период, а именно апрель.

Допустим, общая сумма таких начислений составила 8 000 рублей – в этом случае премия должна быть начислена в размере 800 рублей (рис. 17.1).

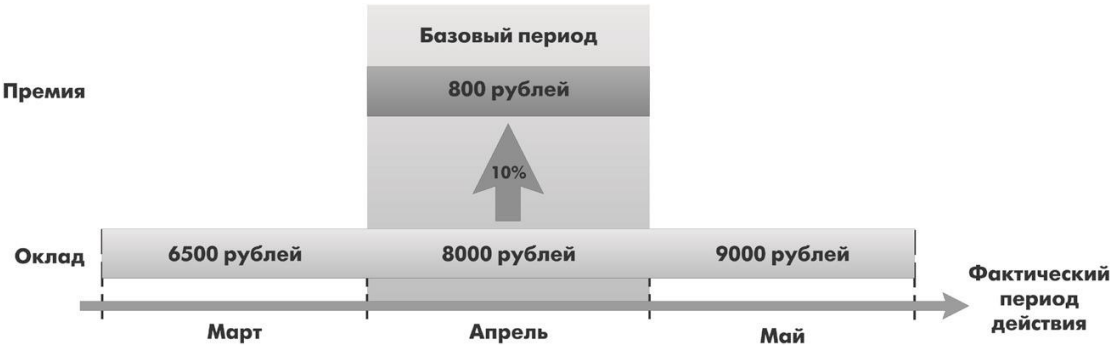


Рис. 17.1. Зависимость премии от оклада по базовому периоду

Вытеснение по периоду действия

Это влияние может быть не на исходные данные, а на сам период, за который производится расчет.

В качестве примера можно привести расчет оплаты по окладу и невыход на работу. Предположим, мы начислили сотруднику оплату по окладу за март. В этом случае период действия такого расчета будет с 01.03.2013 по 31.03.2013.

После этого мы получили информацию от руководителя отдела, что, оказывается, сотрудник отсутствовал на работе с 1 по 10 марта по неизвестной причине. В этом случае нам нужно будет произвести расчет Невыход (в котором можно рассчитать какие-то удержания с сотрудника).

Но кроме этого необходимо будет пересчитать и оклад сотрудника, исходя из того, что фактический период действия расчета Оклад стал теперь с 11.03.2013 по 31.03.2013.

Такое влияние мы будем называть *вытеснением по периоду действия*.

В результате если за полный месяц работы сотруднику должно было быть начислено 9 300 рублей, то теперь, за фактический период работы, начисление составит 6 300 рублей (рис. 17.2).

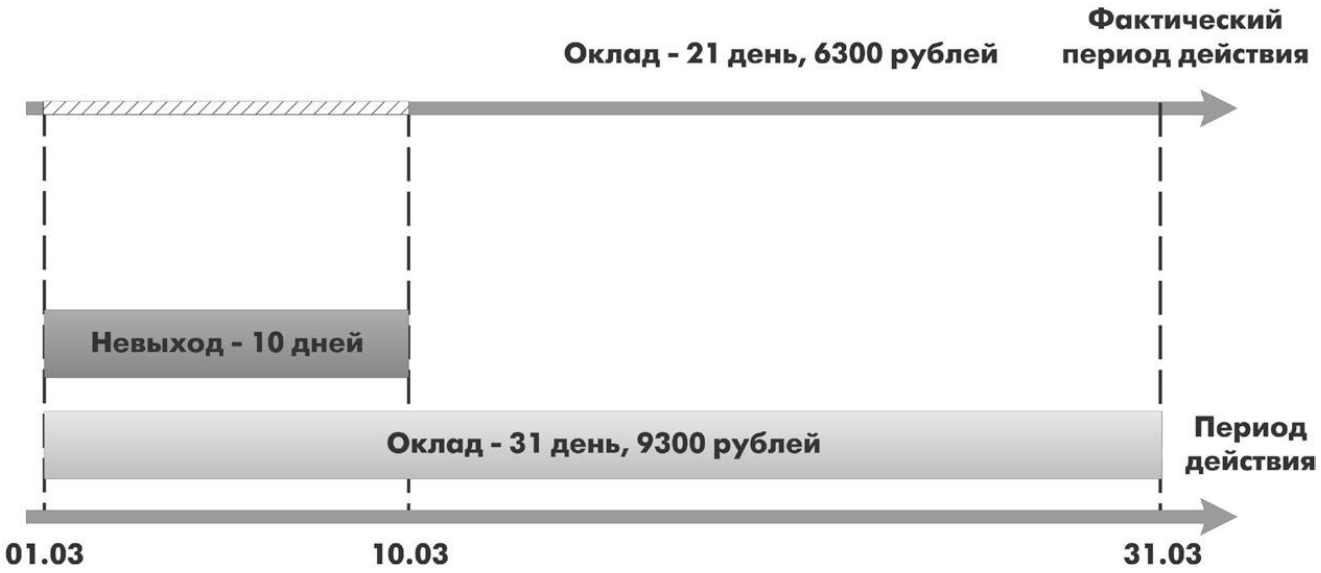


Рис. 17.2. Запись расчета «Невыход» вытесняет запись расчета «Оклад» по периоду действия

Таким образом, исходя из двух видов взаимного влияния расчетов, можно сказать, что в общем случае с каждым видом расчета будет связано три периода: период действия, фактический период и базовый период.

- *Период действия* является «запрашиваемым». То есть, указывая период действия, мы говорим: «Мы хотели бы, чтобы результат действовал в этом периоде».
- *Фактический период* – это то, что получилось из периода действия после анализа всех периодов действия расчетов, которые вытесняют наш по периоду действия.
- *Базовый период* – это период, в котором мы анализируем результаты других расчетов, влияющих на наш по базовому периоду.

Как видите, взаимное влияние между видами расчетов может быть довольно разнообразным и, что самое сложное, многоуровневым. То есть один вид расчета может влиять на другой, который, в свою очередь, влияет на третий и т. д.

Очевидно, что в этой ситуации требуется некий универсальный механизм, позволяющий описать каждый из видов расчетов (его алгоритм, влияние на другие виды расчетов, зависимость от других видов расчетов), обеспечить хранение данных, полученных в результате этих расчетов, и контроль необходимости перерасчета результатов зависимых расчетов в случае изменения результатов первичных расчетов.

В системе «1С:Предприятие» такой универсальный механизм реализован при помощи планов видов расчета и регистров расчета.

И первым объектом конфигурации, с которым мы начнем знакомиться на этом занятии, будет План видов расчета.

Что такое план видов расчета

Объект конфигурации *План видов расчета* предназначен для описания структуры хранения информации о возможных видах расчетов. На основе объекта конфигурации План видов расчета платформа создает в базе данных таблицу, в которой будет храниться информация о том, какие существуют виды расчета и каковы взаимосвязи между ними.

Отличительной особенностью плана видов расчета является то, что пользователь в процессе работы может добавлять новые виды расчета. Такая возможность делает механизм периодических расчетов более гибким и позволяет пользователю создавать собственные виды расчета, помимо тех, которые заданы разработчиком как предопределенные.

Объект конфигурации План видов расчета имеет свойство Использует период действия. С его помощью определяется, будут ли в этом плане находиться виды расчета, которые могут быть вытеснены по периоду действия.

Если это свойство установлено, то разработчик получает возможность указать для каждого вида расчета те виды, которые вытесняют его по периоду действия.

Следующим важным свойством объекта конфигурации План видов расчета является Зависимость от базы. Оно определяет, будут ли в этом плане находиться зависимые по базовому периоду виды расчета.

Если это свойство установлено, появляется возможность указать, в каком плане видов расчета будут находиться базовые виды расчета и, кроме этого, как будет определяться эта зависимость.

Существует возможность указать один из двух видов зависимости от базы: Зависимость по периоду действия и Зависимость по периоду регистрации.

Еще одной важной особенностью плана видов расчета является возможность создания предопределенных видов расчета и описания их взаимного влияния. При этом в общем случае разработчик имеет возможность указать три категории видов расчета, влияющих на предопределенный вид расчета:

- *Базовые* – их результаты должны быть использованы при перерасчете этого вида расчета;
- *Вытесняющие* – вытесняют этот вид расчета по периоду действия;
- *Ведущие* – изменение их результатов должно приводить к необходимости перерасчета этого вида расчета.

Теперь у нас все готово для того, чтобы начать разработку системы расчета заработной платы ООО «На все руки мастер».

В этой главе мы создадим документ, с помощью которого будут выполняться различные виды начислений; посмотрим, как и когда платформа формирует записи перерасчета; увидим, как работают механизмы вытеснения по периоду действия и зависимости по базовому периоду.

Кроме этого, мы создадим отчет, показывающий начисления сотрудникам ООО «На все руки мастер», и сделаем так, чтобы данные расчетов можно было поддерживать в актуальном состоянии.

В заключение мы познакомимся с новым элементом формы – *Диаграмма Ганта* – и с его помощью наглядно проиллюстрируем работу некоторых механизмов расчета.

Добавление документа о начислениях

Для того чтобы иметь возможность регистрировать в базе данных начисления, производимые сотрудникам ООО «На все руки мастер», нам понадобится специальный документ.

Откроем конфигуратор и добавим новый объект конфигурации Документ. Назовем его НачисленияСотрудникам. Зададим представление объекта как Начисление сотрудникам. На закладке Нумерация установим:

- Тип номера – Число,
- Длина номера – 5 (рис. 18.1).

На закладке Подсистемы укажем, что документ будет отображаться в подсистеме РасчетЗарплаты.

На закладке Данные укажем, что этот документ будет иметь табличную часть Начисления, содержащую следующие реквизиты:

- Сотрудник, тип СправочникСсылка.Сотрудники;
- ГрафикРаботы, тип СправочникСсылка.ВидыГрафиковРаботы;
- ДатаНачала, тип Дата;
- ДатаОкончания, тип Дата;
- ВидРасчета, тип ПланВидовРасчетаСсылка.ОсновныеНачисления; Начислено, Число, длина 15, точность 2.

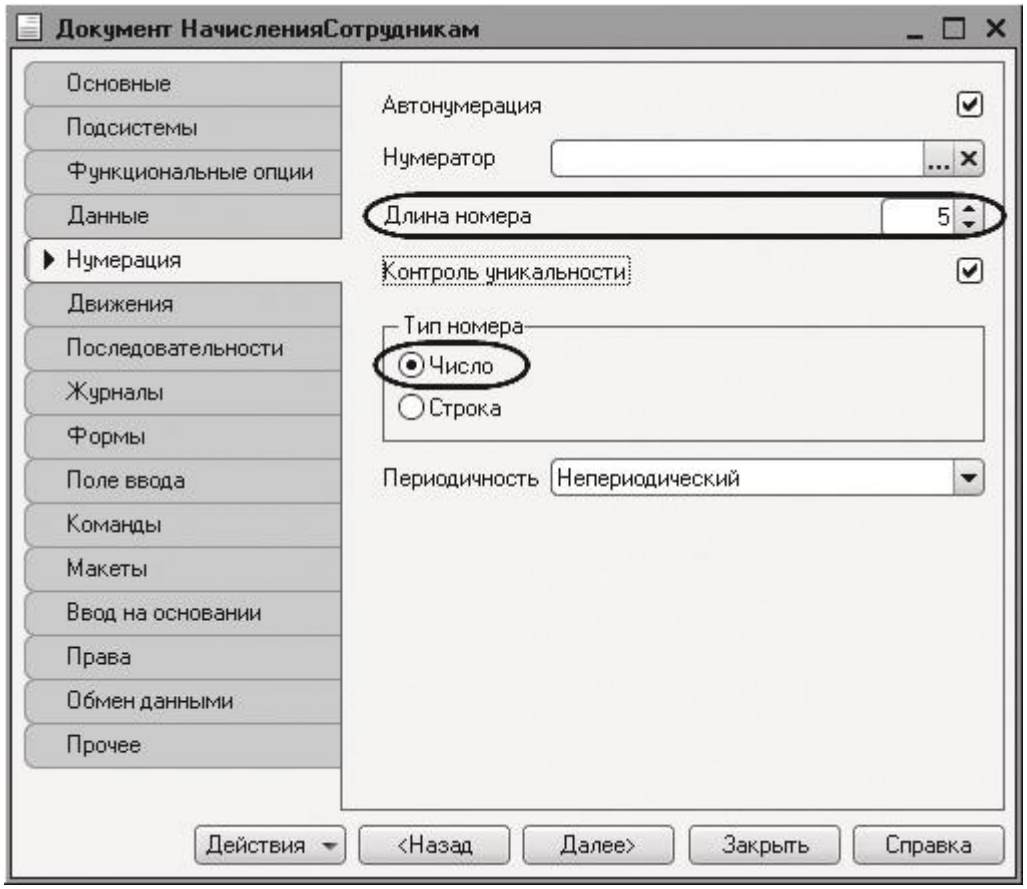


Рис. 18.1. Нумерация документа

Реквизиты ДатаНачала и ДатаОкончания понадобятся нам для того, чтобы задавать период, в котором должна действовать запись расчета.

На закладке Движения запретим оперативное проведение документа.

Отметим, что документ будет создавать движения по регистру расчета Начисления, и запустим конструктор движений (рис. 18.2).

Рис. 18.2. Движения документа

В окне конструктора выберем табличную часть Начисления и нажмем Заполнить выражения.

Для реквизитов ПериодДействияКонец и БазовыйПериодКонец укажем выражение КонецДня(ТекСтрокаНачисления.ДатаОкончания).

Для поля ПериодРегистрации укажем выражение Дата.

Реквизиту ИсходныеДанные поставим в соответствие реквизит табличной части Начислено, а для ресурса Результат оставим пустое выражение, так как мы будем его потом рассчитывать (рис. 18.3).

Информационная база нашей фирмы ООО «На все руки мастер» пока еще очень мала. В самом деле в процессе создания и проверки работы конфигурации мы добавили в нее всего лишь несколько элементов номенклатуры, провели небольшое количество документов.

Однако реальные информационные базы содержат гораздо больше разнообразной информации, и иногда поиск нужных данных становится достаточно сложной задачей, особенно для пользователя, плохо знакомого с номенклатурой товаров (услуг) или с перечнем контрагентов, с которыми работает фирма.

Специально для того, чтобы облегчить поиск незнакомой информации в базе данных, система «1С:Предприятие» содержит механизм полнотекстового поиска в данных. Преимущества этого механизма заключаются в том, что он позволяет искать данные, вводя поисковый запрос в простой и естественной форме, например: «телефон абдулова». При этом можно использовать специальные операторы, наподобие тех, что применяются при поиске в Интернете (И, ИЛИ, НЕ и т. д.).

Полнотекстовый поиск чрезвычайно удобен в тех случаях, когда мы не знаем точно, где находятся нужные нам данные (например, в каком справочнике). Также полнотекстовый поиск оказывается незаменим тогда, когда мы не знаем точно, что нужно искать (например, не помним точное название номенклатуры или контрагента).

Кроме этих возможностей полнотекстовый поиск позволяет находить данные там, где другие методы поиска крайне трудоемки или требуют создания специальных алгоритмов и обработок. Например, полнотекстовый поиск хорошо умеет работать с текстовыми полями большой длины и полями типа ХранилищеЗначения.

На этом занятии мы познакомимся с общими сведениями о механизме полнотекстового поиска в данных, создадим полнотекстовый индекс и на его основе попробуем найти нужные данные в базе данных нашего ООО «На все руки мастер».

Общие сведения о механизме полнотекстового поиска в данных

Механизм полнотекстового поиска «1С:Предприятия» основан на использовании двух составляющих:

- полнотекстового индекса,
- средств выполнения полнотекстового поиска.

Для того чтобы была возможность выполнять полнотекстовый поиск, обязательно должен существовать полнотекстовый индекс. Полнотекстовый индекс создается один раз и затем должен периодически обновляться.

Поиск осуществляется по тем данным, которые содержатся в полнотекстовом индексе. Таким образом, если ведется интенсивная работа с базой данных (например, данные изменяются, добавляются новые), то полнотекстовый индекс следует обновлять как можно чаще. Если же объем изменяемых или новых данных невелик, то обновление полнотекстового индекса можно выполнять реже, например раз в сутки, в период наименьшей загруженности системы.

Создание и обновление полнотекстового индекса может выполняться как интерактивно, в режиме 1С:Предприятие, так и программно, средствами встроенного языка. На этом занятии мы рассмотрим возможности интерактивного индексирования, а на следующем занятии вы узнаете, как можно обновлять полнотекстовый индекс в автоматическом режиме.

В процессе работы информационной базы система отслеживает факт изменения данных в тех объектах конфигурации, которые могут участвовать в полнотекстовом поиске. Такими объектами являются, например, планы обмена, справочники, документы, планы видов характеристик, планы счетов, планы видов расчета, регистры (сведений, накопления, бухгалтерии, расчета), бизнес-процессы и задачи.

Впоследствии при создании или обновлении полнотекстового индекса система анализирует данные, содержащиеся в реквизитах этих объектов, и включает эти данные в полнотекстовый индекс.

При этом анализироваться могут не все реквизиты, а только те, которые имеют тип Строка, Число, Дата, ХранилищеЗначения или ссылочный тип (например, СправочникСсылка.Номенклатура).

Собственно сам полнотекстовый поиск выполняется средствами встроенного языка. Немного забегая вперед, необходимо отметить, что полнотекстовый поиск выполняется в соответствии с правами пользователя. Таким образом, если какая-то информация недоступна данному пользователю, он не сможет получить ее и при помощи полнотекстового поиска.

Результаты полнотекстового поиска возвращаются порциями, и, кроме этого, они отсортированы в определенном порядке. Это сделано для того, чтобы с большой долей вероятности пользователь получал требуемые ему данные в начале первой порции. Как показывает практика, при правильно составленном поисковом запросе требуемые данные возвращаются в первой тройке-пятерке результатов.

Теперь, когда мы в общем виде представляем себе, как работает полнотекстовый поиск, приступим к первой части необходимых действий – созданию полнотекстового индекса.

Далее с помощью стандартного механизма платформы мы будем собственно выполнять полнотекстовый поиск, используя созданный нами индекс.

Что такое роль

Для описания подобных разрешений используются объекты конфигурации *Роль*. С помощью такого объекта разработчик получает возможность описать набор прав на выполнение тех или иных действий над каждым объектом базы данных и над всей конфигурацией в целом.

Как правило, роли создаются отдельно для каждого вида деятельности, и каждому пользователю системы ставится в соответствие одна или несколько ролей.

Если пользователю поставлено в соответствие несколько ролей, предоставление доступа будет осуществляться по следующему алгоритму:

- если хотя бы в одной роли есть разрешение, то доступ будет открыт;
- если во всех ролях разрешение отсутствует, то доступ будет закрыт.

Создание ролей

При создании ролей исходят, как правило, из того, какие полномочия требуются различным группам пользователей на доступ к информации. Для этого мы воспользуемся подсистемами, которые значительно облегчат нашу задачу.

Администратор

Первая роль, которую мы создадим, будет Администратор. Она должна включать в себя полные права на работу с данными информационной базы.

Раскроем ветвь Общие дерева объектов конфигурации. Выделим строку Роли и добавим новый объект конфигурации Роль. Зададим его имя – Администратор (рис. 22.1).

Рис. 22.1. Создание роли

Откроется окно редактирования прав этой роли (рис. 22.2).

Слева, в списке объектов, перечислены все объекты и виды объектов конфигурации, а справа, в окне прав, – доступные права для выбранного объекта или видов объектов конфигурации.

Рис. 22.2. Окно редактирования прав для роли «Администратор»

Администратор должен иметь права на все объекты и все виды объектов. Для этого выполним команду Действия Установить все права в командной панели окна.

После этого все права для всех объектов будут помечены.

Однако можно поставить или снять отметку для прав конкретного объекта конфигурации, пользуясь кнопками Отметить все элементы и Снять отметку со всех элементов, расположенными над окном прав.

Теперь единственное, что следует сделать, – снять разрешение на интерактивное удаление для всех объектов. Это необходимо для того, чтобы администратор случайно не мог удалить какой-либо объект базы данных. Для этого пройдемся по всем видам объектов конфигурации (Справочники, Документы и т. д.) и снимем отметку с команды Интерактивное удаление. Заметьте, что одновременно с отключением права на интерактивное удаление объектов снимается также отметка с права Интерактивное удаление предопределенных (см. рис. 22.2).

Для того чтобы наш Администратор мог работать с объектами, которые мы будем создавать после расстановки прав, зададим для него параметр Устанавливать права для новых объектов (см. рис. 22.2).

На этом создание роли Администратор закончено.

Директор

Следующей ролью, которую мы создадим, будет роль Директор.

Создадим новый объект конфигурации Роль с именем Директор.

Нас устраивает, что у новой роли нет прав на доступ ко всем объектам, за исключением тех видов объектов конфигурации, для которых не создано ни одного объекта. Для таких видов объектов конфигурации останутся установленными полные права.

Убедимся, что право Вывод для всей конфигурации у этой роли установлено.

Теперь нам останется лишь пройти по видам объектов конфигурации и установить для них право Просмотр (права Чтение и Использование при этом установятся автоматически).

Затем раскроем ветвь Общие, выделим ветвь Подсистемы и отметим право Просмотр у всех подсистем. Тем самым мы предоставим директору возможность просматривать все данные информационной базы, а позднее с помощью установки видимости команд по ролям мы исключим из его интерфейса все действия, которые по логике нашей конфигурации не относятся к прикладной ее части (рис. 22.3).

Вторая роль нашей конфигурации готова.

Мастер

Следующая роль, которую мы создадим, будет роль Мастер. Снова добавим новый объект конфигурации Роль с именем Мастер. Выполним команду Действия Установить по подсистемам… и выберем подсистемы УчетМатериалов и ОказаниеУслуг. Нажмем Установить.

В результате будут установлены все права на объекты конфигурации, относящиеся к данным подсистемам.

Если теперь установить фильтр объектов по подсистемам УчетМатериалов и ОказаниеУслуг, то можно при необходимости внести уточнения в установленные права (рис. 22.4).

Рис. 22.4. Установка фильтра по подсистеме

В частности, для справочника Сотрудники мы запретим права Добавление, Изменение и Удаление.

Обратите внимание, что при запрете права Добавление исчезла отметка и у права Интерактивное добавление, так как оно является «уточнением» права Добавление. Точно так же уточненные права запрещаются и при отмене прав на изменение и удаление.

Кроме этого, мы снова снимем разрешения на интерактивное удаление для всех объектов базы данных. Для этого пройдем по всем видам объектов конфигурации и снимем у всех право Интерактивное удаление.

Затем снимем фильтр и установим все права, кроме интерактивного удаления для следующих объектов конфигурации:

- справочник ВариантыНоменклатуры,
- справочник ДополнительныеСвойстваНоменклатуры,
- план видов характеристик СвойстваНоменклатуры, регистр сведений ЗначенияСвойствНоменклатуры.

Эти объекты мы не привязывали ни к каким подсистемам, но они будут нужны для работы с характеристиками номенклатуры.

В заключение раскроем ветвь Общие, выделим ветвь Подсистемы и отметим право Просмотр у подсистемы Предприятие. Тем самым мы предоставим доступ к нормативно-справочной информации, которая будет находиться в этой подсистеме. А ненужную мастерам функциональность скроем с помощью видимости команд по ролям.

Роль Мастер готова.

Расчетчик

В заключение нам с вами осталось создать две роли: Бухгалтер и Расчетчик.

Мы разделим права по расчету зарплаты и по ведению бухгалтерского учета.

Дело в том, что в ООО «На все руки мастер» есть бухгалтер и помощник бухгалтера. Помощник бухгалтера занят в основном расчетом зарплаты, но иногда это делает и главный бухгалтер.

Поэтому главному бухгалтеру необходимо будет назначить обе роли, в то время как помощнику – только роль Расчетчик.

Создадим новый объект конфигурации Роль с именем Расчетчик.

В окне редактирования прав установим их по подсистеме РасчетЗарплаты (и не забудем запретить интерактивное удаление). А также установим право Просмотр для объекта конфигурации: Регистр накопления Продажи и справочника Клиенты.

В заключение установим право Просмотр у подсистемы Предприятие.

Роль Расчетчик готова.

Бухгалтер

В заключение создадим объект конфигурации Роль с именем Бухгалтер. В окне редактирования прав установим их по подсистеме Бухгалтерия.

После этого отфильтруем список объектов по этой подсистеме и для справочника Номенклатура запретим добавление, изменение и удаление.

Также запретим интерактивное удаление для всех объектов.

Затем снимем фильтр и установим все права, кроме интерактивного удаления для объекта конфигурации Справочник Субконто.

А также установим право Просмотр для следующих объектов конфигурации:

- Справочник Склады,
- Справочник ВариантыНоменклатуры,
- Справочник ДополнительныеСвойстваНоменклатуры,
- План видов характеристик СвойстваНоменклатуры, Регистр сведений ЗначенияСвойствНоменклатуры.

В заключение установим право Просмотр у подсистемы Предприятие.

Занятие 1

1. Используя оператор if, вычислите заданное выражение:

$$f(x)=\begin{cases} x^2+3 & ,x<0 \\ 6\sqrt{x} & ,0\le x\le 5 \\ -x+9 & ,x>5 \end{cases}$$

2. Стоимость акций компании Дженерал - Моторс от 0 до 100 долларов (машина генерирует два случайных числа). Вы не знаете колебаний стоимости акций. Машина спрашивает, будете ли вы продавать или покупать акции. Если вы продаете, а цены поднялись – вы банкрот, если покупаете, а цены падают – тоже. В противном случае богатеете на разность стоимости. Составьте программу, организующую эту торговую
сделку и печатающую результат.

Занятие 2

1. Используя оператор выбора, напишите программу решения следующей задачи: Арифметические действия над числами пронумерованы следующим образом: 1 — сложение, 2 — вычитание, 3 — умножение, 4 — деление. Дан номер действия и два числа A и B (B не равно нулю). Выполнить над числами указанное действие и вывести результат.
2. Даны действительные числа a, b, c, x, y. Выясните, пройдет ли кирпич с ребрами a,b,c в прямоугольное отверстие со сторонами x и y. Просовывать кирпич в отверстие разрешается только так, чтобы каждое из его ребер было параллельно или перпендикулярно каждой из сторон отверстия.

Занятие 3

Создайте проект, предназначенный для решения задачи. Ввод данных осуществите с помощью элемента управления текстовое поле, вывод – с помощью элемента управления метка, вычисление должно выполняться при щелчке по кнопке.

- Решение уравнений вида ax+b=0 для любых действительных a, b. Коэффициенты вводятся пользователем, программа выдает корень (в виде x=4.56) или сообщение, что решений нет или бесконечно много.
- Решение квадратных уравнений с целыми коэффициентами. Коэффициенты вводятся пользователем, после чего программа выводит корни (в виде x1=-6, x2=-1) или сообщение, что действительных корней нет.
- Найти количество точек с целочисленными координатами, лежащих внутри круга радиуса R с центром в начале координат.

Занятие 4

Создайте проект, предназначенный для решения задачи. Ввод данных осуществите с помощью элемента управления текстовое поле, вывод – с помощью элемента управления метка, вычисление должно выполняться при щелчке по кнопке.

- Из пунктов A и B навстречу друг другу выехали два автомобиля. Известно расстояние между пунктами, скорость каждого автомобиля и время, которое они пробыли в пути (вводятся пользователем). Выяснить какое расстояние будет между ними (учесть, что до встречи расстояние уменьшается, а после – увеличивается).

- Из трех чисел найти среднее по величине, например, для 6, 5 и 9 это число 6.
- «Кассовый аппарат». Имеются два поля для ввода цены товара и его количества, надпись с общей стоимостью покупок, кнопка «+», позволяющая вычислять стоимость очередного товара и добавлять ее к предыдущему значению суммы стоимостей, а также кнопка «Сброс», обнуляющая сумму.

Занятие 5

Создайте проект, предназначенный для решения задачи. Ввод данных осуществите с помощью элемента управления текстовое поле, вывод – с помощью элемента управления метка, вычисление должно выполняться при щелчке по кнопке.

- По длинам сторон треугольника выяснить прямоугольный он, остроугольный или тупоугольный.
- «Ремонт». Известны размеры комнаты, в которой планируется покрасить пол и стены, а также расход краски (мл/кв.м.). Краска продается в банках емкостью 3 л. Необходимо рассчитать сколько банок необходимо купить для покраски пола, а сколько для покраски стен.
- По длинам трех отрезков выяснить, можно ли построить треугольник с такими сторонами. Если треугольник построить можно, то вычисляется его площадь и выводится ответ в виде S=12.

Занятие 6

Создайте проект, предназначенный для решения задачи. Ввод данных осуществите с помощью элемента управления текстовое поле, вывод – с помощью элемента управления метка, вычисление должно выполняться при щелчке по кнопке.

- По длинам трех отрезков a, b и c выяснить, можно ли построить треугольник со сторонами a, b, c и, если да, то какой - равносторонний, равнобедренный или произвольный.
- Школьник сел делать уроки, когда на часах было h1 часов, m1 минут, s1 секунд и закончил, когда часы показывали h2 часов, m2 минут, s2 секунд. Определить, сколько времени в часах, минутах и секундах школьник делал уроки, если за полночь он не засиделся.
- По номеру года выяснить, является он високосным или нет.

Занятие 7

1. Составьте таблицу сложности выполнения основных операций (добавление элементов, удаление элементов для структурами: вектор, стек, очередь, дек).
2. По заданному текстовому файлу получите новый текстовый файл, в котором слова из первого файла расположены в обратном порядке.
3. Для данной матрицы прямоугольности растрового изображения, содержащей номера цветов пикселей, реализуйте функцию закраски связной области. На вход программа получает размеры матрицы, элементы матрицы в виде чисел [0;255] – номера цветов точек, координаты точки (i,j), с которой начинается заливка связной области и цвет заливки. Программа должна вывести матрицу после заливки.

Занятие 8

1. Реализуйте дек и методы работы с ним при помощи статического массива.
2. Реализуйте при помощи векторов метод Ван Херка определения минимального на всех подотрезках фиксированной длины.
3. Для задачи вычисления биномиальных коэффициентов Cm укажите базу динамики, правило перехода динамики и рассмотрите пример заполнения таблицы динамики для конкретного значения n и m.
4. Укажите способ восстановления ответа в задаче о рюкзаке.
5. Для задачи о рюкзаке рассмотрите пример заполнения таблицы динамики для тестово- го примера, приведенного в задаче лекции.

Занятие 9

1. Укажите решение задачи о рюкзаке, если предметы можно помещать в рюкзак неогра- ниченное количество раз.
2. Классическая задача на вычисления расстояния редактирования предполагает опреде- ление минимального количества вставок, удалений и замен символов, необходимое для преобразования строки s1 в строку s2 (так называемое расстояние Левенштейна). Предложите способ вычисления расстояния Левенштейна для двух строк.
3. Опишите алгоритм генерации всех m-сочетаний с повторениями из множества первых n нату- ральных чисел.
4. Опишите алгоритм генерации всех m-размещений с повторениями из множества первых n натуральных чисел.
5. Опишите алгоритмы получения сочетания по его номеру и, наоборот, номера сочетания по са- мому сочетанию.
6. Опишите алгоритмы получения размещения по его номеру и, наоборот, номера сочетания по самому сочетанию.

Занятие 10

1. Изобразите дерево рекурсивных вызовов для рекурсивного вычисления степени алгоритмом сложности O(log n). В качестве конкретного примера возьмите вычисление a150.
2. Проведите триангуляцию выпуклого шестиугольника – разбиение его непересекающимися диа- гоналями на треугольники. Подсчитайте количеств различных вариантов. Вычислите это коли- чество при помощи соответствующего числа Каталана.
3. Предложите решение следующей задачи. Даны шахматные доски с размером 2, 3, 4,…, в кото- рых закрашены все квадраты северо-западнее главной диагонали. Требуется провести ладью из левого нижнего угла в правый верхний. Причем, двигаться можно только на север и на восток, не заходя при этом на закрашенные клетки. Спрашивается, сколько существует различных пу- тей ладьи на доске со стороной n?

Занятие 11

1. В графе G n вершин и Δ(G)=k. Какое наибольшее количество компонент связности может быть в G?
2. Сколько ребер в графе $K_{m,n}$? Запишите формулу для любых n и m.
3. Эйлеровым циклом в неориентированном графе называется замкнутый путь, проходящий по всем ребрам графа ровно по одному разу. Докажите, что в неориентированном графе эйлеров цикл есть тогда и только тогда, когда он связен и степени всех вершин четны.

   Доказать, что неориентированный граф является двудольным тогда и только тогда, когда в нем нет циклов нечетной длины.

Занятие 12

В системе «1С: Предприятие (учебная версия)» создайте новую информационную базу с именем «На все руки мастер».

- Откройте конфигурацию. Задайте ей имя – НаВсеРукиМастер.
- Создайте подсистемы: Бухгалтерия, РасчетЗарплаты, УчетМатериалов, ОказаниеУслуг, Предприятие. К каждой подсистеме привяжите соответствующую картинку из папки Image.

Занятие 13

В системе «1С: Предприятие (учебная версия)» откройте информационную базу «На все руки мастер» созданную на предыдущем занятии.

- Измените порядок вывода подсистем предприятия: УчетМатериалов, ОказаниеУслуг, Бухгалтерия, РасчетЗарплаты, Предприятие.
- Создайте справочники:
  - Список клиентов (имя – Клиенты, синоним – Клиенты, представление объекта – Клиент, представление списка – Клиенты, подсистемы – ОказаниеУслуг, Бухгалтерия, длина кода – 9, длина наименования – 50). Для подсистемы ОказаниеУслуг добавьте опцию Создать на панель действий. В режиме 1С: Предприятие введите клиентов: Иванов Михаил Юрьевич; Спиридонова Галина Сергеевна; Павлов Роман Иванович.
  - Справочник «Сотрудники» с табличной частью, содержащий список сотрудников предприятия и информацию об их предыдущей профессиональной деятельности. Имя справочника – Сотрудники, синоним – Сотрудники, представление объекта – Сотрудник, представление списка – Сотрудники, подсистемы – ОказаниеУслуг, РасчетЗарплаты, длина кода – 9, длина наименования – 50. Имя табличной части – ТрудоваяДеятельность. Реквизиты табличной части – организация, начало работы, окончание работы, должность. Для подсистемы РасчетЗарплаты добавьте опцию Создать на панель действий. В режиме 1С: Предприятие введите сотрудников: Гусаков Николай Дмитриевич (организация – ЗАО «НТЦ», начало работы – 01.02.2000, окончание работы – 16.04.2003, должность – ведущий специалист); Деловой Иван Сергеевич (организация 1 – ООО «Автоматизация», начало работы – 22.01.1996, окончание работы – 31.12.2002, должность – Инженер; организация 2 – ЗАО «НПО СпецСвязь», начало работы – 20.06.1986, окончание работы – 21.01.1995, должность – Начальник производства); Симонов Валерий

Михайлович (организация – ООО «СтройМастер», начало работы – 06.02.2001, окончание работы – 03.04.2004, должность – Прораб).

Занятие 14

В системе «1С: Предприятие (учебная версия)» откройте информационную базу «На все руки мастер» созданную на предыдущем занятии.
Создайте справочники:

- Иерархический справочник «Номенклатура», содержащий перечень услуг, который оказывает ООО «На все руки мастер» и материалах, которые при этом могут быть использованы, справочник будет содержать две группы – услуги и материалы. Имя – Номенклатура. Подсистемы – УчетМатериалов, Оказание Услуг, Бухгалтерия. Вид иерархии – иерархия групп и элементов. Для подсистем УчетМатериалов и ОказаниеУслуг добавьте опцию Создать на панель действий для элементов и групп. В режиме 1С: Предприятие: Создайте группы: Услуги, Материалы. Для этих групп реквизит Родитель заполнять не следует.В группе Материалы создайте элементы: Строчный трансформатор Samsung, Строчный трансформатор GoldStar, Транзистор Phillips 2N2369, Шланг резиновый, Кабель электрический.В группе Услуги создайте элементы: Диагностика, Ремонт отечественного телевизора, Ремонт импортного телевизора, Подключение воды, Подключение электричества. Разбейте Услуги по двум смысловым группам: Телевизоры и Стиральные машины. Для этого в группе Услуги создайте новые подгруппы, для них реквизит Родитель должен содержать – Услуги. В подгруппу Телевизоры перетащите элементы: Диагностика, Ремонт отечественного телевизора, Ремонт импортного телевизора. В подгруппу Стиральные машины перетащите остальные элементы. Выберите режим просмотра созданного справочника в виде Дерева (Все действия – Режим просмотра – Дерево).
- Справочник «Список складов», на которых могут находиться материалы, с предопределенным элементом Основной (это основной склад, на который вначале поступают все материалы). Имя справочника – Склады, синоним – Склады, представление объекта – Склад, представление списка – Склады, подсистемы – ОказаниеУслуг, УчетМатериалов, длина кода – 9, длина наименования – 50. Для справочника на вкладке Формы установите флажок Быстрый выбор, который позволит отображать раскрывающийся список складов. Задайте предопределенный элемент – Основной. Для подсистемы УчетМатериалов добавьте опцию Создать на панель действий. В режиме 1С: Предприятие добавьте Розничный склад в справочник складов.

Средства контроля качества обучения

Вопросы к зачету:

1. Основные элементы объектной модели.
2. Отношения между объектами и классами.
3. Абстрагирование.
4. Инкапсуляция.
5. Наследование.
6. Полиморфизм.
7. Особенности платформы .NET.
8. Классы с C#, поля класса.
9. Указание области видимости: public; private; protected; internal.
10. Статические поля.
11. Константы.
12. Методы класса: синтаксис; тип возвращаемого значения; список формальных параметров.
13. Модификаторы доступа к методам: public, private, protected, internal, static.
14. Методы с переменным числом параметров.
15. Определение выходных параметров метода.
16. Вызов метода.
17. Рекурсивные методы.
18. Конструкторы.
19. Деструкторы.
20. Перегрузка методов.
21. Перегрузка операторов.
22. Свойства класса.
23. Индексаторы класса.
24. Функциональные типы в C#, делегаты.
25. Функциональные типы в C#, события.
26. Наследование.
27. Виртуальные функции.
28. Абстрактные классы.
29. Создание иерархии исключений.
30. Обобщения, основные понятия.
31. Уточнения, используемые в обобщениях.
32. Обобщенные интерфейсы.
33. Обобщенные методы.
34. Обобщенные делегаты.

Вопросы экзамену:

1. Понятие алгоритма. Свойства алгоритмов. Способы описания алгоритмов.
2. Базовые алгоритмические конструкции. Язык программирования.
3. Типы данных: понятие, простые типы данных, примеры типов, совместимость и преобразование типов. Переменные, операции и выражения.
4. Операторы. Структура программы. Организация ввода и вывода.

5. Операторы ветвления. Операторы циклов.
6. Указатели. Динамические переменные. Массивы.
7. Строковый тип данных. Структуры. Объединения.
8. Функции. Файлы.
9. Технологии разработки приложений с графическим интерфейсом. Основные свойства форм.
10. Добавление форм. Взаимодействие между формами. События форм.
11. Стандартные элементы управления, их основные свойства, основные методы, основные события.
12. Элементы управления для ввода/вывода. Переключатели.
13. Счетчик. Списки. Стандартные диалоги. Меню и панели инструментов.
14. Линейные списки. Стеки. Очереди. Деки. Основные операции, способы представления.
15. Типы данных, структуры данных, абстрактные типы данных.Деревья. Основные понятия.
16. Создание, обход, представление деревьев. Пример использования деревьев: код Хаффмана.
17. Графы. Основные понятия. Способы представления графов. Путь с наименьшим количеством дуг в графе.
18. Волновой обход графов. Путь кратчайшей длины в графе. Алгоритм Дейкстры.
19. Кратчайший путь между парами вершин графа. Алгоритм Флойда.
20. Обход графов. Метод поиска в глубину. Глубинный остовный лес графов.
21. Циклы в графах. Алгоритм нахождения циклов в графе.
22. Сильная связность. Нахождение компонент сильной связности в графе. Хроматическое число графов. Нахождение хроматического числа графов.
23. Внешняя сортировка. Сортировка слиянием. Метод полного перебора. Перебор циклами, рекурсивный перебор.
24. Метод полного перебора. Полный p-ичный перебор. Динамическое программирование. Принцип оптимальности Беллмана. "Жадные" алгоритмы. Метод ветвей и границ.
25. Задача сортировки. Метод прямого выбора. Метод прямого включения. Метод прямого обмена.
26. Улучшенная сортировка. Метод Шелла. Шейкерная сортировка.
27. Улучшенная сортировка. Пирамидальная сортировка. Быстрая сортировка. "Карманная" сортировка.
28. Задача поиска. Последовательный поиск. Поиск в упорядоченной таблице. Бинарный поиск. Поиск по бинарному дереву. Дерево поиска.
29. АВЛ-деревья. Хеш-таблицы. Открытое хеширование. Хеш-таблицы. Прямое хеширование. Методы разрешения коллизий.
30. Эффективность алгоритмов. NP-полные и трудно решаемые задачи.
31. 1С:Предприятие. Приложения «1С» и их назначение, типологизация. Платформа «1С:Предприятие» как средство разработки бизнес-приложений.
32. 1С:Предприятие. Конфигурация и прикладное решение. Режимы работы системы «1С:Предприятие».Создание новой информационной базы в системе «1С:Предприятие».
33. 1С:Предприятие. Возможности режима «Конфигуратор». Возможности режима

«1С:Предприятие». Работа с подсистемами.

34. 1С:Предприятие. Справочники. Документы. Механизм основных форм.
35. 1С:Предприятие. Модули. Форма как программный объект. Обработчики событий в модуле формы.
36. 1С:Предприятие. Анализ кода с помощью синтакс-помощника. Работа с объектами. Сервер и клиенты.
37. 1С:Предприятие. Директивы компиляции. Исполнение кода на клиенте и на сервере. Регистры накопления.
38. 1С:Предприятие.Работа с отчетами. Макеты. Редактирование макетов и форм.
39. 1С:Предприятие. Периодический регистр сведений. Перечисления. Отчеты.
40. 1С:Предприятие. План видов характеристик. Бухгалтерский учет. План видов расчета, регистр расчета.
41. 1С:Предприятие. Организация поиска. Выполнение заданий по расписанию. Редактирование движений в форме документа.
42. 1С:Предприятие. Список пользователей и их роли. Настройка командного интерфейса.
43. 1С:Предприятие. Обмен данными. Функциональные опции. Типовые приемы разработки.
44. 1С:Предприятие. Приемы разработки форм. Приемы редактирования форм.