




Гусин А.Н., Викентьева О.Л.

Проектирование программ и программирование на C++

Часть I: Структурное программирование

Структурное программирование

- Структурное программирование – это технология создания программ, позволяющая путем соблюдения определенных правил, уменьшить время разработки, количество ошибок, а также облегчить возможность модификации программы.
 - Программу можно составить только из трех структур: линейной, разветвляющейся и циклической. Эти структуры называются базовыми конструкциями структурного программирования.
- **Линейной** называется конструкция, представляющая собой последовательное соединение двух или более операторов.
 - **Ветвление** задает выполнение одного из двух операторов, в зависимости от выполнения какого либо условия.
 - **Цикл** задает многократное выполнение оператора.



Технология создания программ

- **Проектирование программы**
- Структурный подход к программированию охватывает все стадии разработки проекта: спецификацию, проектирование, собственно программирование и тестирование.
- Задачи, которые при этом ставятся – это уменьшение числа возможных ошибок за счет применения только допустимых структур, как можно более раннее обнаружение ошибок и упрощение процесса их исправления.
- Ключевыми идеями структурного подхода являются структурное программирование и нисходящее тестирование.

■ Этапы создания программы:

1. **Постановка задачи.** Изначально задача ставится в терминах предметной области, и необходимо перевести ее в термины более близкие к программированию. Это достаточно трудоемкий процесс, т. к. программист обычно плохо разбирается в предметной области, а заказчик не может правильно сформулировать свои требования. Постановка задачи завершается созданием технического задания и внешней спецификации программы.
- Спецификация программы должна включать в себя:
 - описание исходных данных и результатов;
 - описание задачи, реализуемой программой;
 - способ обращения к программе;
 - описание возможных аварийных ситуаций и ошибок пользователя.
2. **Разработка внутренних структур данных.** Большинство алгоритмов зависит от способа организации данных (статические или динамические, массивы, списки или деревья и т. п.).

3. **Проектирование программы**, которое заключается в определении общей структуры и способов взаимодействия модулей. На данном этапе может применяться технология нисходящего проектирования, при котором задачу разбивают на подзадачи меньшей сложности. На этом этапе очень важной является спецификация интерфейсов, т. е. способов взаимодействия подзадач. Для каждой подзадачи составляется внешняя спецификация, аналогичная п. 1. На этом же этапе решаются вопросы разбиения программы на модули, взаимодействие этих модулей должно быть минимальным. На более низкий уровень проектирования переходят только после окончания проектирования верхнего уровня. Алгоритмы для модулей записывают в обобщенной форме (словесная запись, блок-схемы). Проектировать программу надо таким образом, чтобы в нее достаточно легко можно было внести изменения. Процесс проектирования является итерационным, т. к. невозможно учесть все детали с первого раза.

- 4. Структурное программирование.** Процесс программирования также должен быть организован сверху вниз: сначала кодируются модули самого верхнего уровня и составляются тестовые примеры для их отладки, на месте модулей, которые еще не написаны, ставятся, так называемые "заглушки". Заглушки выдают сообщение о том, что им передано управление, а затем снова возвращают управление в вызывающую программу. При программировании следует отделять интерфейс модуля от его реализации и ограничивать доступ к ненужной информации. Этапы проектирования и программирования совмещены во времени: сначала проектируется и кодируется верхний уровень, затем – следующий и т. д. Такая стратегия применяется, т. к. в процессе кодирования может возникнуть необходимость внести изменения, которые потом отразятся на модулях нижнего уровня.

5. **Нисходящее тестирование.** Проектирование и программирование сопровождаются тестированием. Цель процесса тестирования – определение наличия ошибки, нахождение места ошибки, ее причины и соответствующие изменения программы – исправление. Тест – это набор исходных данных, для которых заранее известен результат. Тест, выявивший ошибку, считается успешным. Процесс исправления ошибок в программе называется отладкой, исправляются ошибки обнаруженные при тестировании. Отладка программы заканчивается, когда достаточное количество тестов выполнилось неуспешно, т. е. программа на них выдала правильные результаты.

Цель тестирования показать, что программа работает правильно и удовлетворяет всем проектным спецификациям. Чем больше ошибок обнаружено на начальных стадиях тестирования, тем меньше их остается в программе. Чем меньше ошибок осталось в программе, тем сложнее искать каждую из этих ошибок. Идея нисходящего тестирования заключается в том, что к тестированию программы надо приступить еще до того, как завершено ее проектирование. Только после того как проверен и отлажен один уровень программы, можно приступить к программированию и тестированию следующего уровня.

- Для исчерпывающего тестирования рекомендуется проверить:
 - каждую ветвь алгоритма;
 - граничные условия;
 - ошибочные исходные данные.

- **Кодирование и документирование программы**
- Главная цель, к которой нужно стремиться при написании программы – это получение легко читаемой программы простой структуры. Для этого написание программы рекомендуется начинать с записи на естественном языке или в виде блок-схем ее укрупненного алгоритма (что и как должна делать программа). Алгоритм надо записать как последовательность законченных действий. Каждое законченное действие оформляется в виде функции. Каждая функция должна решать одну задачу. Тело функции не должно быть длинным (30-50 строк), т. к. сложно разбираться в длинной программе, которая содержит длинные функции. Если некоторые действия повторяются более одного раза, их тоже рекомендуется оформить как функцию. Короткие функции лучше оформить как подставляемые функции (`inline`).
- Имена переменных выбираются таким образом, чтобы можно было понять, что делает эта переменная, например, сумму обозначают `sum`, `summa` или `s`, массив – `array` или `arr` и т. п. Для счетчиков коротких циклов лучше использовать однобуквенные имена, например, `i` или `j`. Чем больше область видимости переменной, тем более длинное у нее имя. Не рекомендуется использовать имена, начинающиеся с символа подчеркивания, имена типов, идентификаторы, совпадающие с именами стандартной библиотеки C++.

- Переменные рекомендуется объявлять как можно ближе к месту их использования. Но можно и все объявления локальных переменных функции расположить в начале функции, чтобы их легко можно было найти. Переменные лучше инициализировать при их объявлении.
- Глобальные переменные лучше не использовать. Если использование глобальной переменной необходимо, то лучше сделать ее статической, тогда область ее видимости будет ограничена одним файлом.
- Информация, которая необходима для работы функции, должна передаваться ей в качестве параметров, а не глобальных переменных.
- Входные параметры, которые не должны изменяться в функции лучше передавать как ссылки со спецификатором `const`, а не по значению. Этот способ более эффективен, особенно при передаче сложных объектов.
- Выходные параметры лучше передавать по указателю, а не по ссылке, тогда из семантики вызова функции будет понятно, что этот параметр будет изменяться внутри функции.

- Нельзя возвращать из функции ссылку на локальную переменную, т. к. эта переменная будет автоматически уничтожаться при выходе из функции. Также не рекомендуется возвращать ссылку на динамическую локальную переменную, созданную с помощью операции `new` или функции `malloc()`.
- Если в программе используются числа, например, размеры массивов, то для них лучше использовать символические имена – константы или перечисления. Это делает программу более понятной и, кроме того, в такую программу легче будет вносить изменения, т. к. достаточно будет изменить константу в одном месте.
- Следует избегать лишних проверок условий, т. е. если для вычисления отношений надо вызывать одну и ту же функцию несколько раз, то вычисление функции лучше оформить в виде оператора присваивания, а в условном операторе использовать вычисленное значение. Не следует в условном операторе выполнять проверку на неравенство нулю, т. к. это не имеет смысла. Например, условие
`if (ok != 0)` лучше записать как
`if (ok)`
Более короткую ветвь оператора `if` рекомендуют помещать сверху, иначе управляющая структура может не поместиться на экране.

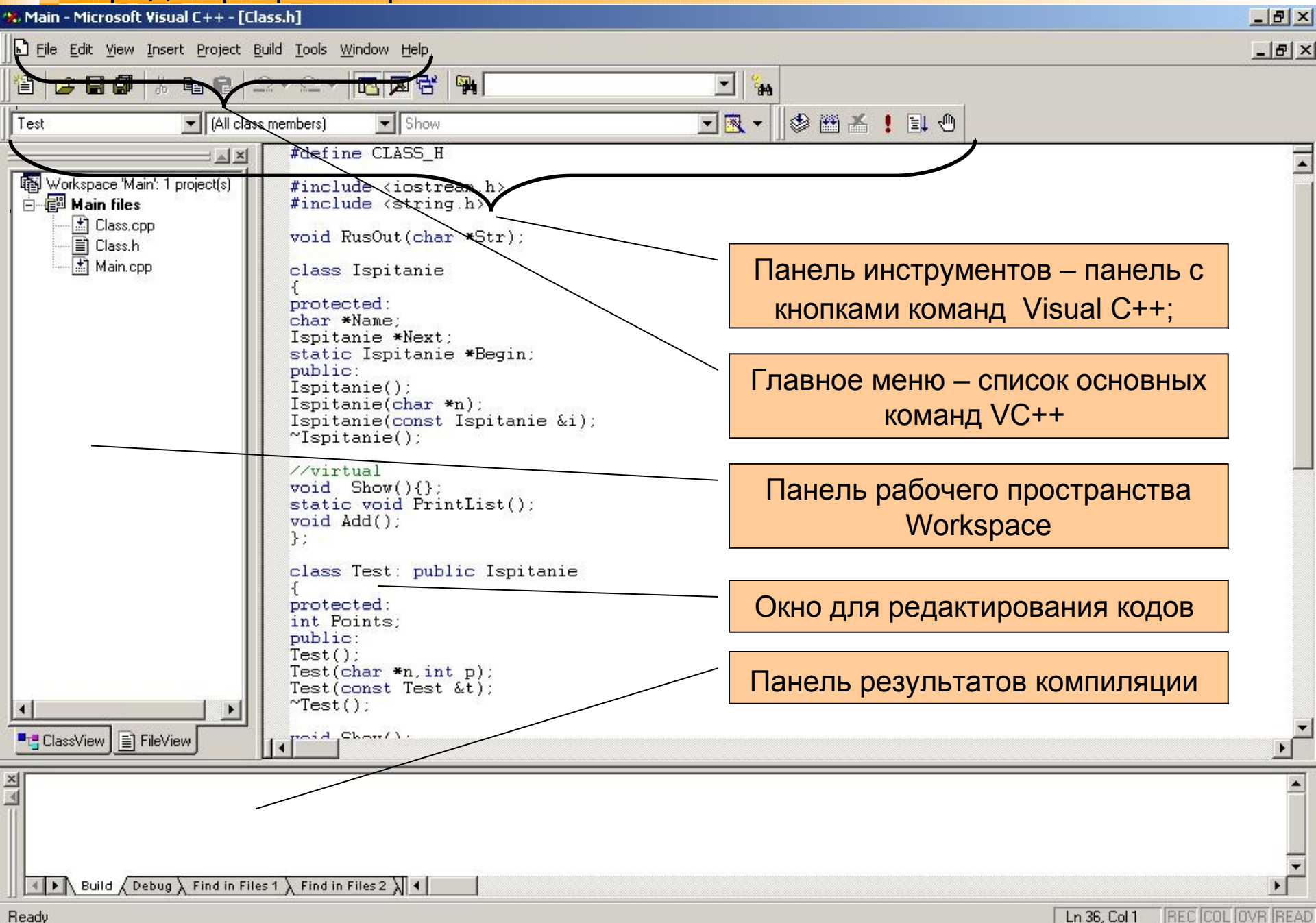
- При использовании циклов надо объединять инициализацию, проверку условия выхода и приращения в одном месте. Если есть хотя бы два из инициализирующего, условного или корректирующего выражения, то лучше использовать цикл `for`. При использовании итеративных циклов необходимо предусматривать выход при достижении максимального количества итераций.
- Необходимо проверять коды возврата ошибок и предусматривать печать соответствующих сообщений. Сообщение об ошибке должно быть информативным и подсказывать пользователю как ее исправить. Например, при вводе неверного значения должен указываться допустимый диапазон.
- Операции выделения и освобождения динамической памяти следует помещать в одну функцию. Иначе может возникнуть ситуация, когда память выделили, а освободить забыли.

- Программа должна иметь комментарии. Комментарии должны представлять собой правильные предложения, но они не должны подтверждать очевидное (за исключением тех случаев, когда программа используется как пример для обучения). Комментарий, который занимает несколько строк, размещают до фрагмента программы. Для разделения функций и других логически законченных фрагментов можно использовать пустые строки или комментарии вида
- Вложенные блоки должны иметь отступы в 3-4 символа, причем блоки одного уровня вложенность должны быть выровнены по вертикали. Закрывающая фигурная скобка должна находиться под открывающей.



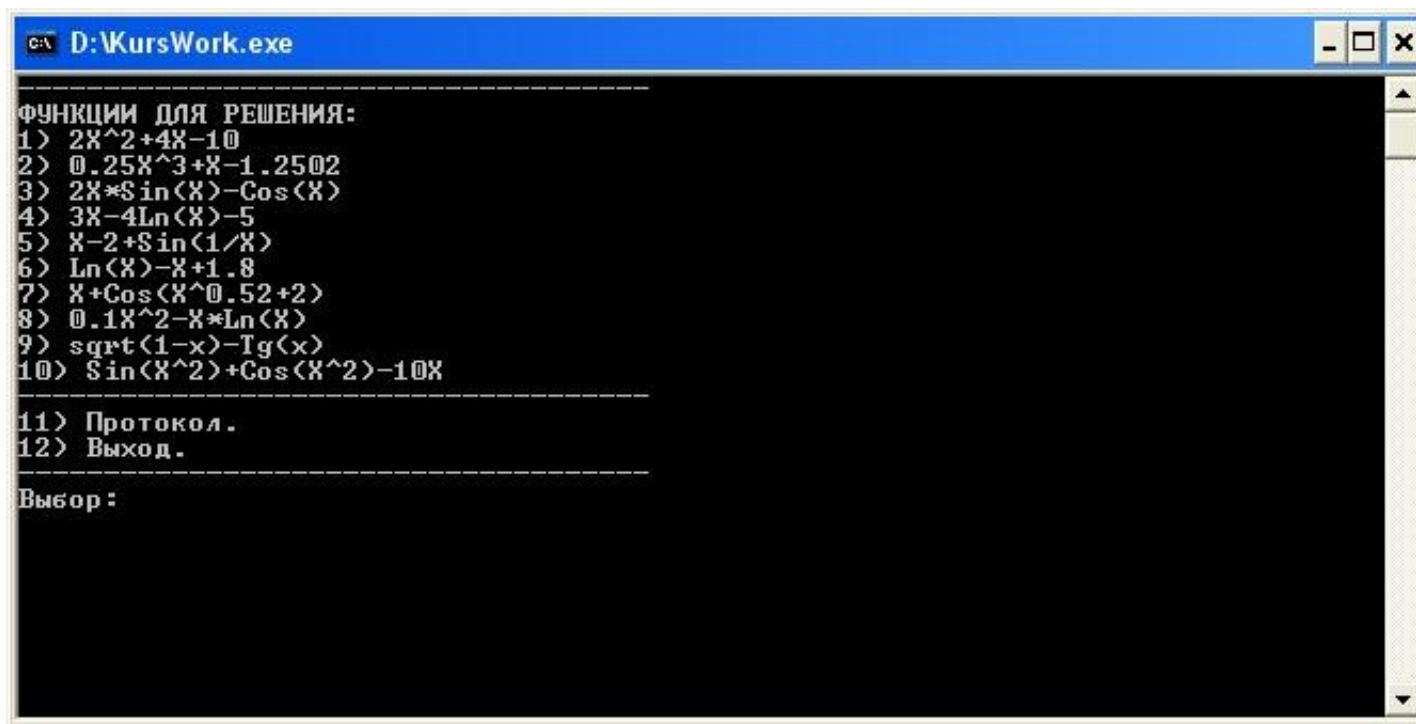
Среда программирования Visual C++ 6.0

Среда программирования Visual C++ 6.0



Среда программирования Visual C++ 6.0

- **Создание консольного приложения и работа с ним**
- Консольное приложение – это приложение, которое с точки зрения программиста является программой DOS, но может использовать всю доступную оперативную память. Этот тип приложения запускается в особом окне, которое называется «Окно MS-DOS».



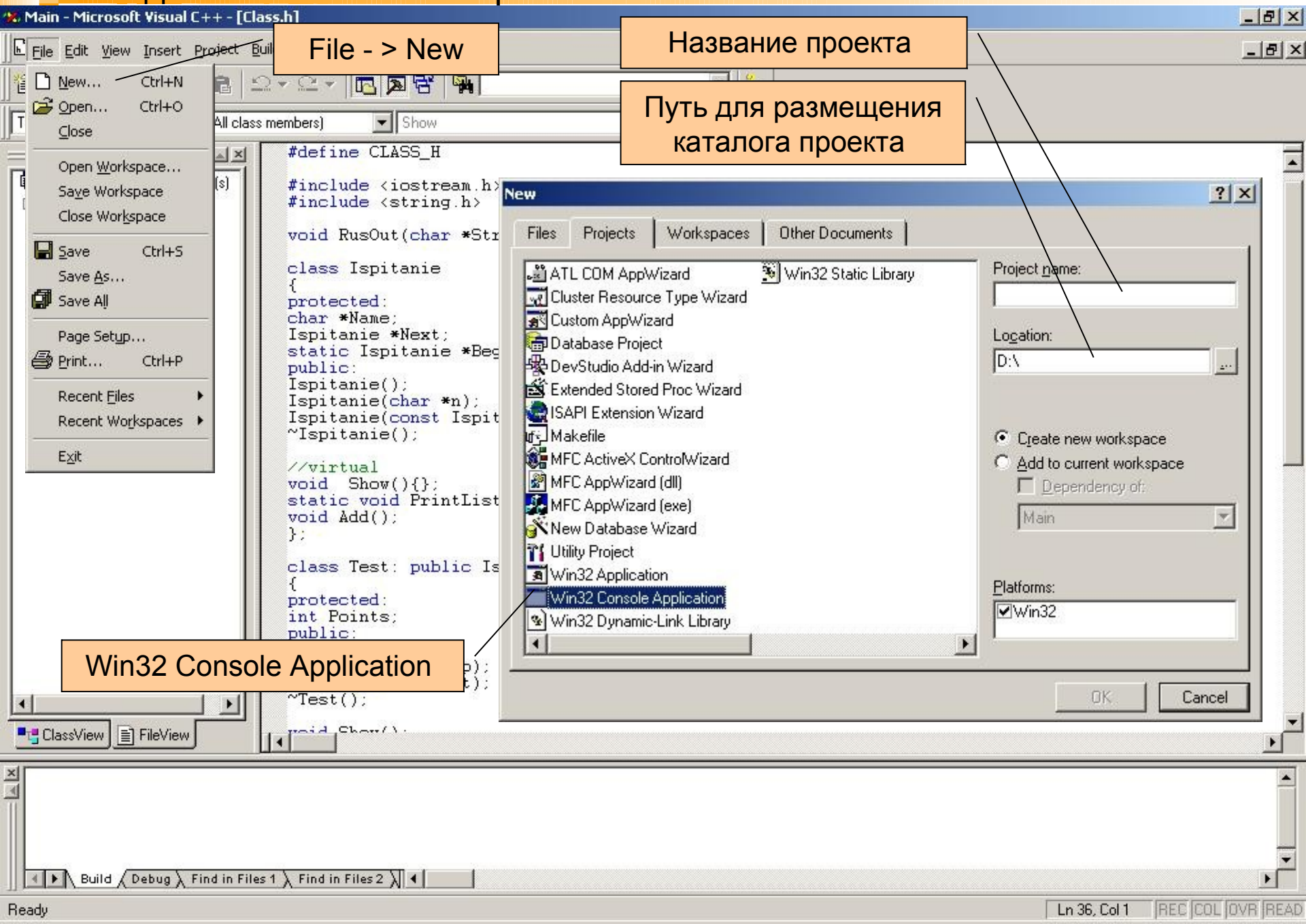
```
C:\ D:\KursWork.exe

-----
ФУНКЦИИ ДЛЯ РЕШЕНИЯ:
1> 2X^2+4X-10
2> 0.25X^3+X-1.2502
3> 2X*Sin(X)-Cos(X)
4> 3X-4Ln(X)-5
5> X-2+Sin(1/X)
6> Ln(X)-X+1.8
7> X+Cos(X^0.52+2)
8> 0.1X^2-X*Ln(X)
9> sqrt(1-x)-Tg(x)
10> Sin(X^2)+Cos(X^2)-10X
-----

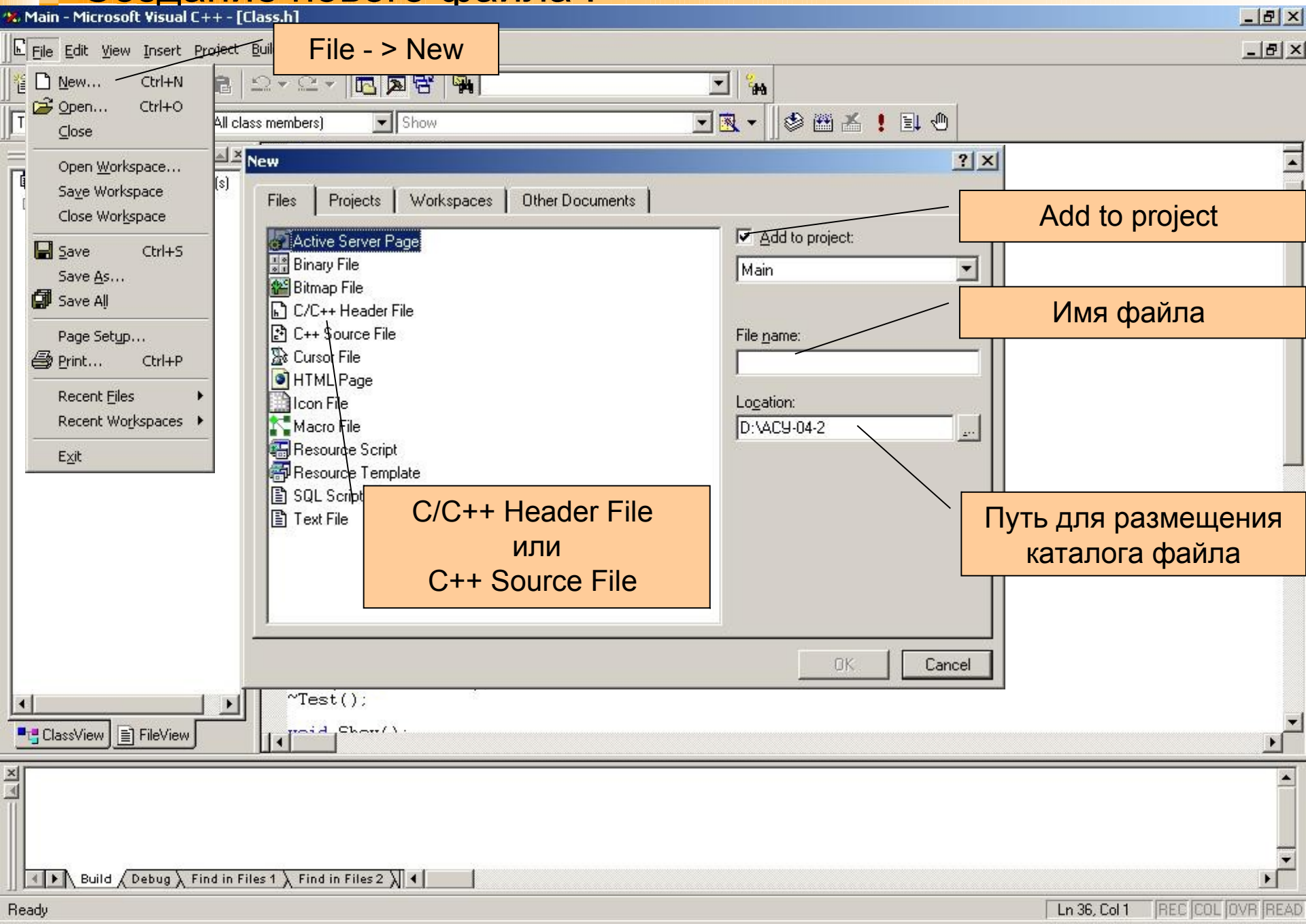
11> Протокол.
12> Выход.
-----

Выбор :
```

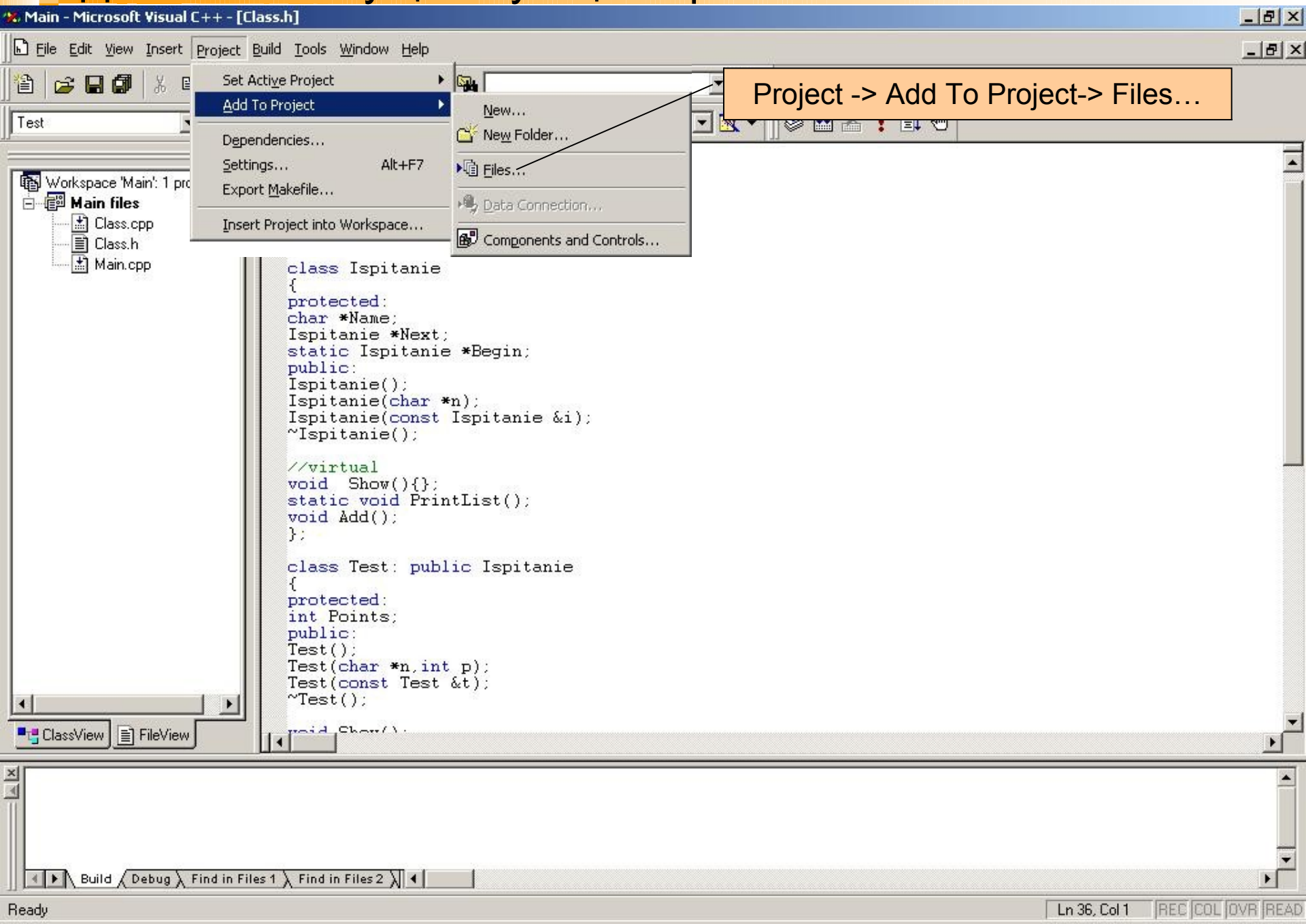

Создание консольного приложения :



Создание нового файла :



Добавление существующего файла :



- **Компиляция и запуск проекта**
- Для компиляции проекта надо выбрать в главном меню **Build -> Build <имя проекта>** или нажать клавишу **F7**.
- Для запуска исполняемого файла надо выбрать в главном меню **Build -> Execute <имя файла>.exe** или нажмите клавиши **Ctrl+F5**.
- Если файл был создан, то он запустится. Для повторного запуска файла не нужно его снова компилировать. Но если в программу были внесены изменения, то перед запуском необходимо выполнить компиляцию.

■ Отладка программы

- Для отладки программы используется команда главного меню **Build->Start Debug-> Step Into** – отладка с заходом в функции, которая начинается с первой строки функции `main` или **Build->Start Debug-> Run to Cursor** – выполнение программы до курсора, т. е. отладка начинается с той строки, в которой установлен курсор.
- Переход к следующей строке программы можно выполнять с помощью команды **Step Into (F11)** (с заходом во все вызываемые функции) или с помощью команды **Step over (F10)** (без захода в вызываемые функции). Выход из функции нижнего уровня выполняется командой **Step Out (Shift+F11)**. Текущие значения переменных можно просматривать:
 - в специальных окнах **Watch** (отображает значения всех используемых переменных) и **Value** (отображает значения заданных пользователем переменных);
 - при наведении курсора мышки на переменную отображается текущее значение этой переменной.

- **Создание рабочего пространства для нескольких проектов**
- Несколько проектов можно объединить в одно рабочее пространство с помощью команды **Project/Insert Project into Workspace**. Активный проект, т. е. тот, который будет выполняться, устанавливается с помощью команды **Project/Set Active Project**. Активный процесс надо отметить галочкой.

Контрольные вопросы

- 1) Как создать новую программу?
- 2) Можно ли объединить несколько проектов в одно рабочее пространство, если да, то как?
- 3) Как откомпилировать и запустить программу(несколькими способами)?



Структура программы на языке C/C++

Структура программы на языке C/C++

```
//Дана последовательность целых чисел из n элементов.  
//Найти среднее арифметическое этой последовательности.
```

Комментарии

```
#include <iostream.h>  
#include <math.h>
```

Директивы препроцессора

```
void main()
```

Заголовок функции, с которой начинается выполнение программы

```
{  
int a,n,i,k=0;  
double s=0;
```

Определения переменных, используемых в функции

```
cout<<"\nEnter n ";  
cin>>n;  
for(i=1;i<=n;i++)
```

Ввод-вывод данных

```
{  
cout<<"\nEnter a";  
cin>>a;  
s+=a;  
k++;  
}
```

Тело цикла

```
s=s/k;
```

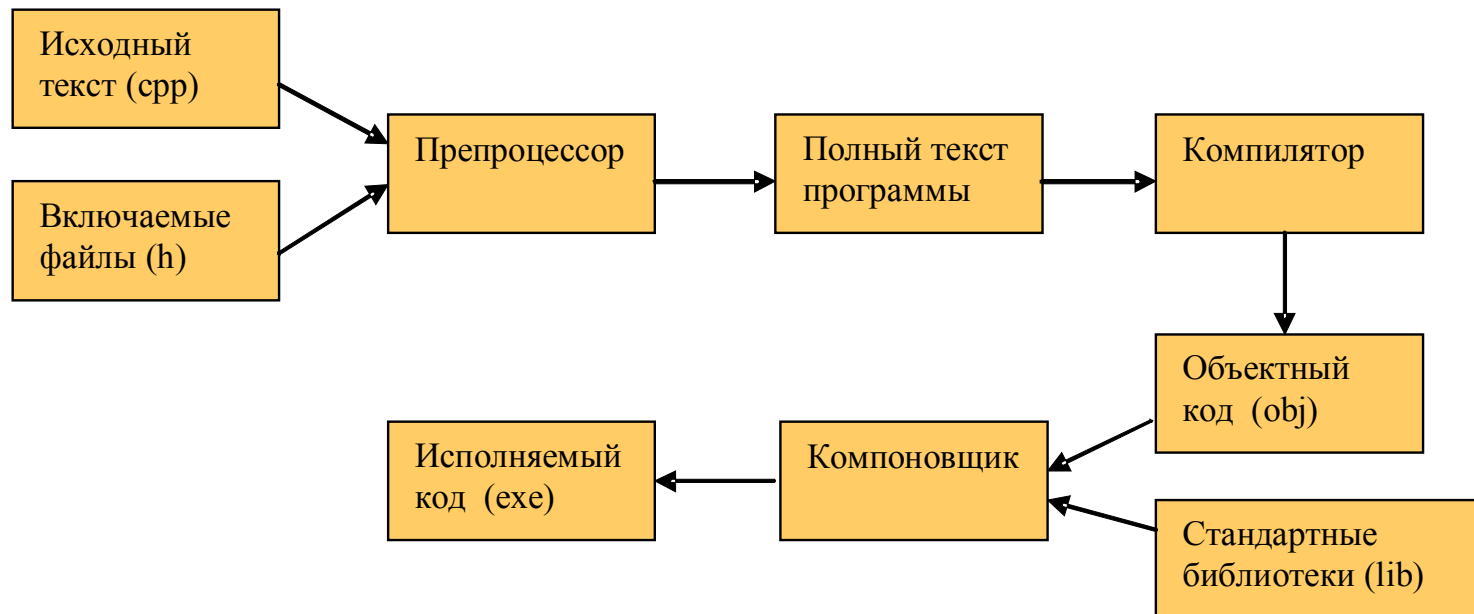
Оператор присваивания

```
cout<<"\nSr.arifm="<<s<<"\n";  
}
```

Вывод результата

Структура программы на языке C/C++

- Директивы препроцессора управляют преобразованием текста программы до ее компиляции. Исходная программа, подготовленная на СИ в виде текстового файла, проходит 3 этапа обработки:
- Препроцессорное преобразование текста.
- Компиляция.
- Компоновка.

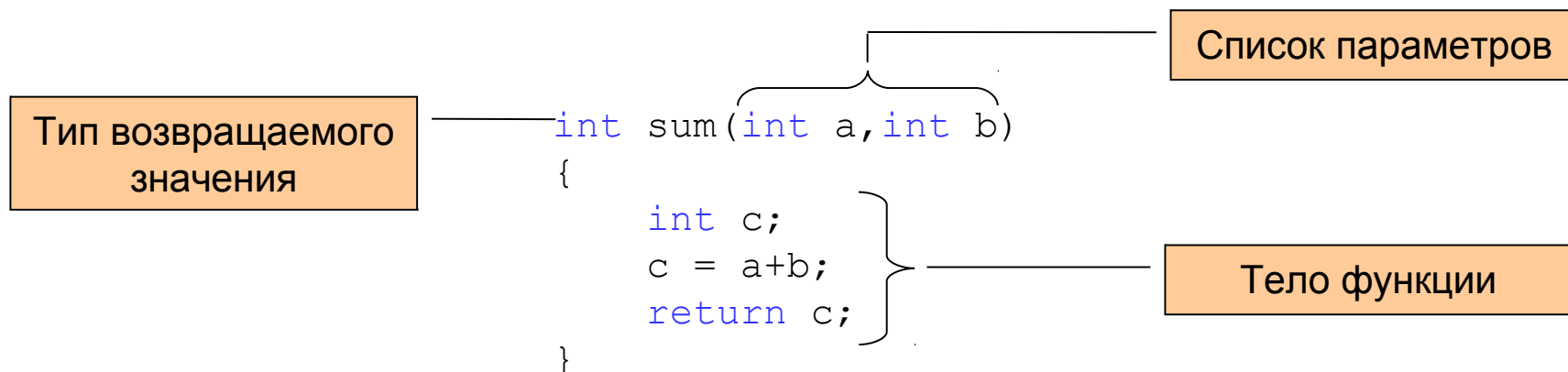


Структура программы на языке C/C++

- Задача препроцессора – преобразование текста программы до ее компиляции.
- Правила препроцессорной обработки определяет программист с помощью директив препроцессора. Директива начинается с символа #.
- `#define` - указывает правила замены в тексте.
`#define ZERO 0.0` – означает, что каждое использование в программе имени `ZERO` будет заменяться на `0.0`.
- `#include <имя заголовочного файла>` – директива для включения в текст программы текста из каталога заголовочных файлов.
- Директива `#pragma` может быть использована в условных определениях, необходимых для обеспечения дополнительной функциональности препроцессора, или для передачи информации о реализации компилятору.

Структура программы на языке C/C++

- Программа представляет собой набор описаний и определений, и состоит из набора функций. Среди этих функций всегда должна быть функция с именем `main`. Перед именем функции помещаются сведения о типе возвращаемого функцией значения. Каждая функция должна иметь список параметров.
- За заголовком функции размещается тело функции. Тело функции – это последовательность определений, описаний и исполняемых операторов, заключенных в фигурные скобки. Каждое определение, описание или оператор заканчивается точкой с запятой.
- Определения – вводят объекты, необходимые для представления в программе обрабатываемых данных.
- Операторы – определяют действия программы на каждом шаге ее исполнения.



Контрольные вопросы

- 1) Для чего нужны директивы препроцессора? Привести примеры
- 2) Из каких частей состоит программа на C++?
- 3) Что такое тело функции?
- 4) Перечислить этапы создания программы на языке C++.



Элементы языка C/C++

- 1. Алфавит языка который включает
 - Прописные и строчные латинские буквы и знак подчеркивания;
 - Арабские цифры от 0 до 9;
 - Специальные знаки:
{ } , | [] () + - / % * . \ ' : ; & ? < > = ! # ^
 - Пробельные символы.
- 2. Из символов формируются лексемы языка:
 - **Идентификаторы** – имена объектов C/C++-программ. В идентификаторе могут быть использованы латинские буквы, цифры и знак подчеркивания. Прописные и строчные буквы различаются. Первым символом должна быть буква или знак подчеркивания (но не цифра). Пробелы в идентификаторах не допускаются.

Prog1	}	Три разных идентификатора
PROG1		
prog1		

- **Ключевые (зарезервированные) слова** – это слова, которые имеют специальное значение для компилятора. Их нельзя использовать в качестве идентификаторов.

`true, false, int, float, switch ...` и.т.д.

- **Знаки операций** – это один или несколько символов, определяющих действие над операндами. Операции делятся на унарные, бинарные и тернарную по количеству участвующих в этой операции операндов.

`+ - * / % < > >= <= == != << >> ! & | && || * ++ -- ...` и.т.д.

- **Константы** – неизменяемые величины.

- **Разделители** – скобки, точка, запятая пробельные символы.



Константы в C/C++

- Константа – это лексема, представляющая изображение фиксированного числового, строкового или символьного значения.
- Константы делятся на 5 групп:
 - целые;
 - вещественные (с плавающей точкой);
 - перечислимые;
 - символьные;
 - строковые.
- Целые константы могут быть десятичными, восьмеричными (начинаются с 0) и шестнадцатеричными (начинаются с 0x).
`10, 0xFF, 016`
- Вещественные константы могут иметь две формы представления:
 - с фиксированной точкой ([цифры].[цифры]).
 - с плавающей точкой ([цифры][.][цифры]E|e[+|-][цифры]).`2.5, 0.5E10`

- Строковая константа – это последовательность символов, заключенная в кавычки. Внутри строк также могут использоваться управляющие символы.

`"\nНовая строка",`

`"\n\"Алгоритмические языки программирования\""`.

- Перечислимые константы вводятся с помощью ключевого слова `enum`. Это обычные целые константы, которым приписаны уникальные и удобные для использования обозначения. Пример:

```
enum{ten=10,three=3,four,five,six};
```

```
enum{Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday};
```

- Символьные константы – это один или два символа, заключенные в апострофы. Последовательности, начинающиеся со знака `\`, называются управляющими, они используются:

- для представления символов, не имеющих графического отображения:

`\a` – звуковой сигнал,

`\b` – возврат на один шаг,

`\n` – перевод строки,

`\t` – горизонтальная табуляция;

- для представления символов `\`, `,`, `'`, `?`, `"` (`\\`, `\'`, `\?`, `\"`);

- для представления символов с помощью шестнадцатеричных или восьмеричных кодов (`\073`, `\0xF5`);

Контрольные вопросы

- 1) Что такое константа? Как константа обрабатывается компилятором?
- 2) Какие типы констант существуют в C++? Привести примеры констант разных типов.
- 3) К какому типу относятся константы 192345, 0x56, 0xCB, 016, 0.7865, .0045, 'c', "x", one, "one", 5, 5.?



Типы данных в C/C++

Типы данных в C/C++

- Типы C/C++ можно разделить на простые и составные. К простым типам относят типы, которые характеризуются одним значением.

int	(целый)	
char	(символьный)	
wchar_t	(расширенный символьный) (C++)	
bool	(логический) (C++)	
float	(вещественный)	} типы с плавающей точкой
double	(вещественный с двойной точностью)	

- Существует 4 спецификатора типа, уточняющих внутреннее представление и диапазон стандартных типов:

short	(короткий)
long	(длинный)
signed	(знаковый)
unsigned	(беззнаковый)

- **Тип `int`**
- Значениями этого типа являются целые числа.
- В 16-битных операционных системах под него отводится 2 байта, в 32-битных – 4 байта.
- Если перед `int` стоит спецификатор `short`, то под число отводится 2 байта, а если спецификатор `long`, то 4 байта. От количества отводимой под объект памяти зависит множество допустимых значений, которые может принимать объект:
 - `short int` – занимает 2 байта, диапазон $-32768 \dots +32767$;
 - `long int` – занимает 4 байта, диапазон $-2\,147\,483\,648 \dots +2\,147\,483\,647$.
- Тип `int` совпадает с типом `short int` на 16-разрядных ПК и с типом `long int` на 32-разрядных ПК.
- Модификаторы `signed` и `unsigned` также влияют на множество допустимых значений, которые может принимать объект:
 - `unsigned short int` – занимает 2 байта, диапазон $0 \dots 65536$;
 - `unsigned long int` – занимает 4 байта, диапазон $0 \dots +4\,294\,967\,295$.

- **Тип char**
- Значениями этого типа являются элементы конечного упорядоченного множества символов. Каждому символу ставится в соответствие число, которое называется кодом символа.
- Под величину символьного типа отводится 1 байт. Тип `char` может использоваться со спецификаторами `signed` и `unsigned`.
`signed char` – диапазон от –128 до 127.
`unsigned char` – диапазон от 0 до 255.
- Для кодировки используется код ASCII (American Standard Code for international interchange). Код ASCII Стандарт кодирования символов латинского алфавита, цифр и вспомогательных символов или действий в виде однобайтового двоичного кода (1 байт = 8 бит). Первоначально стандарт определял только 128 символов, используя 7 битов (от 0 до 127). Использование всех восьми битов позволяет кодировать еще 128 символов. Дополнительные символы могут быть любыми, им отводятся коды от 128 до 255. национальные алфавиты кодируются именно в этой части ASCII-кода.
- Символы с кодами от 0 до 31 относятся к служебным и имеют самостоятельное значение только в операторах ввода-вывода.
- Величины типа `char` также применяются для хранения чисел из указанных диапазонов.

- **Тип `bool`**
- Тип `bool` называется логическим. Его величины могут принимать значения `true` (истина) и `false` (ложь).
- Внутренняя форма представления `false` – 0 (ноль), любое другое значение интерпретируется как `true`.

- **Типы с плавающей точкой (float и double)**
- Внутреннее представление вещественного числа состоит из 2 частей: мантиссы и порядка.
- Мантисса – это численное значение со знаком, порядок – это целое со знаком, определяющее значимость мантиссы.
- Длина мантиссы определяет точность числа, а длина порядка его диапазон.
`1.000000e+001 //представление числа 10`
- В IBM-совместимых ПК величины типа `float` занимают 4 байта, из которых один разряд отводится под знак мантиссы, 8 разрядов под порядок и 24 – под мантиссу.
- Величины типа `double` занимают 8 байтов, под порядок и мантиссу отводятся 11 и 52 разряда соответственно.

- Тип `void`
- К основным типам также относится тип `void`. Множество значений этого типа – пусто.
- Невозможно создать переменную этого типа, но можно использовать указатель.

```
void a;      // нельзя создать переменную...  
void *ptr;   // но можно объявлять указатель.
```

Контрольные вопросы

- 1) Что такое тип данных?
- 2) Чем отличаются типы данных `int` и `unsigned int`?
- 3) Перечислить все типы данных, которые существуют в C++. Сколько места в памяти занимают данные каждого типа?
- 4) Что такое код ASCII?
- 5) Чем отличаются типы данных: `float` и `double`, `int` и `short int`?



Переменные

- Переменная в C++ – именованная область памяти, в которой хранятся данные определенного типа. У переменной есть имя и значение. Имя служит для обращения к области памяти, в которой хранится значение.
- Перед использованием любая переменная должна быть описана.
`int a;`
`float x;`
- Общий вид оператора описания:
[класс памяти] [const] тип имя [инициализатор];
- Класс памяти может принимать значения: `auto`, `extern`, `static`, `register`.
Класс памяти определяет время жизни и область видимости переменной.
- `const` – показывает, что эту переменную нельзя изменять (именованная константа).
- При описании можно присвоить переменной начальное значение (инициализация):
`int a = 10;`
`float b = 20.5;`

- **Классы памяти:**
- `auto` – автоматическая локальная переменная. Этим переменным память выделяется при входе в блок и освобождается при выходе из него. Вне блока такие переменные не существуют.
- `extern` – глобальная переменная, она находится в другом месте программы (в другом файле или далее по тексту). Используется для создания переменных, которые доступны во всех файлах программы.
- `static` – статическая переменная, существующая только в пределах той функции, где определена переменная и сохраняет своё значение и при последующих вызовах этой функции.
- `register` – аналогичны `auto`, но память под эти переменные выделяется непосредственно в регистрах процессора, которая является самой быстродействующей в системе. Если такой возможности нет, то переменные обрабатываются как `auto`.

Переменные

```
int a;                //глобальная переменная

void main()
{
    int b;            //локальная переменная
    extern int x;      //переменная x определена в другом месте
    static int c;      //локальная статическая переменная
    a=1;              //присваивание глобальной переменной
    int a;            //локальная переменная a
    a=2;              //присваивание локальной переменной
    ::a=3;            //присваивание глобальной переменной
}
int x=4;              //определение и инициализация x
```

- Областью действия переменной `a` является вся программа, кроме тех строк, где используется локальная переменная `a`. Переменные `b` и `c` – локальные, область их видимости – блок. Время жизни различно: память под `b` выделяется при входе в блок (т. к. по умолчанию класс памяти `auto`), освобождается при выходе из него. Переменная `c` (`static`) существует в пределах функции, внутри которой она определена и сохраняет своё значение и при последующих вызовах этой функции.
- Имя переменной должно быть уникальным в своей области действия.

- Если локальные объекты описаны со служебным словом `static`, то они также существуют до конца программы(существует в пределах видимости данной функции). Инициализация их происходит, когда в первый раз управление "проходит через" описание этих объектов, например:

```
int a = 1;
void f()
{
    int b = 1;    // инициализируется при каждом вызове f()
    static int c = a; // инициализируется только один раз
    cout << " a = " << a++
          << " b = " << b++
          << " c = " << c++ << '\n';
}

int main()
{
    while (a < 4) f();
}
```

Здесь программа выдаст такой результат:

```
a = 1 b = 1 c = 1
a = 2 b = 1 c = 2
a = 3 b = 1 c = 3
```

Контрольные вопросы

- 1) Что такое переменная? Чем объявление переменной отличается от ее определения? Привести примеры определений и объявлений.
- 2) Что такое класс памяти? Какие классы памяти существуют в C++? Привести примеры объявлений и определений переменных разных классов памяти.



Выражения

Выражения

- Из констант, переменных, разделителей и знаков операций можно конструировать выражения. Каждое выражение представляет собой правило вычисления нового значения.
- Если выражение формирует целое или вещественное число, то оно называется арифметическим. Пара арифметических выражений, объединенная операцией сравнения, называется отношением.
- Если отношение имеет ненулевое значение, то оно – истинно, иначе – ложно.

$a+b+64$ // арифметическое выражение
 $c-4 > d*e$ // отношение



Ввод и вывод данных

Ввод и вывод данных

- В языке C/C++ нет встроенных средств ввода и вывода – он осуществляется с помощью функций, типов и объектов, которые находятся в стандартных библиотеках. Существует два основных способа: функции C и объекты C++.
- Для ввода/вывода данных в стиле C используются функции, которые описываются в библиотечном файле `stdio.h`.
- Функция `printf` (форматная строка, список аргументов);
- Форматная строка – строка символов, заключенных в кавычки, которая показывает, как должны быть напечатаны аргументы. Может содержать:
 - Символы печатаемые текстуально.
 - Спецификации преобразования.
 - Управляющие символы.
- Каждому аргументу соответствует своя спецификация преобразования:

<code>%d, %i</code>	– десятичное целое число;
<code>%f</code>	– число с плавающей точкой;
<code>%e, %E</code>	– число с плавающей точкой в экспоненциальной форме;
<code>%u</code>	– десятичное число в беззнаковой форме;
<code>%c</code>	– символ;
<code>%s</code>	– строка.

- В форматную строку также могут входить управляющие символы:
`\n` – управляющий символ новая строка;
`\t` – табуляция;
`\a` – звуковой сигнал и др.
- Также в форматной строке могут использоваться модификаторы формата, которые управляют шириной поля. Модификаторы – это числа, которые указывают минимальное количество позиций для вывода значения и количество позиций для вывода дробной части числа:

`%[-]m[.p]C`, где

– – задает выравнивание по левому краю,

m – минимальная ширина поля,

p – количество цифр после запятой для чисел с плавающей точкой и минимальное количество выводимых цифр для целых чисел.

C – спецификация формата вывода.

- Пример:

```
printf("\nСпецификации формата:\n%10.5d – целое,\n%10.5f – с плавающей точкой\n%10.5e – в экспоненциальной форме\n%10s – строка", 10, 10.0, 10.0, "10");
```

Ввод и вывод данных

- `scanf` (форматная строка, список аргументов);
- В качестве аргументов используются адреса переменных. Например:
`scanf(" %d%f ", &x, &y);`
- При использовании библиотеки классов C++, используется библиотечный файл `iostream.h`, в котором определены стандартные потоки: ввода данных с клавиатуры `cin` и вывода данных на экран `cout`, а также соответствующие операции:
- `<<` – Операция записи данных в поток.
- `>>` – Операция чтения данных из потока.
- Пример:

```
#include <iostream.h>
void main()
{
    cout << "\nВведите количество элементов:";
    cin >> n;
}
```


Контрольные вопросы

- 1) Что такое выражение? Из чего состоит выражение?
- 2) В каком библиотечном файле определены стандартные потоки ввода данных с клавиатуры и вывода данных на экран в C++?
- 3) Что такое управляющие символы? Для чего они нужны? Приведите примеры.



Операторы языка С

Операторы языка С


- Операторы управления работой программы называют управляющими конструкциями программы. К ним относят:
 - составные операторы;
 - операторы выбора;
 - операторы циклов;
 - операторы перехода.

- **Оператор "выражение"**
- Любое выражение, заканчивающееся точкой с запятой, рассматривается как оператор, выполнение которого заключается в вычислении этого выражения.
`i++;`
`a += 2;`
`x = a+b;`

Операторы языка C


- **Составные операторы**
- К составным операторам относят собственно составные операторы и блоки. В обоих случаях это последовательность операторов, заключенная в фигурные скобки.
- Блок отличается от составного оператора наличием определений в теле блока.

```
{  
    n++;  
    summa += n;  
}
```



составной оператор

```
{  
    int n = 0;  
    n++;  
    summa += n;  
}
```



блок

Операторы языка C

- **Операторы выбора**

- Операторы выбора – условный оператор и переключатель.

- **Условный оператор** имеет полную и сокращенную форму.

```
if (выражение-условие) оператор1;           // сокращенная форма
```

```
if (выражение-условие) оператор1;           // полная форма
```

```
else оператор2;
```

- Если значение выражения-условия **отлично** от нуля, то выполняется оператор1, **иначе** выполняется оператор2.

- **Пример:**

```
if (d>=0)
{
    x1=(-b-sqrt(d))/(2*a);
    x2=(-b+sqrt(d))/(2*a);
    cout<< "\nx1="<<x1<<"x2="<<x2;
}
else cout<<"\nРешения нет";
```

- **Переключатель** определяет множественный выбор.

```
switch (выражение)
{
    case константа1: оператор1;
    case константа2: оператор2;
    . . . . .
    [default: операторы;]
}
```

- При выполнении оператора `switch`, вычисляется выражение, записанное после `switch`, оно должно быть целочисленным. Полученное значение последовательно сравнивается с константами, которые записаны следом за `case`. При первом же совпадении выполняются операторы, помеченные данной меткой.
- Если выполненные операторы не содержат оператора перехода, то далее выполняются операторы всех следующих вариантов, пока не появится оператор перехода или не закончится переключатель.
- Если значение выражения, записанного после `switch`, не совпало ни с одной константой, то выполняются операторы, которые следуют за меткой `default`.

- Пример использования оператора switch:

```
#include <iostream.h>
void main()
{
    int i;
    cout<<"\nEnter the number";
    cin>>i;
    switch(i)
    {
        case 1: cout<<"\nthe number is one";
        case 2: cout<<"\n2*2="<<i*i;
        case 3: cout<<"\n3*3="<<i*i; break;
        case 4: cout<<"\n"<<i<<" is very beautiful!";
        default: cout<<"\nThe end of work";
    }
}
```

Операторы циклов

Цикл с предусловием:

```
while (выражение-условие)  
оператор;
```

Если выражение-условие истинно, то тело цикла выполняется до тех пор, пока выражение-условие не станет ложным.

```
while (a!=0)  
{  
    cin>>a;  
    s+=a;  
}
```

Цикл с постусловием:

```
do  
оператор;  
while (выражение-условие);
```

Тело цикла выполняется до тех пор, пока выражение-условие истинно.

```
do  
{  
    cin>>a;  
    s+=a;  
}  
while (a!=0);
```


■ Цикл с параметром:

```
for (выражение_1; выражение-условие; выражение_3)  
оператор;
```

■ Выражение_1 – задает начальные условия для цикла (инициализация).

■ Выражение-условие определяет условие выполнения цикла, если оно не равно 0, цикл выполняется, а затем вычисляется значение выражения_3.

■ Выражение_3 – задает изменение параметра цикла или других переменных (коррекция).

■ Выражение_1 и выражение_3 могут состоять из нескольких выражений, разделенных запятыми.

■ Любое выражение может отсутствовать, но разделяющие их " ; " должны быть обязательно.

■ Примеры:

■ Уменьшение параметра:

```
for ( n=10; n>0; n--)  
{  
    оператор;  
}
```

Операторы языка C

- Проверка условия отличного от того, которое налагается на число итераций:

```
for (num=1; num*num*num<216; num++)  
{  
    оператор;  
}
```

- Коррекция с помощью умножения:

```
for ( d=100.0; d<150.0; d*=1.1)  
{  
    оператор;  
}
```

- Коррекция с помощью арифметического выражения:

```
for (x=1; y<=75; y=5*(x++)+10)  
{  
    оператор;  
}
```

Операторы перехода

Операторы перехода выполняют безусловную передачу управления.

break – оператор прерывания цикла.

```
{
    оператор;
    if (<выражение_условие>) break;
    оператор;
}
```

Оператор `break` целесообразно использовать, когда условие продолжения итераций надо проверять в середине цикла.

Пример: найти сумму чисел, числа вводятся с клавиатуры до тех пор, пока не будет введено 100 чисел или 0.

```
for (s=0, i=1; i<100; i++)
{
    cin>>x;
    if (!x) break;    // если ввели 0, то суммирование
                      // заканчивается.

    s+=x;
}
```

Операторы языка C

- **continue** – переход к следующей итерации цикла. Используется, когда тело цикла содержит ветвления.
- Пример: найти количество и сумму положительных чисел.

```
for (k=0, s=0, x=1; x!=0; )  
{  
    cin>>x;  
    if (x<=0) continue;  
    k++; s+=x;  
}
```
- **goto <метка>** – передает управление оператору, который содержит метку.
- В теле той же функции должна присутствовать конструкция:
`<метка>: оператор;`
- Применение **goto** нарушает принципы структурного и модульного программирования.
- Нельзя передавать управление внутрь операторов `if`, `switch` и циклов. Нельзя переходить внутрь блоков, содержащих инициализацию, на операторы, которые стоят после инициализации.

Операторы языка С

- `return` – оператор возврата из функции. Он всегда завершает выполнение функции и передает управление в точку ее вызова. Вид оператора:
`return [выражение];`

Контрольные вопросы

- 1) Что такое операторы и какие они бывают?
- 2) Чем отличается цикл с предусловием от цикла с постусловием?
- 3) Для чего нужны операторы перехода и как они работают?
- 4) Для чего нужен переключатель? Как он работает? Приведите примеры.



Массивы

- В языке C, кроме базовых типов, разрешено вводить и использовать производные типы, полученные на основе базовых.
- Стандарт языка определяет три способа получения производных типов:
 - массив элементов заданного типа;
 - указатель на объект заданного типа;
 - функция, возвращающая значение заданного типа.

- **Массив** – это упорядоченная последовательность переменных одного типа. Каждому элементу массива отводится одна ячейка памяти. Элементы одного массива занимают последовательно расположенные ячейки памяти.
- Все элементы имеют одно имя – имя массива и отличаются индексами – порядковыми номерами в массиве.
- Количество элементов в массиве называется его размером. Чтобы отвести в памяти нужное количество ячеек для размещения массива, надо заранее знать его размер. Резервирование памяти для массива выполняется на этапе компиляции программы.

Массивы

■ Определение массива в С

■ Массивы определяются следующим образом:

```
тип имя[размер];
```

■ Примеры:

```
int a[100];
```

```
float b[20];
```

```
char c[32];
```

■ Элементы массива всегда нумеруются с 0.

45	352	63		124	значения элементов массива
0	1	2	99	индексы элементов массива

■ Чтобы обратиться к элементу массива, надо указать имя массива и номер элемента в массиве (индекс):

```
a[55] // индекс задается как константа
```

```
a[i] // индекс задается как переменная
```

```
a[2*i] // индекс задается как выражение
```

Массивы

- Элементы массива можно задавать при его определении:

```
int a[10]={1,2,3,4,5,6,7,8,9,10};
```

- Длина массива может вычисляться компилятором по количеству значений, перечисленных при инициализации:

```
int a[]={1,2,3,4,5};           // длина массива - 5 элементов.
```

- Примеры решения задач и использованием массивов.

- **1.Формирование массива с помощью датчика случайных чисел:**

функция rand() возвращает псевдослучайное (от 0 до 32767).

функция srand() задает новую точку отсчета для генерации случайных чисел.

```
#include <iostream.h>
#include <time.h>
#include <stdlib.h>          // файл с описанием функции rand()
void main()
{
    int a[100];
    int n;
    srand( (unsigned)time( NULL ) );
    cout<<"\nEnter the size of array:"; cin>>n;
    for(int i=0; i<n; i++)
    {
        a[i]=rand()%100;
        cout<<a[i]<< " ";
    }
}
```

Поиск максимального элемента массива.

```
#include <iostream.h>
#include <stdlib.h>
void main()
{
    int a[100];
    int n;
    cout<<"\nEnter the size of array:";
    cin>>n;
    for(int i=0;i<n;i++)    // заполнение массива
    {
        a[i]=rand()%100-50;
        cout<<a[i]<<" ";
    }
    // задаем начальное значение переменной max
    int max = a[0];
    for(i=1;i<n;i++)
    // сравниваем элементы массива с переменной max:
    if(a[i]>max) max=a[i];
    cout<<"\nMax="<<max;
}
```

Поиск суммы элементов массива с четными индексами.

```
#include <iostream.h>
#include <stdlib.h>
void main()
{
    int a[100];
    int n;
    cout<<"\nEnter the size of array:";
    cin>>n;
    for(int i=0;i<n;i++)
    {
        //заполнение массива случайными числами
        a[i]=rand()%100-50;
        cout<<a[i]<<" ";
    }
    int summ = 0;
    for(i=0; i<n; i+=2)
        // суммируются элементы с индексами 0, 2, 4, и т. д.
        summ +=a[i];
    cout<<"\nsumm="<<Sum;
}
```

Контрольные вопросы

- 1) Что такое массив? Способы определения массива.
- 2) Что такое имя массива?
- 3) Чем характеризуется перебор элементов массива?
- 4) Какие способы перебора массива по направлению вы знаете?
- 5) Что такое датчик случайных чисел?
- 6) Что такое сортировка? Какие методы сортировки бывают?
- 7) Какие способы поиска в одномерных массивах вы знаете и как они работают?

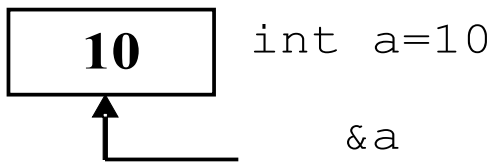


Указатели

■ Понятие указателя.

■ Когда компилятор обрабатывает оператор определения переменной, например,
`int i=10;`

то в памяти выделяется участок памяти в соответствии с типом переменной и записывает в этот участок указанное значение. Все обращения к этой переменной компилятор заменит адресом области памяти, в которой хранится эта переменная.



■ Программист может определить собственные переменные для хранения адресов областей памяти. Такие переменные называются указателями. Указатель не является самостоятельным типом, он всегда связан с каким-то другим типом.

■ Указатели делятся на две категории: указатели на объекты и указатели на функции.

Указатели

- Объявление указателей на объекты, которые хранят адрес области памяти, имеет вид:
тип* имя;
- Знак * обозначает указатель и относится к типу переменной, поэтому его рекомендуется ставить рядом с типом, а от имени переменной отделять пробелом, за исключением тех случаев, когда описываются несколько указателей. При описании нескольких указателей знак * ставится перед именем переменной-указателя, т. к. иначе будет не понятно, что эта переменная также является указателем.

```
int* i;           // i хранит адрес ячейки с целым числом
int* x, y;        // x хранит адрес, а y – целое число!
double *f, *ff;   // два указателя на ячейки с double
char* c;          // указатель на ячейку, хранящую символ
```

- Тип может быть любым, кроме ссылки.
- Размер указателя зависит от модели памяти.
- Можно определить указатель на указатель:

```
int** a;
```

... который отличается от простого указателя тем, что хранит адрес ячейки, которая сама хранит адрес некоего объекта в памяти.

- Указатель можно сразу проинициализировать:

```
int* pi=&i;           // указатель на целую переменную.
```

```
const int* pci=&ci;    // указатель на целую константу.
```

```
int* const cpi=&i;     // указатель-константа на переменную  
                      // целого типа.
```

```
const int* const cpc=&ci; // указатель-константа на целую  
                          // константу.
```

- Если модификатор `const` относится к указателю (т. е. находится между именем указателя и `*`), то он запрещает изменение указателя, а если он находится слева от типа (т. е. слева от `*`), то он запрещает изменение значения, на которое указывает указатель.

- Для инициализации указателя существуют следующие способы:

- С помощью операции получения адреса

```
int a=5;  
int* p=&a; //или int p(&a);
```

- С помощью проинициализированного указателя

```
int* r=p;
```

- Адрес присваивается в явном виде

```
char* cp=(char*) 0x B800 0000;  
0x B800 0000 – шестнадцатеричная константа,  
(char*) – операция приведения типа.
```

- Присваивание пустого значения:

```
int* N=NULL;  
int* R=0;
```

Указатели

- **Динамическая память.**
- Все переменные, объявленные в программе размещаются в одной непрерывной области памяти, которую называют сегментом данных (64 Кб). Такие переменные называются статическими.
- Динамическая память – это память, выделяемая программе для ее работы за вычетом сегмента данных, стека, в котором размещаются локальные переменные подпрограмм и собственно тела программы.
- Для работы с динамической памятью используют указатели. С их помощью осуществляется доступ к участкам динамической памяти, которые называются динамическими переменными.
- Динамические переменные создаются с помощью специальных функций и операций. Они существуют либо до конца работы программ, либо до тех пор, пока не будут уничтожены с помощью специальных функций или операций.
- Для создания динамических переменных используют операцию `new`:
`указатель = new имя_типа [(инициализатор)] ;`
- Для удаления динамических переменных используется операция `delete`:
`delete указатель;`
- Пример:
`int* x = new int(5);`
`delete x;`

- **Операции с указателями.**

- С указателями можно выполнять следующие операции:

- разыменование;
- присваивание;
- арифметические операции;
- сравнение;
- приведение типов.

- Операция разыменования предназначена для получения значения переменной или константы, адрес которой хранится в указателе.

```
int a;           // переменная типа int.  
int* pa=new int; // выделение памяти под динамическую  
                // переменную.  
*pa=10;         // присвоили значение динамической переменной.  
a=*pa;          // присвоили значение переменной a
```

- Присваивать значение указателям-константам запрещено.

■ Приведение типов

■ На одну и ту же область памяти могут ссылаться указатели разного типа. Если применить к ним операцию разыменования, то получатся разные результаты.

```
int a=123;
int* pi=&a;
char* pc=(char*)&a;
float* pf=(float*)&a;
printf("\n%x\t%i",pi,*pi);
printf("\n%x\t%c",pc,*pc);
printf("\n%x\t%f",pf,*pf);
```

■ При выполнении этой программы получатся следующие результаты:

```
66fd9c 123
66fd9c {
66fd9c 0.000000
```

■ При использовании в выражении указателей разных типов, явное преобразование требуется для всех типов, кроме `void*`.

■ Указатель может неявно преобразовываться в значения типа `bool`, при этом ненулевой указатель преобразуется в `true`, а нулевой в `false`.

- Арифметические операции применимы только к указателям одного типа.
- Инкремент увеличивает значение указателя на величину `sizeof(тип)`.

```
char* pc;  
int* pi;  
double* pd;  
. . . . .  
pc++;           //значение увеличится на 1  
pi++;           //значение увеличится на 4  
pd++;           //значение увеличится на 8
```
- Декремент уменьшает значение указателя на величину `sizeof(тип)`.
- Разность двух указателей – это разность их значений, деленная на размер типа в байтах.
- Суммирование двух указателей не допускается.
- Можно суммировать указатель и константу.

Контрольные вопросы

- 1) Что такое адрес?
- 2) Что такое указатель и какие они бывают?
- 3) Какие способы инициализации указателей существуют?
- 4) Какие операции с указателями вы знаете?
- 5) Что такое динамическая память и для чего она используется?
- 6) Какие операции используются для работы с динамическими переменными?



Ссылки

Ссылки

- Ссылка - это синоним имени объекта, указанного при инициализации ссылки.

```
тип& имя = имя_объекта;
```

```
int x; //определение переменной
```

```
int& sx=x; //определение ссылки на переменную x
```

```
const char& CR='\n'; //определение ссылки на константу
```

- *Правила работы с ссылками:*
 - Переменная-ссылка должна явно инициализироваться при ее описании, если она не является параметром функции, не описана как `extern` или не ссылается на поле класса.
 - После инициализации ссылке не может быть присвоено другое значение.
 - Не существует указателей на ссылки, массивов ссылок и ссылок на ссылки.
 - Операция над ссылкой приводит к изменению величины, на которую она ссылается.
- Ссылка, в отличие от указателя, не занимает дополнительного пространства в памяти и является просто другим именем объекта.

■ Пример использования ссылки:

```
#include <iostream.h>
void main()
{
    int i = 123;
    int& si = i;
    cout<<"\ni="<<i<<" si="<<si;
    i=456;
    cout<<"\ni="<<i<<" si="<<si;
    i=0;
    cout<<"\ni="<<i<<" si="<<si;
}
```

Выведется:

```
i=123 si=123
i=456 si=456
i=0 si=0
```



Указатели и массивы

Одномерные массивы и указатели

- При определении массива ему выделяется память. После этого имя массива воспринимается как константный указатель того типа, к которому относятся элементы массива.

```
int a[100];
```

```
/* определение количества занимаемой массивом памяти, в нашем  
случае это 4*100=400 байт */
```

```
int k = sizeof(a);
```

```
/* вычисление количества элементов массива */
```

```
int n = sizeof(a) / sizeof(a[0]);
```

- Результатом операции & является адрес нулевого элемента массива:

```
имя_массива==&имя_массива==&имя_массива[0]
```

- Имя массива является указателем-константой, значением которой служит адрес первого элемента массива. Следовательно, к нему применимы все правила адресной арифметики, связанной с указателями.
- Используя указатели, обращение по индексу можно записать следующим образом: `* (имя_массива+индекс)` .

Указатели и массивы

- Так как имя массива является константным указателем, то его невозможно изменить, следовательно, запись `* (a++)` будет ошибочной, `a * (a+1)` – нет.
- Указатели можно использовать и при определении массивов:

```
int a[100] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

```
// поставили указатель на уже определенный массив :
```

```
int* na = a;
```

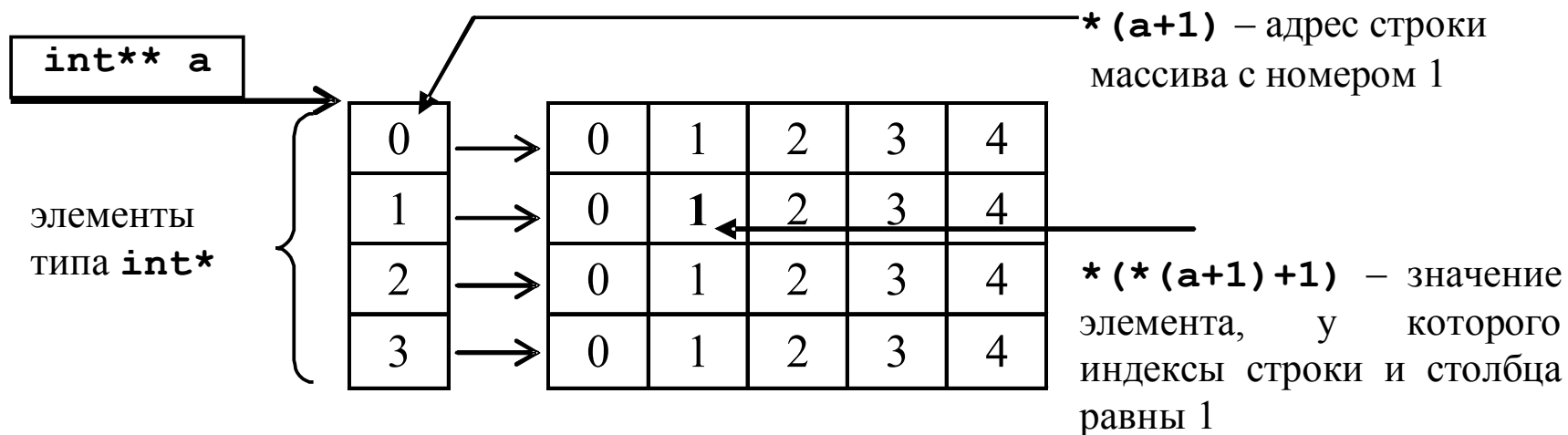
```
// выделили в динамической памяти место под массив из 100
```

```
// элементов :
```

```
int b = new int[100];
```

■ Многомерные массивы и указатели

- Многомерный массив – это массив, элементами которого служат массивы. Например, массив `int a[4][5]` – это массив из указателей `int*`, которые содержат имена одноименных массивов из 5 целых элементов:



Выделение памяти под массив, элементами которого являются массивы.

Указатели и массивы

- Инициализация многомерных массивов выполняется аналогично одномерным массивам.

```
// проинициализированы все элементы массива
```

```
int a[3][4] = {{11, 22, 33, 44}, {55, 66, 77, 88}, {99, 110, 120, 130}};
```

```
// проинициализированы первые элементы каждой строки
```

```
int b[3][4] = {{1}, {2}, {3}};
```

```
// проинициализированы все элементы массива
```

```
int c[3][2] = {1, 2, 3, 4, 5, 6};
```

- Доступ к элементам многомерных массивов возможен и с помощью индексированных переменных и с помощью указателей:
 - `a[1][1]` – доступ с помощью индексированных переменных,
 - `*(*(a+1)+1)` – доступ к этому же элементу с помощью указателей.

- **Динамические массивы**

- Операция `new` при использовании с массивами имеет следующий формат:

```
new тип_массива
```

Такая операция выделяет для массива участок динамической памяти соответствующего размера, но не позволяет инициализировать элементы массива.

- Операция `new` возвращает указатель, значением которого служит адрес первого элемента массива.
- При выделении динамической памяти размеры массива должны быть полностью определены.

```
//выделение динамической памяти 100*sizeof(int) байт
```

```
int* a = new int[100];
```

```
long (*la)[4] //указатель на массив из 4 элементов типа long
```

```
la=new[2][4] //выделение динамической памяти 2*4*sizeof(long) байт
```

- Освобождение памяти, выделенной под массив (а адресует начало):

```
delete[] a;
```

```
delete[] la;
```

- Пример удаления из матрицы строки с номером K:

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
void main()
{
    int n,m; // размерность матрицы
    int i,j;
    cout<<"\nEnter n";    cin>>n; // ввод количества строк
    cout<<"\nEnter m";    cin>>m; // ввод количества столбцов
    // выделение памяти под массив указателей на строки
    int** matr = new int*[n];
    // выделение памяти под элементы матрицы
    for(i=0; i<n; i++)
        matr[i] = new int [m];
    // заполнение матрицы
    for(i=0; i<n; i++)
        for(j=0; j<m; j++)
            matr[i][j] = rand()%10;
    // печать сформированной матрицы
    /*...*/
}
```

Указатели и массивы

```
// удаление строки с номером k
int k;
cout<<"\nEnter k"; cin>>k;
// формирование новой матрицы
int** temp = new int*[n-1];
for(i=0; i<n; i++)
temp[i]=new int[m];
// заполнение новой матрицы
int t;
for(i=0,t=0; i<n; i++)
/*если номер строки отличен от k, переписываем строку в новую
матрицу: */
if(i!=k)
{
for(j=0; j<m; j++)
temp[t][j] = matr[i][j];
t++;
}
// удаление старой матрицы
for(i=0; i<n; i++)
delete matr[i];
delete[] matr;
n--; // уменьшаем количество строк
}
```

Контрольные вопросы

- 1) Что такое ссылка?
- 2) Чем является имя массива?
- 3) Что является результатом операции &?
- 4) Что такое многомерный массив?
- 5) Как возможен доступ к элементам массива?
- 6) Что возвращает операция new?



Символьная информация и строки

Символьная информация и строки

- Для символьных данных в С введен тип `char`. Для представления символьной информации используются *символы*, *символьные переменные* и *текстовые константы*.

```
const char c='c';    // символ
char a,b;            // символьные переменные
const char *s="Пример строки\n"; // текстовая константа
```

- Строка в языке С – это массив символов, заканчивающийся нуль-символом – `'\0'` (нуль-терминатором). По положению нуль-терминатора определяется фактическая длина строки. Количество элементов в таком массиве на 1 больше, чем изображение строки.

Строка "A" = 'A' + '\0'

- Присвоить значение строке с помощью оператора присваивания нельзя. Поместить строку в массив можно либо при вводе, либо с помощью инициализации.

Символьная информация и строки

- Примеры создания строк, выделения памяти и т.д.:

```
void main()
{
    char s1[10] = "string1";
    int k = sizeof(s1);
    cout<<s1<<"\t"<<k<<endl;
    char s2[] = "string2";
    k = sizeof(s2);
    cout<<s2<<"\t"<<k<<endl;
    char s3[] = {'s','t','r','i','n','g','3'}
    k = sizeof(s3);
    cout<<s3<<"\t"<<k<<endl;
    //указатель на строку, ее нельзя изменить:
    const char *s4 = "string4";
    k = sizeof(s4);
    cout<<s4<<"\t"<<k<<endl;
}
```

- Результаты:

```
string1 10 - выделено 10 байтов, в том числе под \0
string2 8  - выделено 8 байтов (7+1 байт под \0)
string3 8  - выделено 8 байтов (7+1 байт под \0)
string4 4  - размер указателя
```

Символьная информация и строки

- Для ввода/вывода символьных данных в С определены следующие функции:
 - `int getchar(void)` – осуществляет ввод одного символа из входного потока, при этом она возвращает один байт информации (символ) в виде значения типа `int`. Это сделано для распознавания ситуации, когда при чтении будет достигнут конец файла.
 - `int putchar (int c)` – помещает в стандартный выходной поток символ `c`.
 - `char* gets(char*s)` – считывает строку `s` из стандартного потока до появления символа `'\n'`, сам символ `'\n'` в строку не заносится.
 - `int puts(const char* s)` - записывает строку в стандартный поток, добавляя в конец строки символ `'\n'`, в случае удачного завершения возвращает значение больше или равное 0 и отрицательное значение (`EOF=-1`) в случае ошибки.
- Функция `gets()` осуществляет чтение строки до символа `'\n'`.
- Функция `scanf()` осуществляет чтение строки до первого пробела.

Символьная информация и строки

- Для работы со строками существуют специальные библиотечные функции, которые содержатся в заголовочном файле `string.h`.
 - `unsigned strlen(const char* s)` – Вычисляет длину строки `s`.
 - `int strcmp(const char* s1, const char* s2)` – Сравнивает строки `s1` и `s2`.
 - `int strncmp(const char* s1, const char* s2)` – Сравнивает первые `n` символов строк `s1` и `s2`.
 - `char* strcpy(char* s1, const char* s2)` – Копирует символы строки `s1` в строку `s2`.
 - `char* strcat(char* s1, const char* s2)` – Приписывает строку `s2` к строке `s1`.
 - `char* strdup(const char* s)` – Выделяет память и переносит в нее копию строки `s`.

Символьная информация и строки

■ Пример программы:

*/*Дана строка символов, состоящая из слов. Удалить из строки все слова, начинающиеся с цифры.*/*

```
#include <stdio.h>
#include <string.h>
void main()
{
    char s[250], //исходная строка
    w[25],       //слово
    mas[10][25]; //массив слов
    puts("\nвведите строку"); gets(s);
    int k=0,t=0,i,len,j;
    Len = strlen(s);
    while(t<len)
    {
        for(j=0,i=t; s[i]!=' '; i++,j++)
            w[j]=s[i]; //формируем слово до пробела
        w[j]='\0';      //формируем конец строки
        strcpy(mas[k],w); //копируем слово в массив
        k++;           //увеличиваем счетчик слов
        t=i+1; //переходим к следующему слову в исходной строке
    }
}
```

Символьная информация и строки

```
strcpy(s, ""); //очищаем исходную строку
```

```
for(t=0;t<k;t++)
```

```
if(mas[t][0]<'0' && mas[t][0]>'9') //если символ не цифра
```

```
{
```

```
    strcat(s,mas[t]); //копируем в строку слово
```

```
    strcat(s," ");    //копируем в строку пробел
```

```
}
```

```
puts(s); //выводим результат
```

```
}
```

Символьная информация и строки

- Воспользуемся следующим шаблоном программы для русификации сообщений в редакторе компилятора и на экране решения. Для этого используется специальная функция CharToOem, для которой нужно подключить библиотеку windows.h

```
#include <iostream.h> // для cout (вывода на экран)
#include <windows.h> // для CharToOem
int main ()
{
    char str[20];
    CharToOem("Русский тест", str);
    cout << str;
}
```

- Существует еще один вариант русификации сообщений с помощью функции **SetConsoleOutputCP** (см. на следующем слайде)

Символьная информация и строки

- В библиотеку windows входит одна функция, которая позволяет сменить кодировку консоли. Дело в том, что консоль использует, так называемую кодировку "ОЕМ". В то время как windows работает в кодировке " Windows 3.1 Cyrillic ", или иначе 1251. Коды символов национальных алфавитов в этих кодировках различные. Поэтому на экране вместо русских букв появляются всякие крючки. Для вывода кириллицы, нужно выполнить два условия:
 - сменить кодировку консоли;
 - использовать на консоли приемлемый шрифт.
- В приведенной ниже программе используется функция SetConsoleOutputCP(номер_кодировки), которая позволяет назначить кодировку, в данном случае это будет " Windows 3.1 Cyrillic ".

```
#include <iostream>
#include <windows>
using namespace std;
int main()
{
    SetConsoleOutputCP(1251);
    cout<<"Привет!"<<endl<<endl;
    return 0;
}
```

- Чтобы программа до конца работала нужно выполнить второе условие – применить нужный шрифт. Необходим шрифт TrueType, растровый не подойдет.

Контрольные вопросы

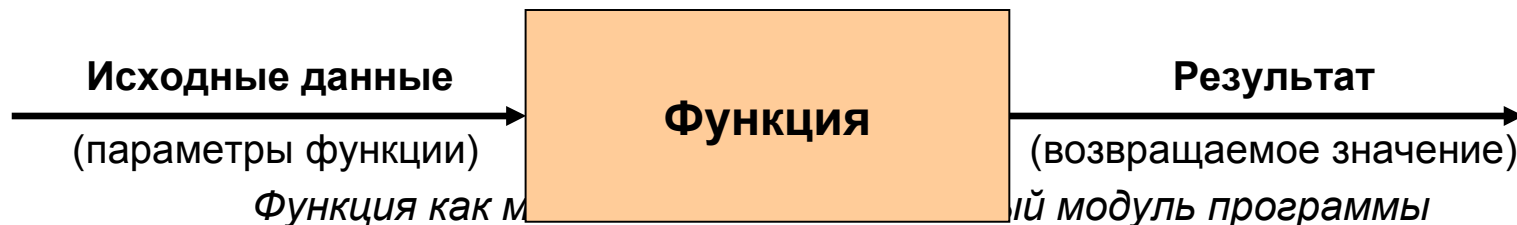
- 1) Какой тип используется для символьных данных в C++?
- 2) Что является символьной информацией?
- 3) Какие функции для ввода/вывода символьных данных существуют в C++?
- 4) В какой библиотеки содержатся функции для работы со строками? Приведите примеры этих функций.
- 5) Какие функции существуют для руссификации сообщений и как с ними работать?



Функции в С

Функции в С

- Чтобы уменьшить сложность программы, ее разбивают на части. В языке С задача может быть разделена на более простые подзадачи с помощью функций.
- **Объявление и определение функций.**
- **Функция** – это именованная последовательность описаний и операторов, выполняющая законченное действие, например, формирование массива, печать массива и т. д.
- Функция, во-первых, является одним из производных типов С++, а, во-вторых, минимальным исполняемым модулем программы.



Функции в С

- Любая функция должна быть объявлена и определена.
- Объявление функции (прототип, заголовок) задает имя функции, тип возвращаемого значения и список передаваемых параметров.
- Определение функции содержит, кроме объявления, тело функции, которое представляет собой последовательность описаний и операторов.

```
тип имя_функции ([ список_формальных_параметров ] )  
{  
    тело_функции  
}
```

- **Тело_функции** – это блок или составной оператор. Внутри функции нельзя определить другую функцию.
- В теле функции должен быть оператор, который возвращает полученное значение функции в точку вызова. Он может иметь две формы:

```
return выражение;  
return;
```
- Первая форма используется для возврата результата.
- Вторая форма используется, если функция не возвращает значения, т. е. имеет тип `void`.

Функции в С

- Тип возвращаемого значения может быть любым, кроме массива и функции, но может быть указателем на массив или функцию.
- Список формальных параметров – это те величины, которые требуется передать в функцию. Элементы списка разделяются запятыми. Для каждого параметра указывается тип и имя. В объявлении имени можно не указывать.
- При вызове функции указываются: имя функции и фактические параметры. Фактические параметры заменяют формальные параметры при выполнении операторов тела функции.
- Фактические и формальные параметры должны совпадать по количеству и типу.
- Объявление функции должно находиться в тексте раньше вызова функции, чтобы компилятор мог осуществить проверку правильности вызова.

- Пример программы с использованием функций:

```
/*Заданы координаты сторон треугольника, найти его площадь. */
#include <iostream.h>
#include <math.h>
double line(double x1,double y1,double x2,double y2) // длина отрезка
{
    return sqrt(pow(x1-x2,2)+pow(y1-y2,2));
}
double square(double a, double b, double c) // площадь треугольника
{
    double s, p=(a+b+c)/2;
    return s=sqrt(p*(p-a)*(p-b)*(p-c)); // формула Герона
}
void main()
{
    double x1=2, y1=3, x2=4, y2=6, x3=7, y3=9; // координаты
    double line1, line2, line3; // переменные-стороны
    line1 = line(x1,y1,x2,y2); // вычисление длин сторон треугольника
    line2 = line(x1,y1,x3,y3);
    line3 = line(x2,y2,x3,y3);
    cout<<"S="<<square(line1,line2,line3)<<endl;
}
```

■ Прототип функции

- Для того, чтобы к функции можно было обратиться, в том же файле должно находиться определение или описание функции (прототип), например:

```
double line(double x1, double y1, double x2, double y2);  
double square(double a, double b, double c);
```

- При наличии прототипов вызываемые функции не обязаны размещаться в одном файле с вызывающей функцией, а могут оформляться в виде отдельных модулей.
- При разработке программ, которые состоят из большого количества функций, размещенных в разных модулях, прототипы функций и описания внешних объектов (констант, переменных, массивов) помещают в отдельный файл, который включают в начало каждого из модулей программы с помощью директивы `#include "имя_файла"`.

- **Параметры функции**
- Основным способом обмена информацией между вызываемой и вызывающей функциями является механизм параметров. Существует два способа передачи параметров в функцию: **по адресу и по значению**.
- При передаче по значению выполняются следующие действия:
 - вычисляются значения выражений, стоящие на месте фактических параметров;
 - в стеке выделяется память под формальные параметры функции;
 - каждому фактическому параметру присваивается значение формального параметра, при этом проверяются соответствия типов и при необходимости выполняются их преобразования.

```
double square(double a, double b, double c)
{
    double s, p = (a+b+c)/2;
    return s = sqrt(p*(p-a)*(p-b)*(p-c)); //формула Герона
}
```

- Передача по адресу:

```
void change (int* a, int* b)
{
    int r=*a;
    *a=*b;
    *b=r;
}
```

Вызов:

```
int x=1, y=5;
change (&x, &y) ;
```

- Для передачи по адресу также могут использоваться **ссылки**. При передаче по ссылке в функцию передается адрес указанного при вызове параметра, а внутри функции все обращения к параметру неявно разыменовываются.

```
void change (int& a, int& b)
{
    int r=a;
    a=b;
    b=r;
}
```

Вызов:

```
int x=1, y=5;
change (x, y) ;
```

- Использование ссылок вместо указателей улучшает читаемость программы, поскольку не надо применять операцию разыменовывания. Использование ссылок вместо передачи по значению также более эффективно, т. к. не требует копирования параметров.
- Если требуется запретить изменение параметра внутри функции, используется модификатор `const`. Рекомендуется ставить `const` перед всеми параметрами, изменение которых в функции не предусмотрено (по заголовку будет понятно, какие параметры в ней будут изменяться, а какие нет).

■ Локальные и глобальные переменные

- Переменные, которые используются внутри данной функции, называются **локальными**. Память для них выделяется в стеке, поэтому после окончания работы функции они удаляются из памяти. Нельзя возвращать указатель на локальную переменную, т. к. память, выделенная такой переменной, будет освобождаться.

```
int* f()  
{  
    int a;  
    //...  
    return &a;    //ОШИБКА!  
}
```

- **Глобальные переменные** – это переменные, описанные вне функций. Они видны во всех функциях, где нет локальных переменных с такими именами.

```
int a,b;      //глобальные переменные
void change()
{
    int r;    //локальная переменная
    r = a;
    a = b;
    b = r;
}
void main()
{
    cin>>a,b;
    change();
    cout<<"a="<<a<<"b="<<b;
}
```

- Глобальные переменные также можно использовать для передачи данных между функциями, но этого не рекомендуется делать, т. к. это затрудняет отладку программы и препятствует помещению функций в библиотеки. Нужно стремиться к тому, чтобы функции были максимально независимы.

Функции в С

- **Функции и массивы**
- При использовании массива как параметра функции, в функцию передается указатель на его первый элемент. При этом теряется информация о количестве элементов в массиве, поэтому размерность массива следует передавать как отдельный параметр.

- **Удалить из массива все четные элементы**

//формирование массива

```
int form(int a[100]) //массив передается в функцию по адресу
```

```
{
```

```
int n;
```

```
cout<<"\nEnter n";
```

```
cin>>n;
```

```
for(int i=0;i<n;i++)
```

```
a[i] = rand()%100;
```

```
return n;
```

```
}
```

//печать массива

```
void print(int a[100],int n)
```

```
{
```

```
for(int i=0;i<n;i++)
```

```
cout<<a[i]<<" ";
```

```
cout<<"\n";
```

```
}
```

Функции в С

*/*удаление четных элементов из массива, массив передается по адресу,
количество элементов передается по адресу как ссылка */*

```
void del(int a[100],int& n)
{
    int j=0,i,b[100];
    for(i=0;i<n;i++)
        if(a[i]%2!=0)
        {
            b[j]=a[i];
            j++;
        }
    n=j;
    for(i=0;i<n;i++)
        a[i]=b[i];
}

void main()
{
    int a[100];
    int n;
    n=form(a); // вызов функции формирования массива
    print(a,n); // вызов функции печати массива
    del(a,n);   // вызов функции удаления элементов массива
    print(a,n); // вызов функции печати массива
}
```

Функции в С

- При использовании динамических массивов, в функцию можно передавать указатель на область динамической памяти, в которой размещаются элементы массива.
- **Передача строк в качестве параметров функций**
- Строки при передаче в функции могут передаваться как одномерные массивы типа `char` или как указатели типа `char*`.

//Найти количество гласных букв в строке

```
int find(char* s, char c) //Функция поиска заданного символа в строке
{
    for (int i=0;i<strlen(s);i++)
        if(s[i]==c) return i;
    return -1;
}

void main()
{
    char s[255];
    gets(s)
    char gl = "aouiey"; //строка, которая содержит гласные
    for(int i=0,k=0;i<strlen(s);i++)
        if(find(gl,s[i])>0) k++;
    printf("%d",k);
}
```

- **Передача многомерных массивов в функцию**
- При передаче многомерных массивов в функцию все размерности должны передаваться в качестве параметров. По определению многомерные массивы в С и С++ не существуют. Если мы описываем массив с несколькими индексами, например, массив `int mas[3][4]`, то это означает, что мы описали одномерный массив `mas`, элементами которого являются указатели на одномерные массивы `int[4]`.
- Например: передача матрицы в функцию транспонирования:

```
void transp(int a[][N], int n)
{
    int r;
    for(int i=0; i<n; i++)
        for(int j=0; j<n; j++)
            if(i<j)
            {
                r = a[i][j];
                a[i][j] = a[j][i];
                a[j][i] = r;
            }
}
```

`transp(N, mas);` <—————>

Вызов этой функции

- **Функции с начальными значениями параметров (по умолчанию)**
- В определении функции может содержаться начальное (умалчиваемое) значение параметра. Это значение используется, если при вызове функции соответствующий параметр опущен. Все параметры, описанные справа от такого параметра, также должны быть умалчиваемыми.

```
void print(char* name = "Номер дома: ", int value = 1)
{
    cout<<"\n"<<name<<value;
}
```

- **Вызовы:**

```
print(); // вывод: Номер дома: 1
print("Номер квартиры", 15); // вывод: Номер квартиры: 15
print(, 15); // ошибка, т. к. параметры можно
// опускать только с конца
```

- **Подставляемые (inline) функции**
- Некоторые функции в С можно определить с использованием служебного слова `inline`. Такая функция называется подставляемой или встраиваемой.

*/*функция возвращает расстояние от точки с координатами (x1,y1)
(по умолчанию центр координат) до точки с координатами(x2,y2)*/*

```
inline float line(float x1,float y1,float x2=0,float y2=0)
{
    return sqrt(pow(x1-x2)+pow(y1-y2,2));
}
```

- **Подставляемые (inline) функции**
- Обработывая каждый вызов подставляемой функции, компилятор пытается подставить в текст программы код операторов ее тела.
- Подставляемые функции используют, если тело функции состоит из нескольких операторов.
- Подставляемыми не могут быть:
 - рекурсивные функции;
 - функции, у которых вызов размещается до ее определения;
 - функции, которые вызываются более одного раза в выражении;
 - функции, содержащие циклы, переключатели и операторы переходов;
 - функции, которые имеют слишком большой размер, чтобы сделать подстановку.

- **Функции с переменным числом параметров**
- В языке С допустимы функции, у которых при компиляции не фиксируется число параметров, и, кроме того, может быть неизвестен тип этих параметров.
- Количество и тип параметров становится известным только в момент вызова, когда явно задан список фактических параметров. Каждая функция с переменным числом параметров должна иметь хотя бы один обязательный параметр.
- Определение функции с переменным числом параметров:
тип имя (явные параметры, ...)
{ тело функции }
- При обращении к функции все параметры и обязательные, и необязательные будут размещаться в памяти друг за другом. Следовательно, определив адрес обязательного параметра как $p = \&k$, где p - указатель, а k – обязательный параметр, можно получить адреса и всех остальных параметров: оператор $k++$ выполняет переход к следующему параметру списка.

■ Функции с переменным числом параметров

- Для доступа к списку параметров используется указатель `p` типа `int*`. Он устанавливается на начало списка параметров в памяти, а затем перемещается по адресам фактических параметров (`++p`).

```
/* Найти среднее арифметическое последовательности чисел, если  
   известен признак конца списка параметров */
```

```
#include <iostream.h>
```

```
int summ(int k, ...)
```

```
{
```

```
    int* p = &k;      // настроили указатель на адрес параметра k
```

```
    int s = *p;       // значение первого параметра присвоили s
```

```
    for(int i=1; p!=0; i++) // пока не конец списка
```

```
        s += *(++p);
```

```
    return s/(i-1);
```

```
}
```

```
void main()
```

```
{
```

```
    cout<<"\n4+6="<<summ(4, 6, 0);
```

```
    cout<<"\n1+2+3+4="<<summ(1, 2, 3, 4, 0);
```

```
}
```

- **Перегрузка функций**
- Цель перегрузки состоит в том, чтобы функция с одним именем по-разному выполнялась и возвращала разные значения при обращении к ней с различными типами и различным числом фактических параметров.
- Для обеспечения перегрузки необходимо для каждой перегруженной функции определить возвращаемые значения и передаваемые параметры так, чтобы каждая перегруженная функция отличалась от другой функции с тем же именем.
- Компилятор определяет, какую функцию выбрать по типу фактических параметров.

Функции в С

```
#include <iostream.h>
#include <string.h>
//сравнение двух целых чисел
int max(int a, int b)
{
    if (a>b) return a;
    else     return b;
}

//сравнение двух вещественных чисел
float max(float a, float b)
{
    if(a>b) return a;
    else     return b;
}

//сравнение двух строк
char* max(char* a, char* b)
{
    if (strcmp(a,b)>0) return a;
    else return b;
}
```

Функции в C

```
void main()
{
    int a1,b1;
    float a2, b2;
    char s1[20];
    char s2[20];
    cout<<"\nfor int:\n";
    cout<<"a=";>>a1;
    cout<<"b=";>>b1;
    cout<<"\nMAX="<<max(a1,b1)<<"\n";
    cout<<"\nfor float:\n";
    cout<<"a=";>>a2;
    cout<<"b=";>>b2;
    cout<<"\nMAX="<<max(a2,b2)<<"\n";
    cout<<"\nfor char*:\n";
    cout<<"a=";>>s1;
    cout<<"b=";>>s2;
    cout<<"\nMAX="<<max(s1,s2)<<"\n";
}
```

- Правила описания перегруженных функций:
 - Перегруженные функции должны находиться в одной области видимости.
 - Перегруженные функции могут иметь параметры по умолчанию, при этом значения одного и того же параметра в разных функциях должны совпадать. В разных вариантах перегруженных функций может быть разное количество умалчиваемых параметров.
 - Функции не могут быть перегружены, если описание их параметров отличается только модификатором `const` или наличием ссылки.

■ Шаблоны функций

- Шаблоны вводятся для того, чтобы автоматизировать создание функций, обрабатывающих разнотипные данные. Например, алгоритм сортировки можно использовать для массивов различных типов.
- При перегрузке функции для каждого используемого типа определяется своя функция. Шаблон функции определяется один раз, но определение параметризуется, т. е. тип данных передается как параметр шаблона.

```
template <class имя_типа [,class имя_типа]>  
заголовок_функции  
{  
    тело функции  
}
```

- Шаблон функций состоит из 2 частей – заголовка шаблона: `template<список параметров шаблона>` и обыкновенного определения функции, в котором вместо типа возвращаемого значения и/или типа параметров, записывается имя типа, определенное в заголовке шаблона.

- Шаблон функции, которая находит абсолютное значение числа любого типа:

```
template<class type>    // type - это имя параметризуемого типа
type abs(type x)
{
    if(x<0) return -x;
    else return x;
}
```

- Шаблон служит для автоматического формирования конкретных описаний функций по тем вызовам, которые компилятор обнаруживает в программе.
- Например, если в программе вызов функции осуществляется как
abs(-1.5)
то компилятор сформирует определение функции
double abs(double x) {...}.

- Шаблон функции, которая меняет местами две переменных:

```
template <class T> // T - имя параметризуемого типа
void change (T* x, T* y)
{
    T z=*x;
    *x=*y;
    *y=z;
}
```

Вызов этой функции может быть:

```
long k=10, l=5;
change (&k, &l);
```

Тогда, компилятор сформирует определение:

```
void change (long* x, long* y)
{
    long z=*x;
    *x=*y;
    *y=z;
}
```

- Шаблон функции, которая находит номер максимального элемента в массиве:

```
#include <iostream.h>
template <class Data>
Data& rmax(int n, Data a[])
{
    int im=0;
    for(int i=0;i<n;i++)
        if(a[i]>a[im]) im=i;
    return a[im];    // возвращает ссылку
}
```

Функции в С

```
void main()
{
    int n = 5;
    int x[] = {10, 20, 30, 15};
    // найти номер максимального элемента в массиве целых чисел
    cout<<"\nrmax(n, x)="<<rmax(n, x)<<"\n";
    rmax(n, x)=0;        // заменить максимальный элемент на 0
    for(int i=0; i<n; i++) // вывод массива
        cout<<x[i]<<" ";
    cout<<"\n";
    // следующий массив
    float y[]={10.4, 20.2, 30.6, 15.5};
    // найти номер максимального элемента в массиве вещественных
    // чисел
    cout<<"\nrmax(n, y)="<<rmax(n, y)<<"\n";
    rmax(4, y)=0;        // заменить максимальный элемент на 0
    for(int i=0; i<n; i++) // вывод массива
        cout<<y[i]<<" ";
}
```

■ Результаты работы программы:

rmax(n, x)=30

10 20 0 15

rmax(n, y)=30.6

10.4 20.2 0 15.5

- Основные свойства параметров шаблона функций:
 - Имена параметров должны быть уникальными во всем определении шаблона.
 - Список параметров шаблона не может быть пустым.
 - В списке параметров шаблона может быть несколько параметров, каждый из них начинается со слова `class`.

- **Указатель на функцию**
- С функцией можно проделать только две вещи: вызвать ее и получить ее адрес. Указатель, полученный взятием адреса функции, можно затем использовать для вызова функции.
- Каждая функция характеризуется типом возвращаемого значения, именем и списком типов ее параметров. Если имя функции использовать без последующих скобок и параметров, то оно будет выступать в качестве указателя на эту функцию, и его значением будет выступать адрес размещения функции в памяти.
- Указатель на функцию определяется следующим образом:
тип_функции (*имя_указателя) (спецификация параметров)

```
int f1(char c) {...}           // определение функции
int (*ptrf1)(char);           // определение указателя на функцию f1
char* f2(int k, char c) {...} // определение функции
char* ptrf2(int, char);        // определение указателя на функцию f2
```
- В определении указателя количество и тип параметров должны совпадать с соответствующими типами в определении функции, на которую ставится указатель.

```
#include <iostream.h>
void f1 ()
{
    cout<<"\nfunction f1";
}
void f2 ()
{
    cout<<"\nfunction f2";
}
void main()
{
    void(*ptr) (); // указатель на функцию
    ptr = f2;      // указателю присваивается адрес функции f2
    (*ptr) ();     // вызов функции f2
    ptr = f1;      // указателю присваивается адрес функции f1
    (*ptr) ();     // вызов функции f1с помощью указателя
}
```

- При определении указатель на функцию может быть сразу проинициализирован:

```
void (*ptr)() = f1;
```

- Указатели и функции могут быть объединены в массивы.

Например, `float (*ptrMas[4])(char)` -

описание массива, который содержит 4 указателя на функции. Каждая функция имеет параметр типа `char` и возвращает значение типа `float`.

Обратиться к такой функции можно следующим образом:

```
float a = (*ptrMas[1])('f'); // обращение ко второй функции
```

Функции в С

```
#include <iostream.h>
#include <stdlib.h>
void f1()
{
    cout<<"\nThe end of work";exit(0);
}
void f2()
{
    cout<<"\nThe work #1";
}
void f3()
{
    cout<<"\nThe work #2";
}
void main()
{
    void(*fptr[]) ()={f1,f2,f3}; // массив и указателей на функции
    int n;
    while(1) // бесконечный цикл
    {
        cout<<"\n Enter the number";
        cin>>n;
        fptr[n] (); // вызов функции с номером n
    }
}
```


- Указатели на функции удобно использовать в тех случаях, когда функцию надо передать в другую функцию как параметр.

```
#include <iostream.h>
#include <math.h>
typedef float (*fptr) (float); // тип-указатель на функцию

/*решение уравнения методом половинного деления, уравнение
передается с помощью указателя на функцию */
float root(fptr f, float a, float b, float e)
{
    float x;
    do
    {
        x=(a+b)/2; // находим середину отрезка
        if ((*f) (a) *f (x)<0) // выбираем отрезок
            b=x;
        else a=x;
    }
    while ((*f) (x)>e && fabs (a-b)>e);
    return x;
}
```

Функции в С

```
//функция, для которой ищется корень
float testf(float x)
{
    return x*x-1;
}
void main()
{
    /*в функцию root передается указатель на функцию, координаты
    отрезка и точность*/
    float es = root(testf,0,2,0.0001);
    cout<<"\nX="<<res;
}
```

- Подобно указателю на функцию определяется и ссылка на функцию:

тип_функции (&имя_ссылки) (параметры)
инициализирующее_выражение;


```
int f(float a, int b) {...}    // определение функции
int (&fref)(float, int)=f;    // определение ссылки
```

- Использование имени функции без параметров и скобок будет восприниматься как адрес функции. Изменить значение ссылки на функцию нельзя, поэтому более широко используются указатели на функции, а не ссылки.

```
#include <iostream.h>
void f(char c)
{
    cout<<"\n"<<c;
}
void main()
{
    void (*pf)(char);    // указатель на функцию
    void (&rf)(char);    // ссылка на функцию
    f('A');              // вызов по имени
    pf = f;              // указатель ставится на функцию
    (*pf)('B');          // вызов с помощью указателя
    rf('C');             // вызов по ссылке
}
```

Контрольные вопросы

- 1) Что такое функция?
- 2) Что указывается при вызове функции?
- 3) Что такое прототип функции?
- 4) Что такое параметры функции и какие существуют способы передачи их в функцию?
- 5) Что происходит при передаче параметров в функцию по значению?
- 6) Что происходит при передаче параметров в функцию по адресу?
- 7) Что такое локальные и глобальные переменные и чем они различаются?
- 8) Что передается в функцию при использовании массива как параметра функции?
- 9) Как могут передаваться строки при передаче в функцию?
- 10) Как определяются функции с начальными значениями параметров?
- 11) Что такое подставляемая функция?
- 12) Какие функции не могут быть подставляемыми?
- 13) Функции с переменным числом параметров.
- 14) Что такое перегрузка функций и как она осуществляется?
- 15) Для чего вводятся шаблоны функций ?
- 16) Какие основные свойства параметров шаблона?
- 17) Указатель на функцию?



Типы данных, определяемые пользователем

Типы данных, определяемые пользователем

- **Переименование типов**

- Объявление, начинающееся с ключевого слова `typedef`, вводит новое имя для типа, а не для переменной данного типа.

```
typedef тип имя_типа [размерность];
```

- Целью такого объявления часто является назначение короткого синонима для часто используемого типа. Например:

```
typedef unsigned int UINT;
```

Такое имя можно затем использовать также как и стандартное имя типа:

```
UINT a, b, c;    //переменные типа unsigned int
```

- Другое использование `typedef` – свести в одно место все непосредственные ссылки на какой-то тип. Например:

```
typedef int int32;
```

При необходимости переноса приложения на машину с `sizeof(int) == 2` достаточно будет заменить единственную строку кода с `int`:

```
typedef long int32;
```

- Имена, вводимые, являются синонимами, а не новыми типами. Следовательно, старые типы можно использовать совместно с их синонимами

Типы данных, определяемые пользователем

- **Перечисления**
- Если надо определить несколько именованных констант таким образом, чтобы все они имели разные значения, можно воспользоваться перечисляемым типом:
`enum [имя_типа] {список констант};`
- Константы должны быть целочисленными и могут инициализироваться обычным образом. Если инициализатор отсутствует, то первая константа обнуляется, а остальным присваиваются значения на единицу большее, чем предыдущее.

```
Enum Err{ErrRead, ErrWrite, ErrConvert};  
Err error;  
// ...  
switch(error)  
{  
    case ErrRead:    // ...  
    case ErrWrite:   // ...  
    case ErrConvert: // ...  
}
```

Типы данных, определяемые пользователем

■ Структуры

- Структура – это объединенное в единое целое множество поименованных элементов данных. Элементы структуры (поля) могут быть различного типа, они все должны иметь различные имена. Форматы определения структурного типа следующие:

```
struct имя_типа      // способ 1
{
    тип1 элемент1;
    тип2 элемент2;
    //...
};
```

■ Пример:

```
struct Date           // определение структуры
{
    int day;
    int month;
    int year;
};

Date birthday;        // переменная типа Date
```


Типы данных, определяемые пользователем

```
struct    //способ 2
{
    тип1 элемент1;
    тип2 элемент2;
    //...
} список идентификаторов;
```

■ Пример:

```
struct
{
    int min;
    int sec;
    int msec;
} time_beg, time_end;
```

- В первом случае описание структур определяет новый тип, имя которого можно использовать наряду со стандартными типами.
- Во втором случае описание структуры служит определением переменных.

Типы данных, определяемые пользователем

Структуры

- Структурный тип можно также задать с помощью ключевого слова `typedef`:

```
typedef struct    // способ 3
{
    float re;
    float im;
} complex;
```

```
complex a[100]; // массив из 100 комплексных чисел.
```

Типы данных, определяемые пользователем

- **Работа со структурами.**

- 1. **Инициализация структур.**

- Для инициализации структур значения ее полей перечисляют в фигурных скобках.

- **Пример 1:**

```
struct student
{
    char name[20];
    int kurs;
    float rating;
};
student s = {"Иванов", 1, 3.5};
```

- **Пример 2:**

```
struct
{
    char name[20];
    char title[30];
    float rate;
} employee = {"Петров", "программист", 10000};
```

Типы данных, определяемые пользователем

2. Присваивание структур

- Для переменных одного и того же структурного типа определена операция присваивания. При этом происходит поэлементное копирование.

```
student t = s;
```

Типы данных, определяемые пользователем

3. Доступ к элементам структур

- Доступ к элементам структур обеспечивается с помощью уточненных имен:

имя_структуры.имя_элемента

```
#include <iostream.h>
void main()
{
    struct student
    {
        char name[30];
        char group[10];
        float rating;
    };
    student mas[35];          // массив структур
    for(int i=0;i<35;i++)    // ввод значений массива
    {
        cout<<"\nEnter name:";    cin>>mas[i].name;
        cout<<"\nEnter group:";   cin>>mas[i].group;
        cout<<"\nEnter rating:";  cin>>mas[i].rating;
    }
    cout<<"Rating <3:";
    for(i=0; i<35; i++)// вывод студентов, у которых рейтинг меньше 3
    if(mas[i].name<3)
    cout<<"\n"<<mas[i].name;
}
```

Типы данных, определяемые пользователем

4. Указатели на структуры

- Указатели на структуры определяются также как и указатели на другие типы.

```
Student* ps;
```

- Можно ввести указатель для типа `struct`, не имеющего имени :

```
struct
{
    char *name;
    int age;
} *person; // указатель на структуру
```

- При определении указатель на структуру может быть сразу же проинициализирован:

```
Student* ps = &mas[0];
```

- Указатель на структуру обеспечивает доступ к ее элементам 2 способами:

- `(*указатель).имя_элемента`
- `указатель->имя_элемента`

```
cin>>(*ps).name;
```

```
cin>>ps->title;
```

Типы данных, определяемые пользователем

- Битовые поля – это особый вид полей структуры. При описании битового поля указывается его длина в битах (целая положительная константа).

```
struct
{
    int a:10;    //длина поля 10 бит
    int b:14;    //длина поля 14 бит
} xx, *pxx;
```

```
xx.a = 1;
pxx = &xx;
pxx->b = 8;
```

- Битовые поля могут быть любого целого типа. Они используются для плотной упаковки данных. Например, с их помощью удобно реализовать флажки типа "да" / "нет".
- Особенностью битовых полей является то, что нельзя получить их адрес.
- Размещение битовых полей в памяти зависит от компилятора и аппаратуры.

Типы данных, определяемые пользователем

- **Объединения**
- Объединение (`union`) – это частный случай структуры.
- Все поля объединения располагаются по одному и тому же адресу.
- Длина объединения равна наибольшей из длин его полей.
- В каждый момент времени в такой переменной может храниться только одно значение.
- Объединения применяют для экономии памяти, если известно, что более одного поля не потребуется. Также объединение обеспечивает доступ к одному участку памяти с помощью переменных разного типа.

```
union
{
    char s[10];
    int x;
} u1;
```

0 1 2 3 4 5 6 7 8 9

x – занимает 4 байта

s – занимает 10 байтов

Типы данных, определяемые пользователем

- Пример использования объединения :

```
enum paytype {CARD,CHECK};      //тип оплаты
struct
{
    /*поле, которое определяет, с каким полем объединения будет
    выполняться работа*/
    paytype ptype;
    union
    {
        char card[25];
        long check;
    };
} info;

switch (info.ptype)
{
    case CARD:  cout<<"\nОплата по карте:"<<info.card; break;
    case CHECK: cout<<"\nОплата чеком:"<<info.check; break;
}
```

Контрольные вопросы

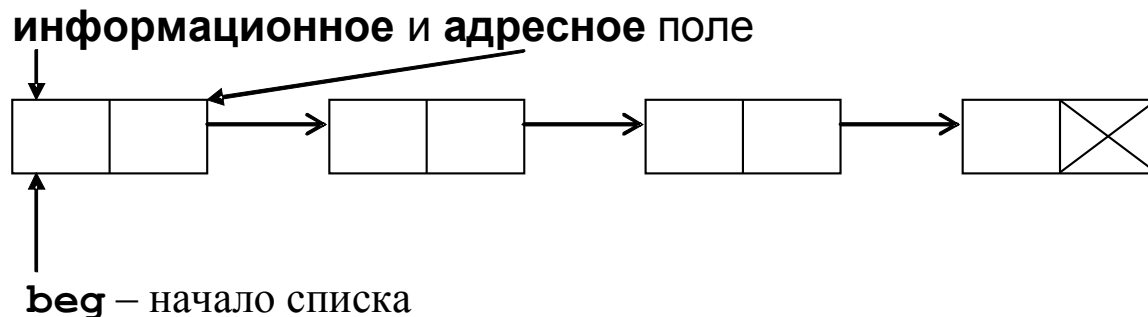
- 1) Что такое переименование типов и для чего оно нужно?
- 2) Что такое перечисления?
- 3) Что такое структура? Какими могут быть элементы структуры?
- 4) С помощью чего можно задать структурный тип?
- 5) Как инициализируются структуры? Как происходит доступ к элементам структуры?
- 6) Указатели на структуру.
- 7) Объединения.



Динамические структуры данных

Динамические структуры данных

- Во многих задачах требуется использовать данные, у которых конфигурация, размеры и состав могут меняться в процессе выполнения программы. Для их представления используют динамические информационные структуры. К таким структурам относят:
 - линейные списки;
 - стеки;
 - очереди;
 - бинарные деревья;
- Они отличаются способом связи отдельных элементов и допустимыми операциями. Динамическая структура может занимать несмежные участки динамической памяти.
- Наиболее простой динамической структурой является линейный однонаправленный список, элементами которого служат объекты структурного типа.



Динамические структуры данных

- Описание простейшего элемента такого списка выглядит следующим образом:

```
struct имя_типа
{
    информационное_поле;
    адресное_поле;
};
```

- информационное поле – это поле любого типа;
- адресное поле – это указатель на объект того же типа, что и определяемая структура, в него записывается адрес следующего элемента списка.
- Информационных полей может быть несколько.

```
struct point
{
    char* name;        // информационное поле
    int age;           // информационное поле
    point* next;       // адресное поле
};
```

Динамические структуры данных

- Над списками можно выполнять следующие операции:
 - начальное формирование списка (создание первого элемента);
 - добавление элемента в конец списка;
 - добавление элемента в начало списка;
 - поиск элемента с заданным ключом;
 - удаление элемента с заданным ключом;
 - удаление элемента с заданным номером;
 - добавление элемента номером;
 - добавление элемента в заданное место списка (до или после элемента с заданным ключом или номером);
 - упорядочивание списка по ключу.

Динамические структуры данных

//Создание и печать однонаправленного списка

```
#include <iostream.h>
```

```
#include <string.h>
```

// описание структуры

```
struct point
```

```
{
```

```
    char* name;           // информационное поле
```

```
    int age;              // информационное поле
```

```
    point* next;          // адресное поле
```

```
};
```

// создание одного элемента

```
point* make_point()
```

```
{
```

```
    point* p = new(point); // выделить память под элемент списка
```

```
    char s[20];           // вспомогательная переменная для ввода строки
```

```
    cout<<"\nEnter the name:";
```

```
    cin>>s;
```

```
    p->name=new char[strlen(s)+1]; // память под строку символов
```

```
    strcpy(p->name,s); // записать информацию в информационное поле
```

```
    cout<<"\nEnter the age";
```

```
    cin>>p->age;
```

```
    p->next=0;             // сформировать адресное поле
```

```
    return p;             // вернуть указатель на созданный элемент
```

```
}
```

Динамические структуры данных

```
// печать информационных полей одного элемента списка
void print_point(point* p)
{
    cout<<"\nNAME:"<<p->name;
    cout<<"\nAGE:"<<p->age;
    cout<<"\n-----\n";
}
// формирование списка из n элементов
point* make_list(int n)
{
    point* beg=make_point(); // сформировать первый элемент
    point* r; // вспомогательная переменная для добавления
    for(int i=1;i<n;i++)
    {
        r = make_point(); // сформировать следующий элемент

        // добавление в начало списка
        r->next = beg;
        beg = r;
    }
    return beg; // вернуть адрес начала списка
}
```


Динамические структуры данных

```
// печать списка, на который указывает указатель beg
int print_list(point* beg)
{
    point* p=beg; // p присвоить адрес 1-го элемента списка
    int k=0;       // счетчик количества напечатанных элементов
    while(p)      // пока нет конца списка
    {
        // печать элемента, на который указывает элемент p
        print_point(p);
        p = p->next; // переход к следующему элементу
        k++;
    }
    return k; // количество элементов в списке
}

void main()
{
    int n; // количество элементов в списке
    cout<<"\nEnter the size of list";
    cin>>n;
    point* beg = make_list(n); // формирование списка
    if(!print_list(beg))      // печать списка
        cout<<"\nThe list is empty";
}
```

Динамические структуры данных

- Удаление из однонаправленного списка элемента с номером k :

// удаление элемента с номером k

```
point* del_point(point* beg, int k)
{
    // поставить вспомогательную переменную на начало списка
    point* p = beg;
    point* r;    // вспомогательная переменная для удаления
    int i=0;     // счетчик элементов в списке
    if(k==0)     // если удалить первый элемент
    {
        beg = p->next;
        delete[] p->name;    // удалить динамическое поле
        delete[] p;          // удалить элемент из списка
        return beg;          // вернуть адрес первого элемента
    }

    /*...*/
}
```

Динамические структуры данных

```
while(p)    //пока не конец списка
{
    /*дошли до элемента с номером k-1, чтобы поменять его
    поле next*/
    if(I == k-1)
    {
        // поставить r на удаляемый элемент
        r=p->next;
        if(r)    // если r не последний элемент
        {
            p->next=r->next;    // исключить r из списка
            delete[]r->name;    // удалить динамическое поле
            delete[]r;          // удалить элемент из списка
        }
        /*если p - последний, то полю next присвоить 0*/
        else p->next = 0;
    }
    p=p->next;    // переход к следующему элементу
    i++;         // увеличить счетчик элементов
}
return beg;     // вернуть адрес первого элемента
}
```

Контрольные вопросы

- 1) Какие существуют динамические структуры и чем они отличаются?
- 2) Что такое список и какие операции с ним можно выполнять?



Условная компиляция

Условная компиляция

- Условная компиляция обеспечивается набором препроцессорных директив, которые управляют препроцессорной обработкой текста программы:

<code>#if</code>	константное_выражение	
<code>#ifdef</code>	препроцессорный_идентификатор	выполняют проверку условий
<code>#ifndef</code>	препроцессорный_идентификатор	
<code>#else</code>		определяют диапазон действия
<code>#endif</code>		проверяемого условия
<code>#elif</code>		используется для организации
		мультиветвлений

- нуля, то считается, что проверяемое условие истинно.
- `#ifdef` препроцессорный_идентификатор – проверяет, определен ли с помощью команды `#define` препроцессорный идентификатор.
- `#ifndef` препроцессорный_идентификатор – проверяет неопределенность препроцессорного идентификатора, т. е. тот случай, когда идентификатор не был использован командой `#define`.
- `#endif` – заканчивает выполнение `#if`.

Условная компиляция

- Файлы, предназначенные для препроцессорного включения в модули программы, обычно снабжают защитой от повторного включения. Такое повторное включение может произойти, если несколько модулей, в каждом из которых в текст с помощью директивы `#include` включаются одни и те же файлы, объединяются в общий текст программы. Схема защиты от повторного включения может быть следующей:

```
//Файл и именем prim.h
#ifndef PRIM_H
#define PRIM_H
#include < ... > //заголовочные файлы
//текст модуля
#endif
```

- Здесь `PRIM_H` – зарезервированный для файла `prim.h` препроцессорный идентификатор, который не должен встречаться в других текстах программы.