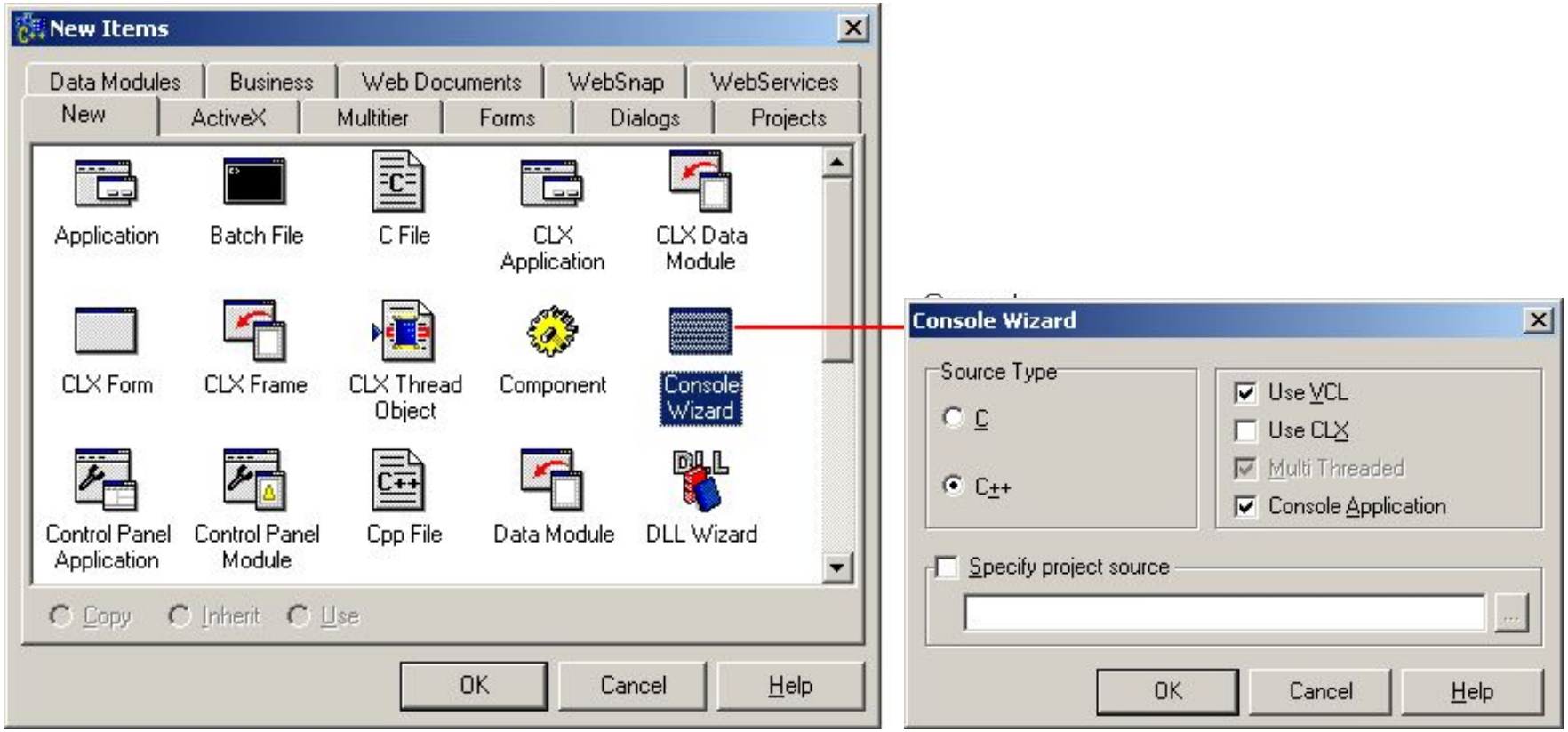
# Алгоритмизация и программирование

# Создание консольного приложения

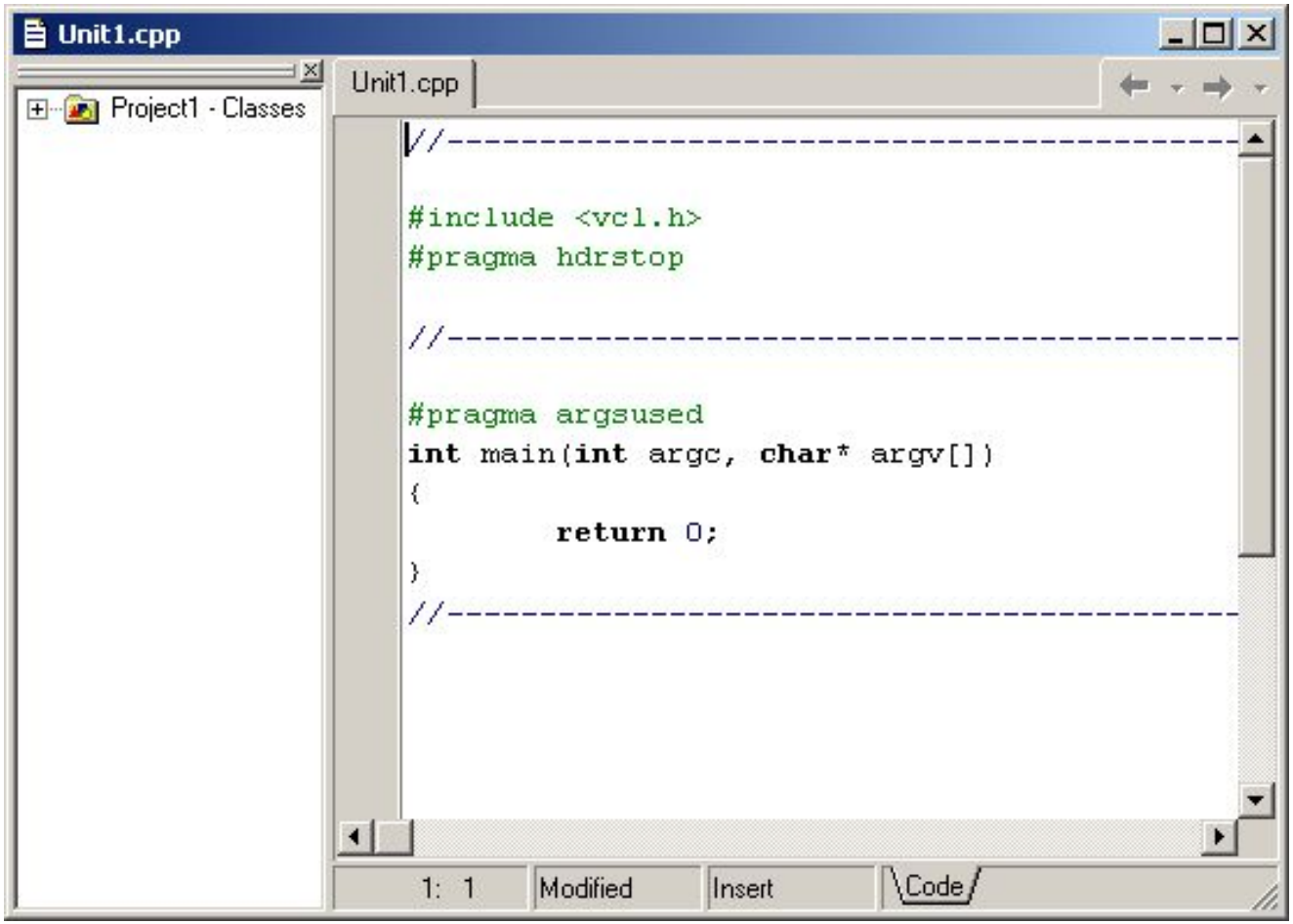
# Консольное приложение

- **Консольное приложение**– это приложение, которое для взаимодействия с пользователем не использует графический интерфейс.
- Устройством, обеспечивающим взаимодействие с пользователем, является консоль – клавиатура и монитор.
- В операционной системе консольное приложение работает в окне командной строки.

# Создание консольного приложения в C++ Builder 6

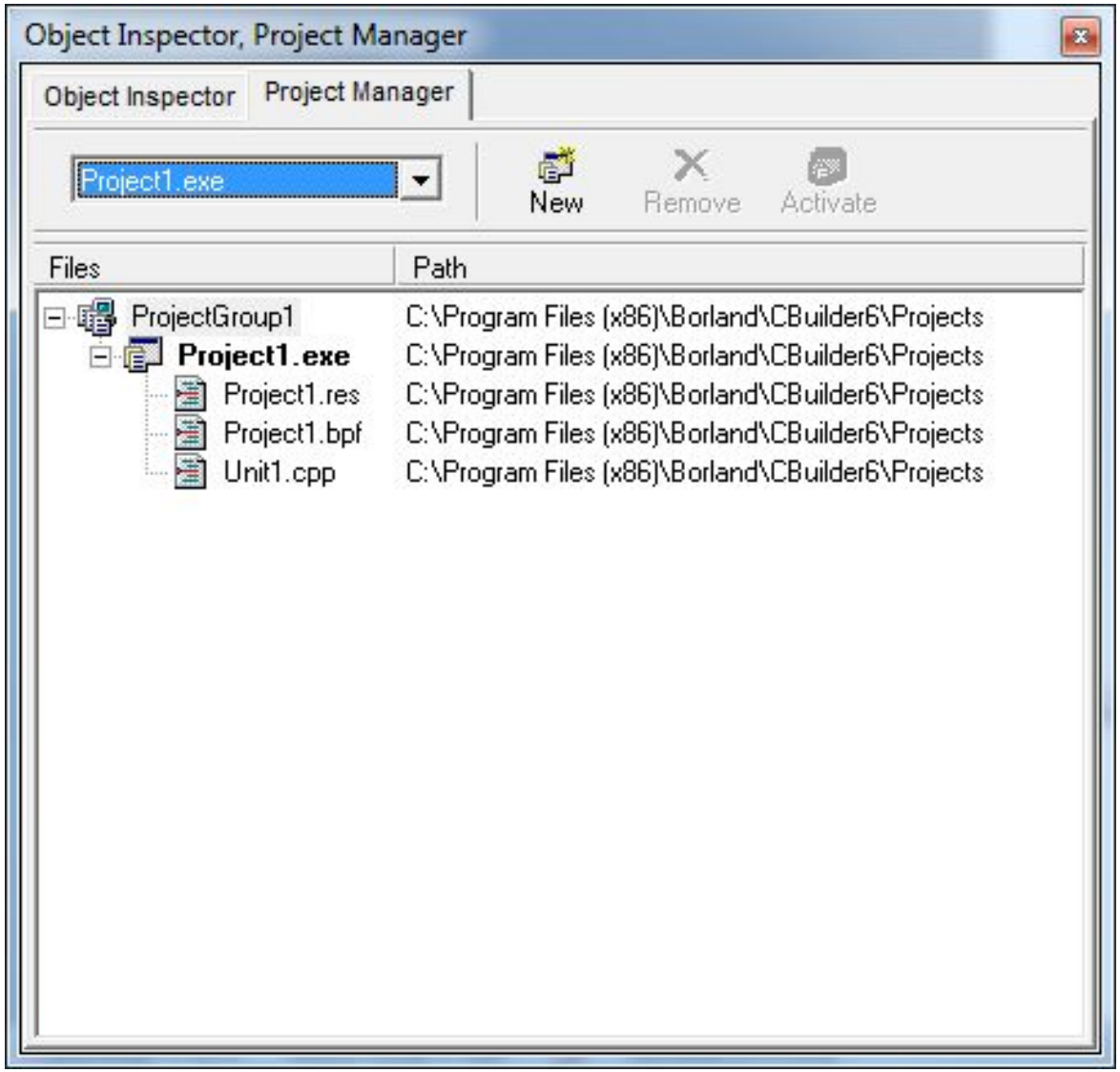


C++ Builder создаст проект консольного приложения и на экране появится окно редактора кода, в котором находится шаблон консольного приложения – функция **main**.



- Директива препроцессора #include <vcl\vcl.h> предназначена для включения в текст проекта заголовочного файла, ссылающегося на описания классов библиотеки компонентов.
- Директива **#pragma hdrstop** запрещает выполнение предварительной компиляции подключаемых файлов.
- Затем вставляются директивы **#include**, обеспечивающие подключение необходимых библиотек.
- Директива **#pragma argsused** отключает предупреждение компилятора о том, что аргументы, указанные в заголовке функции, не используются.
- Консольное приложение разрабатывается в Windows, а выполняется как программа DOS. В DOS используется кодировка ASCII, а в Windows – ANSI. Это приводит к тому, что консольное приложение вместо сообщений на русском языке выводит "абракадабру".

# Файлы проекта

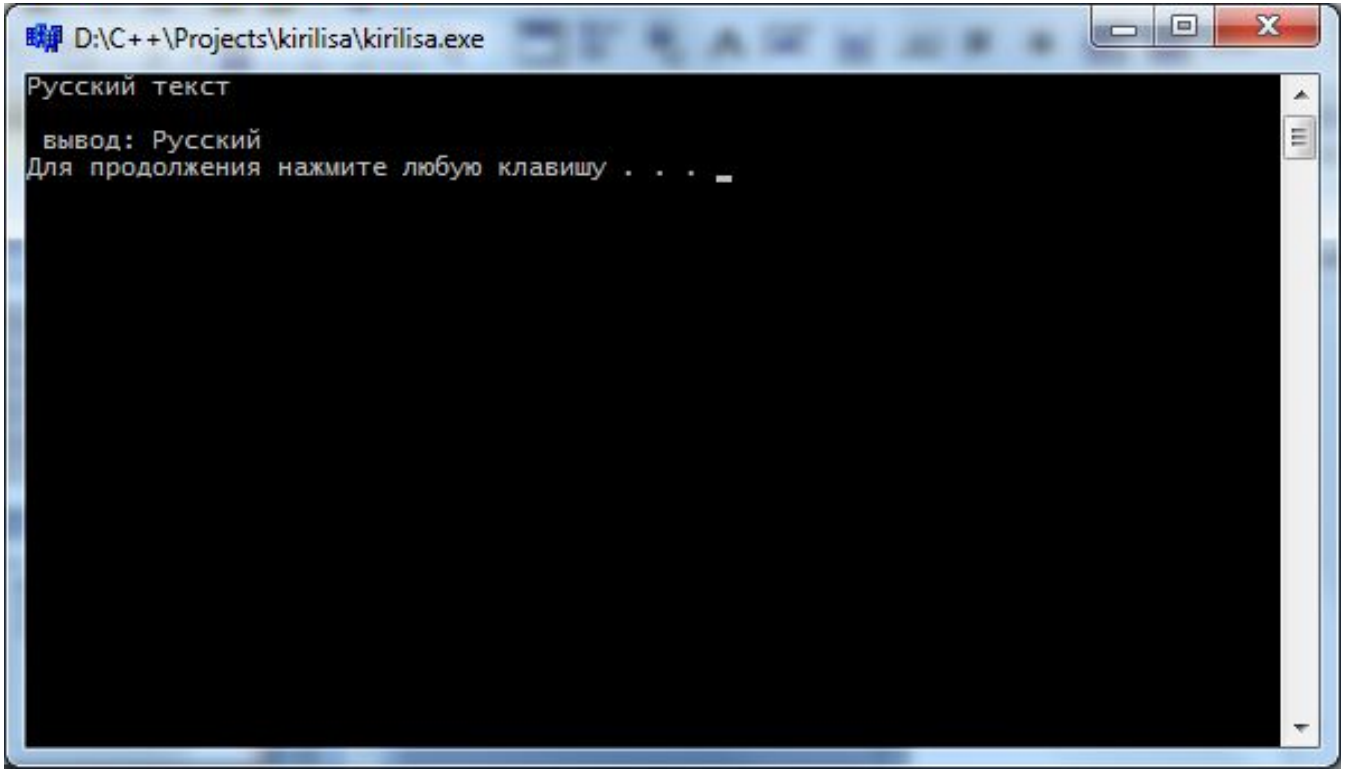


# Файлы проекта

- В каталог с проектом C++ Builder запишет файлы с текстами программ .cpp, заголовочные файлы .h и .hpp, файл установок формы проекта .dfm, файл ресурсов .res и файл проекта .bpr, .bpg или .bpk.
- Файлы ресурсов имеют расширение .res и содержат битовые матрицы (.bmp), пиктограммы (.ico), изображения курсоров (.cur), используемые в проекте.
- Файлы с расширениями ~bpr, ~dfm,~cpp, ~h, obj, .tds и .exe создаются в момент компиляции проекта и могут быть восстановлены после удаления в любое время. Наибольшим объемом обладает файл .tds, предназначенный для отладки программы.

# Компиляция и сохранение

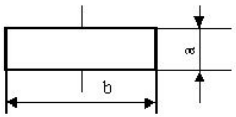
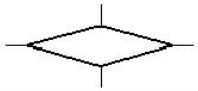
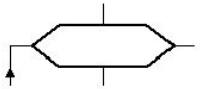
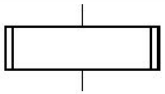
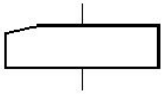
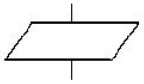
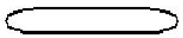
- Компиляция консольного приложения выполняется выбором из меню **Project** команды **Compile**.
- После успешной компиляции программа может быть запущена выбором из меню **Run** команды **Run**.



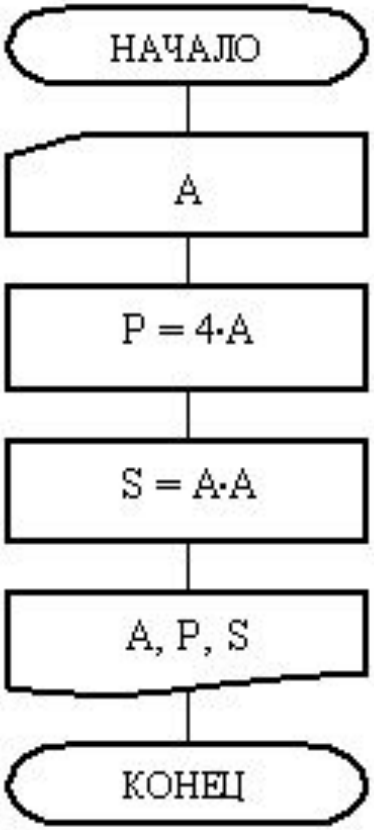
# Алгоритмы

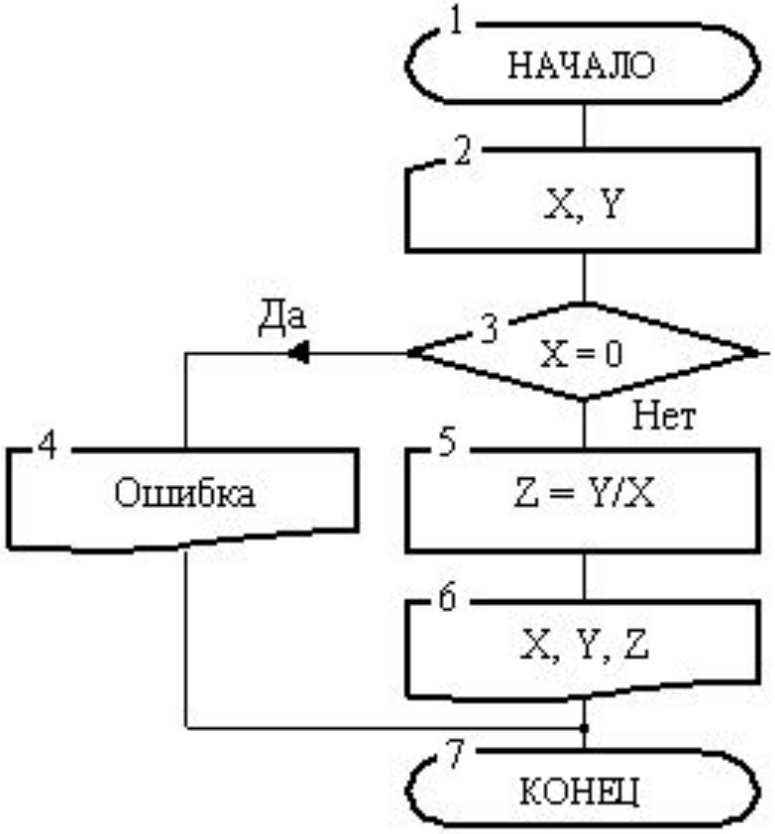
# Алгоритм и алгоритмизация

- ***Алгоритм*** – это инструкция о том, в какой последовательности нужно выполнить действия при переработке исходного материала в требуемый результат.
- ***Алгоритмизация*** – совокупность приемов и способов составления алгоритмов для решения алгоритмических задач.
- Неотъемлемым свойством алгоритма является его *результативность*, то есть алгоритмическая инструкция лишь тогда может быть названа алгоритмом, когда при любом сочетании исходных данных она гарантирует, что через конечное число шагов будет обязательно получен результат.

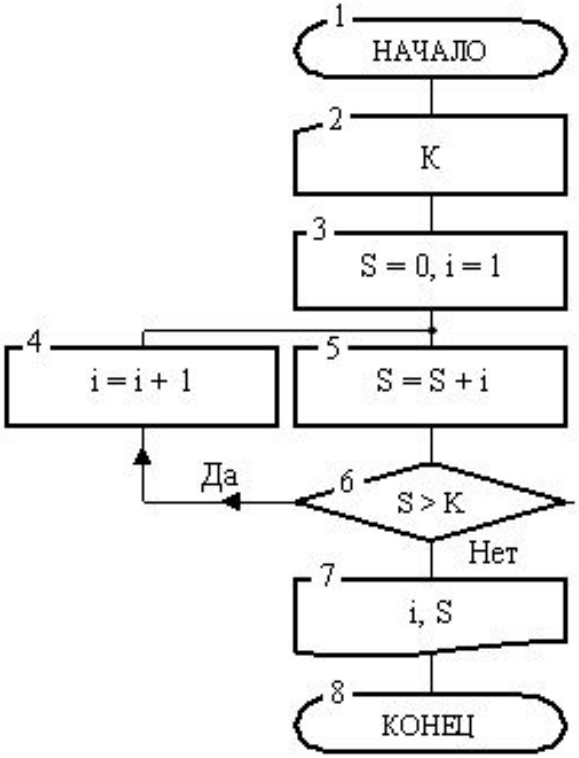
| Название | Элемент | Комментарий |
|---|---|---|
| Процесс |   | Вычислительное действие или последовательность вычислительных действий |
| Решение |  | Проверка условия |
| Модификация |  | Заголовок цикла |
| Предопределенный процесс |  | Обращение к процедуре |
| Документ |  | Вывод данных, печать данных |
| Перфокарта |  | Ввод данных |
| Ввод/Вывод |  | Ввод/Вывод данных |
| Соединитель |  | Разрыв линии потока |
| Начало, Конец |  | Начало, конец, пуск, останов, вход и выход во вспомогательных алгоритмах |
| Комментарий |  | Используется для размещения надписей |
| Горизонтальные и вертикальные потоки |  | Линии связей между блоками, направление потоков |
| Слияние |  | Слияние линий потоков |
| Межстраничный соединитель |  | Нет |

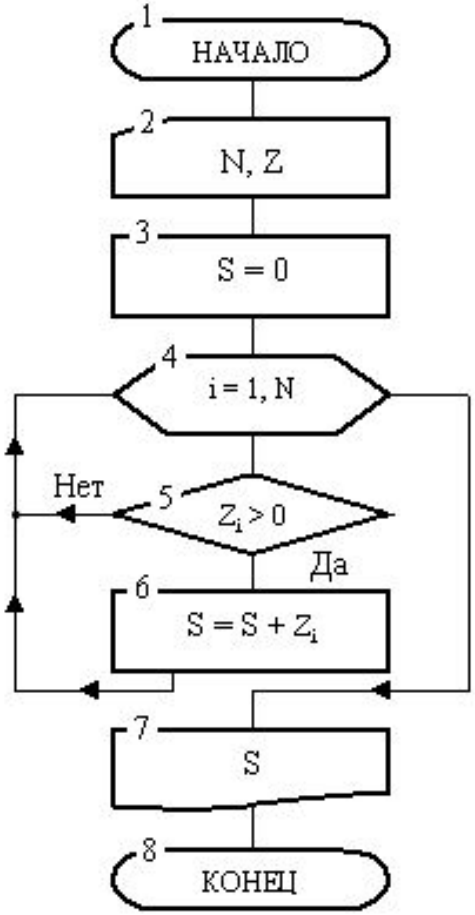
# Линейный и разветвляющийся алгоритмы



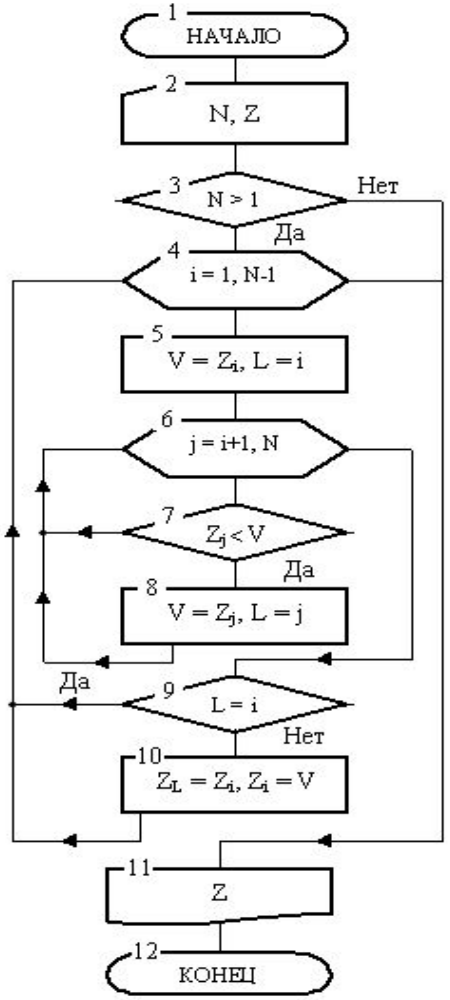



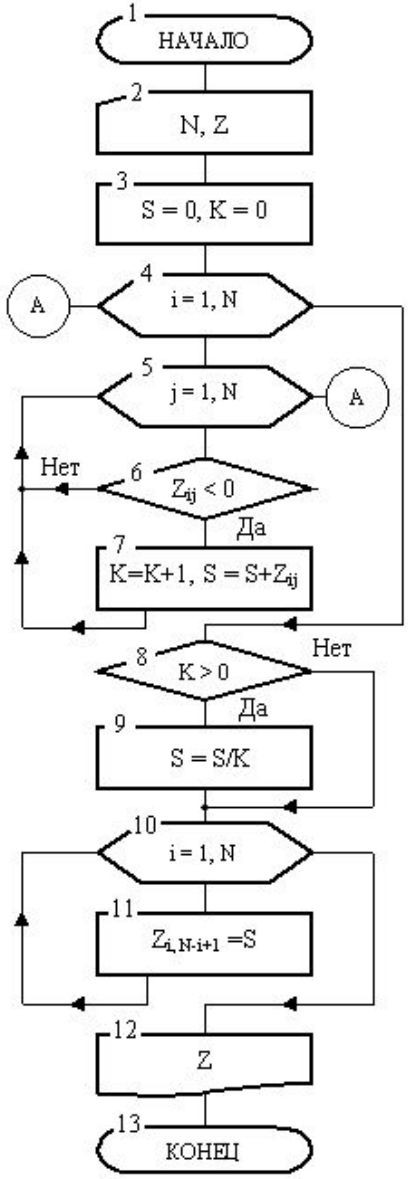
# Циклические алгоритмы



# Алгоритмы со структурами вложенных циклов



# Операции и выражения

# Первичные операции

В соответствии с количеством операндов, которые используются в операциях, они делятся на *унарные* (один операнд), *бинарные* (два операнда) и *тернарную* (три операнда).

| Группа | Обозначение | Название |
|---|---|---|
| первичные | () | круглые скобки |
| | [] | индексация |
| | -> | Обращение к элементу структуры по указателю на структурную переменную |
| | :: | Разрешение видимости |
| | . | Обращение к элементу структуры по имени структурной переменной |

# УНАРНЫЕ ОПЕРАЦИИ

| | | |
|---|---|---|
| унарные | ! | Логическое отрицание |
| | ~ | Побитовое инвертирование |
| | ++ | Автоматический инкремент ++x;<br>Отложенный инкремент x++ |
| | -- | Автоматический декремент --x;<br>Отложенный декремент x-- |
| | - | Обращение знака |
| | + | Подтверждение знака |
| | (тип) | Явное приведение типа |
| | * | Разыменование указателя |
| | & | Взятие адреса |
| | sizeof | Размер в байтах аргумента |
| | new | Операция выделения динамической памяти |
| | delete | Операция освобождения динамической памяти |

| | | |
|---|---|---|
| мультипликатив-ные | * | умножение |
| | / | деление |
| | % | Деление по модулю |
| аддитивные | +<br>- | Сумма<br>разность |
| сдвиги | << | Побитовый сдвиг влево |
| | >> | Побитовый сдвиг вправо |
| отношения | <<br><=<br>><br>>= | Меньше<br>Меньше равно<br>Больше<br>Больше равно |
| Сравнение | == | Сравнение на равенство |
| | != | Сравнение на неравенство |
| поразрядные | & | Побитовое «И» |
| | ^ | Побитовое взаимоисключающее «ИЛИ» |
| | \| | Побитовое «ИЛИ» |

| | | |
|---|---|---|
| логические | && | Логическое «И» |
| | \|\| | логическое «ИЛИ» |
| тернарная | ?: | Условное выражение |
| присваивания | = | Присваивание |
| | *= /= %= += -= &= ^= \|= <<= >>= | Комбинированные присваивания |
| запятая | , | запятая |

# Операции инкремента и декремента

- Эти операции осуществляют увеличение или уменьшение операнда на единицу и имеют две формы записи – *префиксную* и *постфиксную*.
- В *префиксной* форме операция записывается перед операндом. Сначала изменяется операнд, а затем его значение становится результирующим значением выражения.
- В *постфиксной* форме операция записывается после операнда. Значением выражения является исходное значение операнда, после чего он изменяется.

```
#include <iostream.h>
void main ( ) {
int x = 3 , y =3; // определяет и инициализирует переменные целого типа
cout << ”++x = ” << ++x <<’\n’ ; // выводит на экран: ++x = 4
cout << ”y++ = ” << y++ <<’\n’ ; // выводит на экран: y++ = 3
cout << ”x = ” << x <<’\n’ ; // выводит на экран: x = 4
cout << ”y = ” << y <<’\n’ ; // выводит на экран: y = 4 }
```

# Операция определения размера SIZEOF

Операция определения размера *sizeof* предназначена для вычисления размера объекта или типа в байтах, и имеет две формы: *sizeof выражение* и *sizeof ( тип )*

```
#include <iostream.h>
 void main ( ) {
float x = 1; // определяет и инициализирует переменную вещественного типа
cout << ”sizeof (float) = ” << sizeof (float) << ’\n’ ;
// выводит на экран: sizeof (float) = 4
cout << ”sizeof x  = ” << sizeof x <<’\n’ ;
// выводит на экран: sizeof x = 4
cout << ”sizeof (x + 1.0) = ” << sizeof (x + 1.0) <<’\n’ ; // выводит на экран: sizeof (x + 1.0) = 8
}
```

# Операции деление и вычисление остатка от деления

- *Операция деления* применима к операндам арифметического типа. Если оба операнда целочисленные, то результат не содержит дробной части. Если один из операндов вещественный, то тип результата определяется правилами преобразования.
- *Операция остатка от деления* применяется только к целочисленным операндам. Знак результата зависит от реализации.

```
#include <iostream.h>
void main ( ) {
int x = 11 , y = 4 ; // определяет и инициализирует переменные целого типа
float z= 4 ; // определяет и инициализирует переменную вещественного типа
cout << ”x/y = ” << x/y << ’\n’ ; // выводит на экран: x/y = 2 cout << ”x/z = ” << x/z <<’\n’ ; // выводит на экран: x/z = 2.75 cout << ”x%y = ” << x%y <<’\n’ ; // выводит на экран: x%y = 3
}
```

# Оператор присваивания

Для эффективного использования возвращаемого операциями значения предназначен *оператор присваивания* ( **=** ) и его модификации: сложение с присваиванием ( **+=** ), вычисление остатка от деления с присваиванием ( **%=** ), вычитание с присваиванием ( **– =** ), умножение с присваиванием ( ***=** ), деление с присваиванием ( **/=** ).

```
#include <iostream.h>
void main ( ) {
int a , b ; a = b = 36;
a = a – 7 ; b – = 7 ;
a = a / 6 ; b /= 6 ; // a = 4 b = 4
a = a + 7 ; b += 7 ; // a = 11 b = 11
a = a % 4 ; b % = 4 ; // a = 3 b = 3
a = a * 4 ; b * = 4 ; // a = 12 b = 12
}
```

**a += b** является более компактной записью выражения **a = a + b**.

# Операции сравнения

- Для того чтобы имелась возможность сравнивать между собой значения каких-либо переменных, язык C++ предусматривает *операторы сравнения* – бинарные операторы вида:

  *Операнд1 ОператорСравнения Операнд2*

- Операнды могут быть числового типа или указателями. В результате работы операторов сравнения возвращается логическое значение **true** (истина), если проверяемое условие верно, или **false** (ложь) в противном случае.
- Нельзя путать операцию проверки на равенство ( **==** ) и операцию присвоения ( **=**). Синтаксис разрешает использовать операцию присвоения в тех фрагментах программы, где нужно использовать операцию сравнения на равенство. Поэтому компилятор таких ошибок не обнаруживает. Операция проверки на равенство возвращает **true** или **false**, а результатом операции присваивания является значение, присвоенное левому операнду.

# Логические операции

- Операнды логических операций **И** ( **&&** ), **ИЛИ** ( **||** ) **НЕ** ( **!** ) могут иметь логический или числовой тип или быть указателями, при этом операнды в каждой операции могут быть различных типов. Каждый операнд оценивается с точки зрения его эквивалентности нулю (нуль – **false**, не нуль – **true**).
- Результат операции логическое **И** имеет значение **true** только если оба операнда имеют значение **true**.
- Результат операции логическое **ИЛИ** имеет значение **true**, если хотя бы один из операндов имеет значение **true**.
- Результат унарной операции логического отрицания **НЕ** имеет значение **true**, если операнд имеет значение **false**.
- Логические операции выполняются слева направо. Если значения первого операнда достаточно, чтобы определить результат операции, второй операнд не вычисляется.

# Условная операция

- Эта операция тернарная, то есть имеет три операнда. Ее формат:

  **операнд1 ? операнд2 : операнд3**

- Первый операнд может иметь логический или числовой тип или быть указателем. Если результат вычисления операнда1 равен **true**, то результатом условной операции будет значение второго операнда, иначе – третьего операнда. Типы операндов могут различаться.

```
#include <iostream.h>
void main ( ) {
int a = 11 , b = 4 , max ;
max = b > a ? b : a ;
cout << ”max = ” << max <<’\n’ ; }
```

# Операция "запятая"

- Операция "запятая" связывает между собой несколько выражений таким образом, что последние рассматриваются компилятором как единое выражение.
- Благодаря использованию данной операции при написании программ достигается высокая эффек тивность. К примеру, в операторе ветвления *if* можно в качестве выражения ввести:

  $$if\ (\ i = CallFunc\ (\ )\ ,\ i > 7\ )$$

- Еще большей эффективности можно достичь при использовании операции "запятая" в операторе цикла *for*.

# Выражения

- В любой программе требуется производить вычисления. Для вычисления значений используются *выражения*, которые *состоят из операндов*, *знаков операций и скобок*.
- Операнды задают данные для вычислений. Операции задают действия, которые необходимо выполнить.
- Каждый операнд является выражением или одним из его частных случаев, например, константой, переменной, типизованной константой или значением, которое возвращает функция.
- Операции выполняются в соответствии с приоритетами. Для изменения порядка выполнения операций используются круглые скобки. Если в одном выражении записано несколько операций одинакового приоритета, унарные операции, условная операция и операции присваивания выполняются справа налево, остальные – слева направо.

# Преобразования типов

- В выражение могут входить операнды различных типов. Если операнды имеют одинаковый тип, то результат операции будет иметь тот же тип. Если операнды разного типа, перед вычислениями выполняются преобразования типов по определенным правилам, обеспечивающим преобразование более коротких типов в более длинные для сохранения значимости и точности.
- Преобразования бывают двух типов:
  - изменяющие внутреннее представление величин (с потерей точности или без потери точности);
  - изменяющие только интерпретацию внутреннего представления.
- К первому типу относится, например, преобразование целого числа в вещественное (без потери точности) и наоборот (возможно, с потерей точности), ко второму – преобразование знакового целого в беззнаковое.
- Величины типов **char**, **signed char**, **unsigned char**, **short int** и **unsigned short int** преобразуются в тип **int**, если он может представить все значения, или в **unsigned int** в противном случае.
- Программист может задать преобразования типа явным образом.

# Типы данных. Ввод, вывод данных

# Переменная

- **Переменная** – именованная область памяти.
- Каждая переменная перед ее использованием в программе должна быть определена, т. е. для переменной должна быть выделена память.
- Размер участка памяти, выделяемой для переменной, и обработка его содержимого зависят от **типа переменной**, который указывается при ее определении.
- Простейшая форма определения переменных:

  *тип список_имен_переменных;*

  *список_имен_переменных* – идентификаторы переменных, которые разделяются запятыми.
- **Идентификаторы** (имена переменных) могут включать в себя строчные и прописные буквы латинского алфавита, цифры, знак подчеркивания, но начинаться они должны только с буквы или знака подчеркивания. Строчные и прописные буквы в идентификаторах различаются.

| Тип данных | Размер участка памяти, бит | Диапазон значений |
|---|---|---|
| int | 16 | -32768...32767 |
| unsigned int | 16 | 0...65535 |
| long | 32 | -2147483648...2147483647 |
| unsigned long | 32 | 0...4294967295 |
| float | 32 | 3.4e-38...3.4e+38 |
| double | 64 | 1.7e-308...1.7e+308 |
| long double | 80 | 3.4e-4932...1.1e+4932 |
| signed char | 8 | -128...127 |
| unsigned char | 8 | 0...255 |

# Инициализация и преобразование типа переменной

После объявления переменные имеют неопределенные значения. Переменным можно присваивать начальные значения (**инициализировать**) непосредственно при указании их типа:

*тип имя_переменной=начальное_значение;*

```
char symbol_a=198, symbol_b='b';
float pi=3.14;
```

Преобразование типа переменной имеет вид: *(новый_тип) имя_переменной;*

```
int a=3, b=2;
double result1, result2;
result1=a / b;
result2=(double) a / (double) b;
```

# Форматный вывод

*printf(форматная_строка, список_аргументов);*

Функция *printf*() преобразует данные из внутреннего представления в символьный вид в соответствии с форматной строкой и выводит их на экран.

*Форматная_строка* ограничена двойными кавычками и может включать:

- произвольный текст;
- управляющие символы;
- спецификации преобразования данных.

# Управляющие символы

- Управляющие символы используются для вывода на экран кодов, не имеющих графического представления на экране или клавиатуре.
- Для их отображения в форматной строке используются комбинации нескольких символов, имеющих графическое представление.
- Каждая такая комбинация начинается с символа ‘\’ и называется управляющей последовательностью.
- Примеры некоторых управляющих символов:
  - ‘***n**’ перевод строки;
  - ‘***t**’ горизонтальная табуляция;
  - ‘\\’ обратная косая черта;
  - ‘\”’ кавычка.

# Спецификация преобразования

- Для каждого аргумента функции *printf* должна быть указана точно одна спецификация преобразования:

*%флаги_ ширина_поля.точность_ модификатор_спецификатор*

Символ *%* является признаком спецификации преобразования. В спецификации преобразования обязательными являются только два элемента: признак *%* и *спецификатор*.

- Спецификация преобразования не должна содержать внутри себя пробелов. Каждый элемент спецификации является одиночным символом или числом.
- *Спецификатор* определяет как будет интерпретироваться соответствующий аргумент (т. е. тип информации, хранимой в выводимой переменной): как символ, как строка или как число

# Флаги

- *Флаги* управляют выравниванием вывода и печатью знака числа, пробелов, десятичной точки. Флаги могут отсутствовать, а если они и есть, то могут стоять в любом порядке. Смысл некоторых флагов следующий:

  «-» - выводимое изображение прижимается к левому краю поля;

  «+» - если выводимое значение имеет знак (любой), то он выводится. Без этого флага знак выводится только при отрицательном значении;

  « » (пробел) – используется для вставки пробела на месте знака перед положительными числами.

# Ширина_поля

- *Ширина_поля* (положительное целое число) определяет минимальное количество позиций, отводимое для представления выводимого значения.
- Если число символов в выводимом значении меньше, чем указано в ширине поля, то выводимое значение дополняется пробелами до заданной минимальной длины.
- Если ширина поля задана с начальным нулем, то не занятые значащими цифрами выводимого значения позиции слева заполняются нулями.
- Если число символов в выводимом значении больше, чем определено в ширине поля, то печатаются все символы выводимого значения.

# Точность

- *Точность* указывается с помощью точки и необязательного положительного целого числа.
- *Точность* задает:
  - минимальное число цифр, которые могут быть выведены при использовании спецификаторов *d, i, o, u, x;*
  - число цифр, которые будут выведены после десятичной точки при спецификаторах *e* и *f;*
  - максимальное число значащих цифр при спецификаторе *g;*
  - максимальное число символов, которые будут выведены при спецификаторе *s*.

| Спецификатор | Тип аргумента | Формат вывода |
|---|---|---|
| D | int, char, unsigned | Десятичное целое со знаком |
| i | int, char, unsigned | Десятичное целое со знаком |
| u | int, char, unsigned | Десятичное целое без знака |
| o | int, char, unsigned | Восьмеричное целое без знака |
| X | int, char, unsigned | Шестнадцатеричное целое без знака |
| f | float | Вещественное значение со знаком |
| e | float | Вещественное число в экспоненциальной форме |
| g | float | Вещественное число со знаком печатается в формате спецификаторов *«e»* или *«f»* в зависимости от того, какой из них наиболее компактен для данного значения и точности |
| C | int, char | Одиночный символ |
| S | char * | Символьная строка. Символы печатаются либо до первого нулевого символа (‘\0’) (конец строки) либо печатается то количество символов, которое задано в поле *точность* спецификации преобразования |

# Форматный ввод

*scanf(форматная_строка, список_аргументов);*

- Функция *scanf*() читает последовательности кодов символов, поступающих с клавиатуры, и интерпретирует их в соответствии с форматной строкой как целые числа, вещественные, одиночные символы и строки.
- После преобразования во внутреннее представление поступившие данные записываются в области памяти, определенные аргументами, которые следуют за форматной строкой. Поэтому каждый аргумент должен быть указателем на область памяти, соответствующую данной переменной (***&a)***.

- Функция scanf завершает работу, если исчерпана форматная строка. *Форматная_строка* ограничена двойными кавычками и в нее рекомендуется включать только пробельные символы, отслеживающие разделение входного потока на поля;
- Спецификация преобразования имеет вид: % ширина_поля модификатор спецификатор
- Обязательными, как и у функции *printf*(), являются символы *%* и спецификатор, который указывает ожидаемый тип данных при вводе. Спецификаторы, используемые в форматной строке функции *scanf*() те же, что и у функции *printf*(), за исключением спецификатора ***g***.
- Смысл необязательных элементов спецификации преобразования (*ширина_поля* и *модификатор*) тот же, что и у функции *printf*().

```
main()
{   int a;
float b,c;
char symbol, str[10];

scanf("%d",&a);
scanf("%f %f",&b,&c);
scanf("%s",str);
scanf("%c",&symbol);
}
```

# Дополнительные функции для ввода-вывода данных

- Библиотечные функции ввода-вывода данных не ограничиваются двумя, названными выше. Другие из них описаны в файлах stdio.h, io.h, conio.h, iostream.h
- Некоторые, наиболее часто встречающиеся на практике, библиотечные функции ввода-вывода:
  - *getchar*(), *putchar*() – ввод-вывод отдельных символов;
  - *gets*(), *puts*() – ввод-вывод строк;
  - *fprintf*(), *fscanf*() – вывод информации в файл и ввод ее из файла;
  - *sprintf*(), *sscanf*() – вывод информации в строку и ввод ее из строки.
  - cin >>, cout << – ввод, вывод потока данных.

# Операторы

# Условные операторы

***if*** *(выражение_ условие) оператор;*

***if*** (*выражение_условие*) *оператор*_1; ***else*** *оператор*_2;

В выражении_условия могут использоваться арифметические, логические операции и операции отношения.

| |
|---|
| **!** (не), **++** (увеличить на 1), **--** (уменьшить на1) |
| * (умножить), / (разделить), **%** (остаток от деления) |
| **+** (прибавить), **-** (вычесть) |
| **<** (меньше), **<=**(меньше или равно), **>** (больше), **>=** (больше или равно) |
| **= =** (равно), **!=** (не равно) |
| && (логическое и) |
| **\|\|** (логическое или) |

# Условная операция

*(выр1 )?(выр2):(выр3)*

- Вычисляется выражение (выр1). Если это выражение имеет ненулевое значение, то вычисляется выражение (выр2). Результатом операции будет значение выражения (выр2).
- Если значение выражения (выр1) равно нулю, то вычисляется выражение (выр3) и его значение будет результатом операции.

$$max = (x>y) \ ? \ x: \ y \ ;$$

$$abs = (x>0) \ ? \ x: \ -x \ ;$$

# Оператор switch

***switch*** *( выражение )*

**{** ***case*** *константа* 1*: операторы_*1*;*

***case*** *константа* 2*: операторы_*2*;*

*……..*

***default:*** *операторы;*

**}**

```
switch ( X )
  { case 0: printf("ноль"); X++; break;
  case 1: printf("один"); X--; break;
  case 2: printf("два"); X*=2; break;
  default: printf("необрабатываемое значение
  "); break;
   }
```

# Оператор цикла с параметром (for)

***for*** *( выражение 1 ; выражение 2 ; выражение 3 ) тело*

- Выражение 1 обычно используется для установления начального значения переменных, управляющих циклом.
- Выражение 2 - это выражение, определяющее условие, при котором тело цикла будет выполняться.
- Выражение 3 определяет изменение переменных, управляющих циклом после каждого выполнения тела цикла.

# Примеры

```
int i,b;
for (i=1; i<10; i++)
b=i*i;
```

```
int top, bot;
 char string[100], temp;
 for ( top=0, bot=100 ; top < bot ; top++, bot--)
      {
      temp=string[top];
      string[bot]=temp;
      }
```

# Оператор цикла с предусловием (*while*)

*while (выражение) тело*;

- В качестве выражения допускается использовать любое выражение языка Си, а в качестве тела любой оператор, в том числе пустой или составной.
- Схема выполнения оператора *while* следующая:
  - вычисляется выражение
  - если выражение ложно, то выполнение оператора *while* заканчивается и выполняется следующий по порядку оператор, если выражение истинно, то выполняется тело оператора *while*.

# ОПЕРАТОРЫ ЦИКЛА С ПОСТУСЛОВИЕМ

*do тело while (выражение);*

Схема выполнения оператора *do while* :

- выполняется тело цикла (которое может быть составным оператором).
- вычисляется выражение.
- если выражение ложно, то выполнение оператора *do while* заканчивается и выполняется следующий по порядку оператор.

Операторы *while* и *do while* могут быть вложенными.

```
 int i,j,k;
 ...
i=0; j=0; k=0;
do {
 i++;
 j--;
 while (a[k] < i)
k++;
}
while (i<30 && j<-30);
```

# Массивы

# Определение массива

- Конечная именованная последовательность однотипных величин называется массивом.
- Отдельная единица таких данных, входящих в массив, называется *элементом массива*. В качестве элементов массива могут выступать данные любого типа, а также указатели на однотипные данные. Мас сивы бывают *одномерными* и *многомерными*.
- Синтаксис определения массива имеет вид

  *тип_элемента имя_массива [n1][n2]...[nk];*

где имя_массива - идентификатор, определяемый в качестве имени массива, а $n_i$ - размеры массива.

- Тип элемента массива может быть одним из основных типов, типом указателя (pointer), типом структуры (struct) или типом объединения (union). Хотя элементы массива не могут быть функциями, они могут быть указателями на функции.

Примеры определений массива:

*int page[10];*

*char line[81*

*float big[10][10], sales[10][5][8];*

# Инициализация одномерного массива

- Инициализацию массивов, содержащих элементы базовых типов, можно производить при их объявлении. При этом непосредственно после объявления необходимо за знаком равенства ( = ) перечислить значения элементов в фигурных скобках через запятую ( , ) по порядку их следования в массиве.
- Если элементов в массиве больше, чем инициализаторов, элементы, для которых значения не указаны, обнуляются и имеет место частичная инициализация. В этом случае иногда после последнего значения в инициализирующем выражении для наглядности ставят запятую:

```
int b [ 5 ] = { 3 , 2 , 1 , } ;
```

# Обращение к элементам массива с помощью индекса

- Обращение к элементам массива может осуществляться одним из двух способов:
  - по номеру элемента в массиве через индекс;
  - по указателю.
- Для доступа к элементу массива после его имени указывается номер элемента в квадратных скобках. Элементы массива нумеруются, начиная с 0.

```
#include <iostream.h>
void main ( ) {
const int n = 10 ;
int m [ n ] = { 3 , 4 , 5 , 4 , 4 } ;
for ( int i = 0 , sum = 0 ; i < n ; i++ ) sum += m [ i ] ;
cout << ”Summa of elements: \t” << sum ; }
```

# Переменные типа указатель

Существуют переменные типа указатель. Значением переменной типа char является целое число длиной 1 байт, а значением переменной типа указатель служит адрес переменной.

Операция косвенной адресации * позволяет обратиться к переменной через указатель, содержащий адрес этой переменной.

Пусть ptr – указатель, тогда * ptr – это значение переменной, на которую указывает ptr .

# Описание указателей

<тип> *<имя указателя на переменную заданного типа>;

int *ptri - указатель на переменную целого типа;

char *ptrc - на переменную символьного типа;

float *ptrf - на переменную с плавающей точкой.

# Указатели и одномерные массивы

Пусть mas[6] – массив из 6 элементов, тогда mas и &mas[0] эквивалентны и определяют адрес первого элемента массива.

Оба значения являются константами типа указатель, поскольку они не изменяются на протяжении работы программы. Эти значения можно присваивать переменным типа указатель.

```
int  a[4], *ptra, i ;
float b[4], *ptrb ;
ptra=a; ptrb=b; //присваивают указателям адреса массивов
```

# Указатели и одномерные массивы

```
for (i=0; i<4; i++)
    cout<< ”указатель + ”<<i<< ” : ”<<(ptra+i)
          << ”  ”<<(ptrb+i)<<” \n ”;
```

Результат может быть таким:

указатель+0 : 0x2e2112b2  0x2e2112ee

указатель+1 : 0x2e2112b4  0x2e2112f2

указатель+2 : 0x2e2112b6  0x2e2112f6

указатель+3 : 0x2e2112b8  0x2e2112fa

Тип int занимает 2 байта, тип float – 4 байта.

# Указатели и многомерные массивы

```
int mas [4] [2];
int *ptr;
```

Тогда выражение ptr=mas указывает на первый столбец первой строки матрицы, т.е. записи mas и mas[0][0] равносильны.

Выражение ptr+1 указывает на элемент mas[0][1], далее идут элементы: mas[1][0], mas[1][1] и т.д.; ptr+5 указывает на mas[2][1] .

# Динамические массивы

- Динамическим называется массив, размерность которого становится известной в процессе выполнения программы.
- С помощью операции new выделяется память под динамический массив, а с помощью операции delete – освобождается.

```
int  n;
cin>>n;  // размерность массива
int *mas=new int [n]; // выделение памяти
delete mas; // освобождение памяти
```

# Выделение памяти под многомерные массивы

Требуется создать двумерный динамический массив целых чисел размерностью n*k.

```
int  n, k, i;
cin >> n; cin >> k;
int  **mas=new  int *[n]; // выделение памяти под  n указателей на строку
for (i=0; i<n; i++) mas[i]=new int[k]; // выделение памяти для каждой строки по числу столбцов k
for (i=0; i<n; i++) delete mas[i];
delete [ ] mas; // освобождение памяти
```

# Обращение к элементам массива с помощью указателя

Имя объявляемого массива ассоциируется компилятором с адресом его самого первого элемента с индексом 0. Таким образом можно присвоить указателю адрес нулевого элемента, используя имя массива:

**char A [ ] = { ’w’ , ’o ’ , ’r ’ , ’l’ , ’d ’ } ;** *// объявляет и инициализирует массив символов A*

**char* pA = A ;** *// pA указывает на A [ 0 ]*

- Разыменовывая указатель **pA**, можно получить доступ к содержимому **A [ 0 ]**:

**char Letter = *pA ;** *// объявляет и инициализирует символьную переменную*

**cout << Letter << ’\n’ ;** *// выводит на экран w*

# Операция разадресации (разыменования)

Операция разадресации, или разыменования, предназначена для доступа к величине, адрес которой хранится в указателе.

**char A;** *// определяет переменную типа char*

**char * pA = new char ;** *// выделяет память под указатель и под динамическую переменную типа char*

***pA = 'U' ;** *// передаёт значение в динамическую переменную*

**A = *pA ;** *// считывает значение из динамической переменной и передаёт его в переменную A*

**delete pA ;** *// удаляет динамическую переменную из оперативной памяти*

# Модификация указателя

- Увеличивая или уменьшая значение указателя на массив, программист получает возможность доступа ко всем элементам массива путем соответствующей модификации указателя:

  **pA+= 2 ;** *// увеличивает адрес на 2 байта*

  **cout << *pA << '\n' ;** *// выводит на экран r*

- К этому же элементу можно обратиться иным способом:

  **Letter = *( A + 2 ) ;** *// эквивалент Letter = A [ 2 ] ;*

- **char ( *pA ) [ 10 ] ;** определяет указатель **pA** на массив из 10 символов.
- Если же опустить круглые скобки, то компилятор поймет запись **char *pA [ 10 ]** как объявление массива из 10 указателей на тип **char**.

# ССЫЛКИ

Ссылка представляет собой псевдоним объекта, указанного при инициализации ссылки. Ссылку можно рассматривать как указатель, который всегда разыменовывается. Формат объявления ссылки:

*тип & имя_ссылки = имя_переменной ;*

где тип – это тип величины, на которую указывает ссылка, & – оператор ссылки, означающий, что следующее за ним имя является именем переменной ссылочного типа.

**int V= 0 ;** *// определяет переменную типа int*

**int & ref = V ;** *// определяет ссылку на переменную типа int и инициализирует её*

**ref += 10 ;** *// то же, что и V += 10*

**const double & pi = 3.1415 ;** *// определяет ссылку на константу*

# Ссылки

Для ссылок определены следующие правила:

- переменная-ссылка должна явно инициализироваться при ее описании, кроме случаев, когда она является параметром функции, описана как extern или ссылается на поле данных класса;
- после инициализации ссылке не может быть присвоена другая переменная;
- тип ссылки должен совпадать с типом величины, на которую она ссылается;
- не разрешается определять указатели на ссылки, создавать массивы ссылок и ссылки на ссылки;
- нельзя объявить ссылку на тип объекта.

# Операции со строками

# МАССИВЫ СИМВОЛОВ В C++

- В стандарт C++ включена поддержка нескольких наборов символов. Традиционный 8-битовый набор символов называется "узкими" символами. Кроме того, включена поддержка 16-битовых символов, которые называются "широкими". Для каждого из этих наборов символов в библиотеке имеется своя совокупность функций.
- Как и в ANSI C, для представления символьных строк в C++ не существует специального строкового типа. Вместо этого строки в C++ представляются как массивы элементов типа **char**, заканчивающиеся *терминатором строки* – символом с нулевым значением **'\0'**.
- Символьные строки состоят из набора символьных констант, заключенного в двойные кавычки:

**"Это строка символов..."**

# НАБОР КОНСТАНТ, ПРИМЕНЯЮЩИХСЯ В C++ В КАЧЕСТВЕ СИМВОЛОВ

| | | | |
|---|---|---|---|
| прописная буква | от 'A' до 'Z' , от 'А' до 'Я' | | |
| строчная буква | от 'a' до 'z' , от 'а' до 'я' | | |
| цифра | от '0' до '9' | | |
| пустое место | '\t' | – горизонтальная табуляция | код ASCII – 9, |
| | '\n' | – перевод строки | код ASCII – 10, |
| | '\v' | – вертикальная табуляция | код ASCII – 11, |
| символы пунктуации | ! " # $ % & ' ( ) * + , – . / : ; < = > ? @ [ \ ] ^ _ { \| } ~ | | |
| управляющий символ | все символы с кодами от 0 до 1F и символ с кодам 7F | | |

# Терминатор строки

- При объявлении строкового массива необходимо принимать во внимание наличие терминатора в конце строки, отводя тем самым под строку на один байт больше:
- **char buffer [10] ;** *// объявление строки размером 10 символов, включая терминатор. Реальный размер строки: 9 символов + терминатор.*
- Строковый массив может при объявлении инициализироваться начальным значением. При этом компилятор автоматически вычисляет размер будущей строки и добавляет в конец нуль-терминатор:

**char Wednesday [ ] = ”Среда”;**

**char Wednesday [ ] = {’С’, ’р’, ’е’, ’д’, ’а’, ’\0’} ;**

# Ввод строк

- В качестве оператора ввода при работе со строками вместо оператора записи в поток **>>** лучше использовать функцию **getline ( )**, так как потоковый оператор ввода игнорирует вводимые пробелы и может продолжить ввод элементов за пределами массива, если под строку отводится меньше места, чем вводится символов. Синтаксис функции **getline ( )** имеет вид:

**istream& getline ( char* pch , int nCount , char delim = '\n' ) ;**

- Функция **getline ( )** принимает два обязательных параметра: первый аргумент **pch** указывает на строку, в которую осуществляется ввод, а второй параметр **nCount** – число символов, подлежащих вводу. Третий необязательный параметр **delim** – символ, который будет преобразован в нуль-терминатор. По умолчанию это символ конца строки **'\n'**.

**getline ( ) char S [ 6 ] ;** *// объявляет и инициализирует строку длиной в 5 символов*

**cout << "Введите строку:" ;** *// выводит на экран приглашение*

**cin.getline ( S , 6 , '.' ) ;** *// ввод строки длиной не более 5 символов, завершается точкой*

**cout <<"Ваша строка: "<< S <<'\n' ;** *// выводит строку на экран*

# ОПРЕДЕЛЕНИЕ ДЛИНЫ СТРОК

- Для определения длины строки в заголовочном файле **string.h** описана функция **strlen ( )**. Синтаксис этой функции имеет вид:

**size_t strlen ( const char* string ) ;**

- Данная функция в качестве единственного параметра принимает указатель на начало строки **string**, вычисляет количество символов строки и возвращает полученное беззнаковое целое число типа **size_t**.
- Функция **strlen ( )** возвращает значение на единицу меньше, чем отводится под массив по причине резер вирования места для символа **'\0'**.

```
char S [ ] = "0123456789" ;
cout << "Lenght=" << strlen ( S ) << '\n' ; // 11
cout << "Size =" << sizeof ( S ) << '\n' ; // 10
```

- Часто функция **sizeof** используется при вводе строк в качестве второго параметра конструкции **cin.getline ( )**, что делает код более универсальным.
- Если теперь потребуется изменить размер символьного массива, достаточно модифицировать лишь одно число при объявлении строки символов:

**char S [20] ;** *// объявляет строку длиной 19 символов*

**cin.getline ( S , sizeof ( S ) ) ;** *// ввод строки длиной не более 19 символов с клавиатуры*

| Функция | Прототип и краткое описание действий |
|---|---|
| atof | double atof (char *str). Преобразует строку str в вещественное число типа double. |
| atoi | int atoi (char *str). Преобразует строку str в десятичное целое число. |
| atol | long atol (char *str). Преобразует строку str в длинное десятичное целое число. |
| itoa | char *itoa (int v, char *str, int baz). Преобразует целое v в строку str. При изображении числа используется основание baz (2 < baz < 36). Для отрицательного числа и baz = 10 первый символ – "минус" (-). |
| ltoa | char *ltoa (long v, char *str, int baz). Преобразует длинное целое v в строку str. При изображении числа используется основание baz (2 <= baz <= 36). |
| strcat | char *strcat (char *sp, char *si). Приписывает строку si к строке sp (конкатенация строк). |
| strchr | char *strchr (char *str, int c). Ищет в строке str первое вхождение символа c. |
| strcmp | int strcmp (char *strl, char *str2). Сравнивает строки strl и str2. Результат отрицателен, если strl < str2; равен нулю, если strl == str2 и положителен, если strl > str2 (сравнение без знаковое). |

| | |
|---|---|
| strcpy | char *strcpy (char *sp, char *si). Копирует байты строки si в строку sp. |
| strcspn | int strcspn (char *strl, char *str2). Определяет длину первого сегмента строки strl, содержащего символы, не входящие во множество символов строки str2. |
| strdup | char *strdup (const char *str). Выделяет память и переносит в нее копию строки str. |
| strlen | unsigned strlen (char *str). Вычисляет длину строки str. |
| strlwr | char *strlwr (char *str). Преобразует буквы верхнего регистра в строке в соответствующие буквы нижнего регистра. |
| strncat | char *strncat (char *sp, char *si, int kol). Приписывает kol символов строки si к строке sp (конкатенация). |
| strncmp | int strncmp (char *str1, char *str2, int kol). Сравнивает части строк strl и str2, причем рассматриваются первые kol символов. Результат отрицателен, если strl < str2; равен нулю, если strl == str2 и положителен, если strl > str2. |
| strncpy | char *strncpy (char *sp, char *si, int kol). Копирует kol символов строки si в строку sp ("хвост" отбрасывается или дополняется пробелами). |

| | |
|---|---|
| strnset | char *strnset (char *str, int c, int kol). Заменяет первые kol символов строки str символом c. |
| strpbrk | char *strpbrk (char *strl, char *str2). Ищет в строке strl первое появление любого из множества символов, входящих в строку str2. |
| strrchr | char *strrchr (char *"tr, int c). Ищет в строке str последнее вхождение символа c. |
| strset | int strset (char *str, int c). Заполняет строку str заданным символом c. |
| strspn | int strspn (char *strl, char *str2). Определяет длину первого сегмента строки strl, содержащего только символы, из множества символов строки str2. |
| strstr | char *strstr (const char *strl, const char *str2). Ищет в строке strl подстроки str2. Возвращает указатель на тот элемент в строке strl, с которого начинается подстрока str2. |

# Функции

# МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ

- С увеличением объема программы становится невозможным удерживать в памяти все детали. Естественным способом борьбы со сложностью любой задачи является ее разбиение на части.
- В каждой программе на C++ должна присутствовать функция main(), которая получает управление при запуске программы. Все остальные функции, необходимые для решения задачи, вызываются из main().
- В C++ задача может быть разделена на более простые с помощью *функций,* после чего программу можно рассматривать в более укрупненном виде – на уровне взаимодействия функций. Использование функций является первым шагом к повышению степени абстракции программы и ведет к упрощению ее структуры.
- Разделение программы на функции позволяет также избежать избыточности кода, поскольку функцию записывают один раз, а вызывать ее на выполнение можно многократно из разных точек программы.
- Часто используемые функции можно помещать в библиотеки. Таким образом создаются более простые в отладке и сопровождении программы.

# Объявление и определение функций

- *Функция – это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие*. Функция может принимать параметры и возвращать значение.
- Любая программа на C++ состоит из функций, одна из которых должна иметь имя **main ( )**, с которой начинается выполнение программы.
- Функция начинает выполняться в момент *вызова*.
- Любая функция должна быть *объявлена* и *определена*.

# Объявление и определение функции

***тип имя ( [ список параметров ] ) { тело функции }***

- *Объявление функции (прототип, заголовок, сигнатура)* задает ее имя, тип возвращаемого значения и список передаваемых параметров.

  **int sum(int a, int b); //прототип**

- *Определение функции* содержит *заголовок* и *тело* функции.
- *Тело* функции представляет собой последовательность операторов и описаний в фигурных скобках:

  **int sum(int a, int b) { return (a + b);}**

- *Тип* возвращаемого функцией значения может быть любым, кроме массива и функции (но может быть указателем на массив или функцию). Если функция не должна возвращать значение, указывается тип **void**.
- *Список параметров* определяет величины, которые требуется передать в функцию при ее вызове. Элементы списка параметров разделяются запятыми. Для каждого параметра, передаваемого в функцию, указывается его тип и имя (в объявлении имена можно опускать).

# Вызов функции

- Для вызова функции в простейшем случае нужно указать ее имя, за которым в круглых скобках через запятую перечисляются имена передаваемых аргументов:

  **имя ( аргумент1 , аргумент2 , . . . , аргументN ) ;**

- Вызов функции может находиться в любом месте программы, где по синтаксису допустимо выражение того типа, который формирует функция.
- Если тип возвращаемого функцией значения не **void**, то вызов функции может входить в состав выражений или располагаться в правой части оператора присваивания.
- Каждый аргумент функции представляет собой переменную, выражение или константу, передаваемые в тело функции для дальнейшего использования в вычислительном процессе. Список аргументов функции может быть пустым.
- Функция может вызывать другие функции (одну или несколько), а те, в свою очередь, производить вызов третьих и т.д. Кроме того, функция может вызывать сама себя. Этот приём в программировании называется *рекурсией*.

# Пример вызова функции

```
int sum(int a, int b){ return (a + b);}
…
int a = 2, b = 3, c, d;
c = sum(a, b);
cin >> d;
cout << sum(c, d);
```