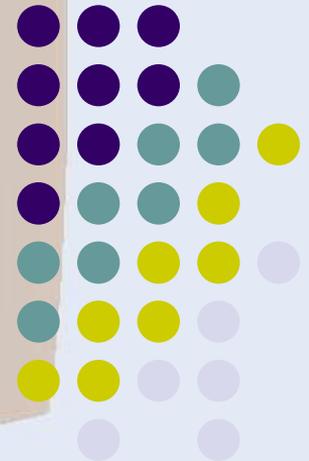


# Лекция 8

Язык программирования  
C/C++  
Основные сведения



# Язык C

- ❑ Язык C был создан в 1972 году на основе предыдущего языка В как инструментальное средство для реализации операционной системы Unix.
- ❑ В отличие от изначально нацеленного на учебные цели языка Паскаль, язык C был разработан как инструмент для программистов-практиков.
- ❑ Однако теперь популярность этого языка быстро переросла рамки конкретной операционной системы и конкретных задач системного программирования.
- ❑ До недавнего времени языки C и C++ были практически единственным эффективным средством профессионального системного программирования.
- ❑ Язык C, как и классический вариант языка Pascal, представляет собой процедурный язык.
- ❑ Современные unix-подобные операционные системы до сих пор пишутся на языке C.

# C: Немного истории

- ❑ 1970 г., Кен Томпсон, Денис Ритчи, AT&T Bell Laboratories
- ❑ UNIX PDP-11: A (ассемблер) – B – C
- ❑ Первый продукт – компилятор
- ❑ 1976 г. – перенос Unix на Interdata 8/32 (затем и на другие платформы)
- ❑ 1978 г. – первая книга
- ❑ 1987 г. – стандарт ANSI C
- ❑ UNIX, OS IBM, MS DOS, Mac OS, ...
- ❑ 1999 г. – текущий стандарт ISO 9899:1999

# C++: Продолжение C

- ❑ 1980 г., Бьёрн Страуструп, AT&T Bell Laboratories
  - C++ почти включает C (но есть и несовместимости)
  - C++ расширяет C за счет ООП и строгой проверки типов
- ❑ Borland C++, Visual C++, Borland C++ Builder, Microsoft Visual Studio C++ Compiler, Intel C++ Compiler, GNU C++ Compiler...
- ❑ 1998 г. – стандарт ISO 14882:1998
- ❑ 2003 г. – стандарт ISO 14882:2003
- ❑ 2005 г. – стандарт C++/CLI
- ❑ 2011 г. – текущий стандарт C++11

# Язык C++

- ❑ Язык C++ является самостоятельным языком программирования, разработанным на основе C.
- ❑ Особенностью языка C++ является включение в язык объектно-ориентированного подхода и введение дополнительных упрощающих синтаксических конструкций.
- ❑ Язык C++ позволяет пользоваться всеми средствами языка C, однако не следует воспринимать его как набор дополнений к C. C++ – самостоятельный язык и его отличие от классического C намного больше, даже чем отличие Турбо Паскаля от классического Паскаля.
- ❑ Важной особенностью языка C++ является его стандартизированность. На язык C++ существует ANSI стандарт, позволяющий без проблем использовать и компилировать программы, написанные на этом языке при помощи разных компиляторов.

# Общая характеристика C

- ❑ C – элементы языка низкого уровня
  - компьютерные (аппаратные) типы данных
  - логические операции над битами, сдвиги, работа с адресами и регистрами
- ❑ C – язык высокого уровня
  - структуры данных и операторы структурного программирования
  - необычно большой набор операций
  - указатели и функции
- ❑ Простой компилятор
  - эффективность
  - экономичность
  - переносимость

# Общая характеристика C

- ❑ Препроцессор – преобразование исходного текста программы до компиляции
- ❑ Развитые библиотеки
  - богатая стандартная библиотека
  - большое число разнообразнейших библиотек
- ❑ Существенные недостатки
  - трудность чтения низкоуровневого кода:  
`a++ << --b | 4`
  - «слабый» синтаксис
    - ✓ `'a' + 28`
    - ✓ контроль параметров функций

# Преимущества C++

1. Стандартизированность.
  2. Эффективность генерируемого кода.
  3. Универсальность. Возможность гибкой реализации требуемых алгоритмов.
  4. Удобство. Достаточная строгость и структурированность без излишних формальных рамок.
- Иными словами – язык C++ - это язык, идеально подходящий для системного программирования или написания приложений, требующих оптимального и эффективного выполнения.

# Недостатки C++

1. По сравнению с Паскалем, языки C и C++ являются языками более низкого уровня, т.е. более приближенными к системе. C++ - язык более понятный для компьютера, чем для человека.
  2. Синтаксические конструкции языка C++ удобны для компиляции и краткости написания, однако не очень удобны для чтения и проектирования программ.
  3. Отсутствие жестких структурных рамок может привести к достаточно неупорядоченной программе.
- ❑ Одним словом, язык C++ не очень удобен как учебный язык или язык для написания прикладных программ.
  - ❑ Все доводы за и против языков C и C++ являются достаточно субъективными.

# Литература

- **Страуструп Б. Язык программирования Си++. Специальное издание.** Пер. с англ. — М.: ООО «Бином-Пресс», 2004. — 1104 с.
- **Липпман С. Основы программирования на С++. Вводный курс.** Пер. с англ. — М.: Вильямс, 2002.  
<http://anatolix.naumen.ru/Books/EssentialCPP?v=yuq>
- **Липпман С. С++ для начинающих.**  
[http://anatolix.naumen.ru/files/books/lippman\\_cpp\\_primer\\_rus.zip](http://anatolix.naumen.ru/files/books/lippman_cpp_primer_rus.zip)
- **Лафоре Р. Объектно-ориентированное программирование в С++.** Пер. с англ. — СПб.: Питер, 2003. — 928 с.
- **Шилдт Г. Полный справочник по С++. Пер. с англ. — М.: Вильямс, 2004. — 800 с.**
- **М. Уэйт, С. Прата, Д. Мартин. Язык Си: руководство для начинающих.** М., "Мир", 1988.  
<http://www.jinr.ru/~dushanov/book/c/index.html>
- **Подбельский В. В. Язык Си++: Учебное пособие. — 5-е изд., дораб. — М.: Финансы и статистика, 2006. — 560 с.**  
<http://www.jinr.ru/~dushanov/book/cpp/>

# Еще литература

- Керниган Б., Ричи Д. **Язык С**. <http://infocity.kiev.ua/m.php?f=0&id=107>
- Кетков Ю.Л., Кетков А.Ю., **Практика программирования: Бейсик, Си, Паскаль. Самоучитель.** – СПб.: БХВ-Петербург, 2002. – 480 с.
- Лаптев В. **С++. Экспресс-курс.** – СПб.: БХВ – Петербург, 2004 – 512с.
- Ю.Ю.Громов, С.И.Татаренко. **Программирование на языке СИ: Учебное пособие.** -Тамбов,1995.- 169 с.  
<http://www.helloworld.ru/texts/comp/lang/c/c/dir.htm>
- А. Богатырев. **Руководство полного идиота по программированию (на языке Си).** <http://www.helloworld.ru/texts/comp/lang/c/c6/>
- Аллен И. Голуб. **Веревка достаточной длины, чтобы выстрелить себе в ногу. Правила программирования на С и С++.**  
<http://www.helloworld.ru/texts/comp/lang/c/c11/cpprules.zip/INTRODUC.DOC>
- Стефан К.Дьюхерст. **Скользкие места С++. Как избежать проблем при проектировании и компиляции ваших программ.** – М.:ДМК Пресс, 2006. – 264с.
- Мейерс С. **Эффективное использование С++. 55 верных советов улучшить структуру и код ваших программ.** – М. ДМК Пресс, 2006.  
[http://wmate.ru/ebooks/dl\\_book250.html?mirror=1&SID](http://wmate.ru/ebooks/dl_book250.html?mirror=1&SID)

# Сайты:

- ❑ Сайт алгоритмов: <http://algotlist.manual.ru/>
- ❑ Сайт литературы по C++:  
<http://anatolix.naumen.ru/Books/cplusplus>
- ❑ E-books: Книги по C, C++  
<http://wmate.ru/ebooks/cat5/> E-books: Книги по C, C++ <http://wmate.ru/ebooks/cat5/> (62 книги список [здесь](#))

# Компиляторы C++

- ❑ **Borland C++** и **Borland C++ Builder** – Компилятор и среда визуального программирования фирмы Borland.
- ❑ **Microsoft Visual C++** - среда визуального программирования на основе C++. Имеет все недостатки продукции фирмы Microsoft.
- ❑ Компилятор **gcc** – свободно распространяемый компилятор для Unix, имеет версию для Windows – **mingw**.
- ❑ На основе компилятора **mingw** создана оболочка программирования **Dev-C++** и среда визуального программирования **wxDev-C++**.
- ❑ **Eclipse 3.5 (Galileo)** – CDT (C/C++ Development Tools) – среда разработки на C/C++ (C/C++ IDE)
- ❑ **Intel® Parallel Studio XE 2013, Intel® C++ Studio XE 2013**

# Алфавит языка C/C++

□ Алфавит языка C/C++ включает

- прописные и строчные латинские буквы и знак подчеркивания;
- арабские цифры от 0 до 9;
- специальные знаки “{ }, | [ ] ( ) + - / % \* . \ ' : ; & ? < > = ! # ^
- пробельные символы (пробел, символ табуляции, символы перехода на новую строку).

□ Комментарий к тексту программы помещается между символами /\* и \*/ , что исключает сам текст комментария из транслируемой программы. Кроме этого, комментарий может помещаться после символов // , исключая весь последующий до конца строки текст.

□ При формировании текстовых строк и для записи комментария к программам допускается использовать

# Лексемы языка

□ Из символов формируются лексемы языка:

- *Идентификаторы* – имена объектов С-программ.

В идентификаторе могут быть использованы латинские буквы, цифры и знак подчеркивания.

Прописные и строчные буквы различаются, например, **PROG1**, **prog1** и **Prog1** – три различных идентификатора. Первым символом должна быть буква или знак подчеркивания (но не цифра).

Пробелы в идентификаторах не допускаются.

Идентификатор может содержать до 32 символов

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

# ЛексемЫ языка

- *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.
- *Константы* – это неизменяемые величины. Существуют целые, вещественные, символьные и строковые константы. Компилятор выделяет константу в качестве лексемЫ (элементарной конструкции) и относит ее к одному из типов по ее внешнему виду.
- *Разделители* – скобки, точка, запятая, пробельные СИМВОЛЫ.

# Структура программы в C/C++

**#директивы препроцессора**

.....

**#директивы препроцессора**

**функция а ( )**

операторы

**функция в ( )**

операторы

**void main ( )** //функция, с которой начинается выполнение программы

**операторы**

**описания**

**присваивания**

**функция**

**пустой оператор**

**составной**

**выбора**

**циклов**

**перехода**

# Особенности компиляции в 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 (редактирование связей или сборка).



# Директивы препроцессора C/C++

- Задача препроцессора – преобразование текста программы до ее компиляции. Правила препроцессорной обработки определяет программист с помощью директив препроцессора.
- После выполнения препроцессорной обработки в тексте программы не остается ни одной препроцессорной директивы.
- Директива начинается с #. Например:
  - Директива **#define** – указывает правила замены в тексте.
  - Пример:

```
#define ZERO 0.0
```
  - Означает , что каждое использование в программе имени ZERO будет заменяться на 0.0.

# Директивы препроцессора C/C++

Пример использования директивы **#define**:

```
#define N 20
```

```
int a[N];
```

```
int x;
```

```
void func (void)
```

```
{
```

```
int i;
```

```
for (i = 0; i < N; i++)
```

```
    x += a[i];
```

```
}
```

```
int a[20];
```

```
int x;
```

```
void func (void)
```

```
{
```

```
int i;
```

```
for (i = 0; i < 20;
```

```
    i++)
```

```
    x += a[i];
```

```
}
```



# Директивы препроцессора C/C++

Пользоваться директивой `#define` нужно аккуратно:

```
#define N 20
```

```
int a[N];
```

```
...
```

```
void func (void)
```

```
{
```

```
int N;
```

```
N++;
```

```
}
```

```
int a[20];
```

```
...
```

```
void func (void)
```

```
{
```

```
int 20;
```

```
20++;
```

```
}
```



**Ошибка при  
компиляции!**

# Директивы препроцессора C/C++

Более сложный пример использования директивы **#define**:

```
#define SQUARE(val) val * val  
void func (int x, int y)  
{  
    int a, b;  
    a = SQUARE (x);  
    b = SQUARE (y);  
}
```



```
void func (int x, int y)  
{  
    int a, b;  
    a = x * x;  
    b = y * y;  
}
```

При написании макроса с параметрами не должно быть пробела между именем макроса и открывающей скобкой при описании директивы **#define**; открывающая скобка должна идти вплотную к имени макроса!

# Директивы препроцессора C/C++

- Директива `#define` – не функция!

```
#define SQUARE(val) val * val  
void func (int x)  
{  
    int a;  
    a = SQUARE (x+1);  
}
```

```
void func (int x)  
{  
    int a;  
    a = x+1 * x+1;  
}
```



- Оператор `##` в теле макроса:

Если между двумя "словами" в теле макроса поставить `##`, то эти два "слова" будут склеены в одно (так называемый оператор конкатенации):

```
#define MACRO1(x,y) x##y  
#define MACRO2(x,y) x##y##trampampam  
MACRO1 (abc, def)  
MACRO2 (abc, def)
```

```
abcdef  
abcdeftrampampam
```



# Директивы препроцессора C/C++

- Для того, чтобы отменить макрос, существует директива **#undef**. Как только препроцессор встречает такую директиву, он "забывает" опеределённый ранее макрос и больше не заменяет его.

- Пример:

```
#define N 20
```

```
int a[N];
```

```
...
```

```
#undef N
```

```
void func (void)
```

```
{
```

```
int N;
```

```
N++;
```

```
}
```

```
int a[20];
```

```
void func (void)
```

```
{
```

```
int N;
```

```
N++;
```

```
}
```

# Директивы препроцессора C/C++

- Директива **#include** – вставляет текст из указанного файла.
  - Каждая библиотечная функция C имеет соответствующее описание в одном из заголовочных файлов.
  - Употребление директивы `include` не подключает соответствующую стандартную библиотеку, а только позволяют вставить в текст программы описания из указанного заголовочного файла.
  - Подключение кодов библиотеки осуществляется на этапе компоновки, т. е. после компиляции.
  - Хотя в заголовочных файлах содержатся все описания стандартных функций, в код программы включаются только те функции, которые используются в программе.
  - Пример:
    - #include** <stdio.h> – стандартная библиотека ввода/вывода C
    - #include** <iostream.h> – стандартная библиотека потокового ввода/вывода C++
    - #include** <math.h> – подключение математической библиотеки.

# Директивы препроцессора C/C++

- Директива **#pragma** — действие, зависящее от конкретной реализации компилятора. Это не совсем директива препроцессора в прямом смысле термина.
- Например, **#pragma** имя  
где имя — это имя директивы **#pragma**. Borland определяет 14 директив **#pragma**:
  - argused
  - anon\_struct
  - codeseg
  - comment
  - exit
  - hdrfile
  - hdrstop
  - inline
  - message
- Директивой **#pragma omp parallel** подключается внутреннее распараллеливание в технологии OpenMP

# Директивы препроцессора C/C++

- Директива `argused` должна стоять перед функцией. Она используется для устранения предупреждений, если аргумент функции, перед которой стоит `#pragma`, не используется в теле функции.
- Директива `exit` определяет одну или несколько функций, вызываемых при завершении программы.
- Директива `startup` определяет одну или несколько функций, вызываемых при запуске программы. Они имеют следующий вид:  
`#pragma exit имя_функции приоритет`  
`#pragma startup имя_функции приоритет`
- Приоритет — это значение от 64 до 255 (значения от 0 до 63 зарезервированы). Приоритет определяет порядок вызова функций. Если приоритет не указан, то по умолчанию предполагается значение 100.

# Директивы препроцессора C/C++

- Следующий пример определяет функцию `start()`, выполняющуюся в начале программы.

```
#include <stdio.h>
void start(void);
#pragma startup start 65
int main(void)
{
    printf("In main\n");
    return 0;
}
void start (void)
{
    printf("In start\n");
}
```

- В результате работы программы на экране появится:

In start

In main

# Директивы препроцессора C/C++

- Также имеется директива `inline`, имеющая следующий вид:

**#pragma inline**

Она сообщает компилятору, что программа содержит внутренний ассемблерный код. При создании самого быстрого кода компилятор должен знать, что в программе содержится ассемблерный код.

- Используя директиву `intrinsic`, можно указать компилятору на необходимость подстановки кода функции вместо ее вызова. Директива имеет вид:

**#pragma intrinsic имя\_функции**

где `имя_функции` — это имя функции, которую необходимо сделать внутренней.

- Директива `warn` позволяет запретить или разрешить различные предупреждения. Она имеет вид:

**#pragma warn установки**

# Директивы препроцессора C/C++. Условная компиляция

- Препроцессор языка Си предоставляет возможность компиляции с условиями. Это допускает возможность существования различных версий одного кода. Обычно, такой подход используется для настройки программы под платформу компилятора, состояние (отлаживаемый код может быть выделен в результирующем коде), или возможность проверки подключения файла строго один раз.
- Директивы, которые управляют условной компиляцией, позволяют исключить из процесса компиляции какие-либо части исходного файла посредством проверки условий (константных выражений).

```
#if <ограниченное-константное-выражение> [<текст>]  
[#elif <ограниченное-константное-выражение> <текст>]  
[#elif <ограниченное-константное-выражение> <текст>]  
[#else <текст>]  
#endif
```

- Директива `#if` совместно с директивами `#elif`, `#else` и `#endif` управляет компиляцией частей исходного файла.

# Условная компиляция. Примеры

```
#if условие 1  
фрагмент кода 1  
#elif условие 2  
фрагмент кода 2  
#else  
фрагмент кода 3  
#endif
```

- фрагмент кода 1 будет компилироваться, если выполняется условие 1,
- фрагмент кода 2 будет компилироваться, если выполняется условие 2,
- а фрагмент кода 3 будет компилироваться, если не выполняется ни одно из предыдущих условий.

## Условная компиляция. Примеры

- ❑ Необходимо написать программу, которая может работать на разных платформах.
- ❑ При этом в момент компиляции нужно знать, наша платформа `big endian` или `little endian`.
- ❑ На разных платформах работают компиляторы от разных разработчиков. Как правило, каждый компилятор выставляет некие предопределённые макросы, в том числе и макросы, определяющие тип процессора.
- ❑ Например, большинство компиляторов под архитектуру Intel выставляют макрос `__i386__`, компиляторы под Sparc архитектуру выставляют макрос `__sparc__` и т.д.

# Условная компиляция. Примеры

- Можно это учесть следующим образом:

```
#if (defined __i386__ ) || (defined __alpha__ )  
/* Архитектуры с little endian */  
#define LITTLE_ENDIAN  
#elif (defined __sparc__ )  
/* Архитектуры с big endian */  
#define BIG_ENDIAN  
#else  
#error "unknown architecture"  
#endif
```

- Здесь используется директива препроцессора **#error**, которая позволяет выдать сообщение с ошибкой или предупреждением. Директиву применяют совместно с директивами условной компиляции, чтобы отсечь некоторые недопустимые или неподдерживаемые комбинации настроек.

## Условная компиляция. Примеры

- ❑ В коде из предыдущего примера можно использовать вспомогательные директивы `#ifdef`, `#ifndef`:

```
#ifdef (__i386__) || (__alpha__)  
/* Архитектуры с little endian */  
#define LITTLE_ENDIAN  
#elif (defined __sparc__)  
/* Архитектуры с big endian */  
#define BIG_ENDIAN  
#else  
#error "unknown architecture"  
#endif
```

- ❑ `#ifndef (__i386__)` эквивалентно `#if (defined __i386__)`

# Константы

- *Константа* – это лексема, представляющая изображение фиксированного числового, строкового или символьного значения.
- Константы делятся на 5 групп:
  - целые;
  - вещественные (с плавающей точкой);
  - перечислимые;
  - символьные;
  - строковые.
- Компилятор выделяет лексему и относит ее к той или другой группе, а затем внутри группы к определенному типу по ее форме записи в тексте программы и по числовому значению.

# Константы

- *Целые константы* могут быть десятичными, восьмеричными и шестнадцатеричными.
  - Десятичная константа определяется как последовательность десятичных цифр, начинающаяся не с 0, если это число не 0 (примеры: 8, 0, 192345).
  - Восьмеричная константа – это константа, которая всегда начинается с 0. За 0 следуют восьмеричные цифры (примеры: 016 – десятичное значение 14, 01).
  - Шестнадцатеричные константы – последовательность шестнадцатеричных цифр, которым предшествуют символы 0x или 0X (примеры: 0xA, 0X00F).
- *Константы вещественного типа* занимают в памяти 8 байт (64 бита) и состоят из целой части (со знаком); десятичной точки, отделяющей дробную часть от целой; символа экспоненты e или E: 1.75, 2.5e-2, -0.02E+13

# КОНСТАНТЫ

- *Перечислимые константы* вводятся с помощью ключевого слова **enum**. Это обычные целые константы, которым приписаны уникальны и удобные для использования обозначения.

Примеры:

- `enum { one=1, two=2, three=3, four=4 };`
- `enum { zero, one, two, three }.`
- `enum { ten=10, three=3, four, five, six };`
- `enum { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday } ;`

# Константы

- ❑ *Символьные константы* – это один или два символа, заключенные в апострофы.
- ❑ Символьные константы, состоящие из одного символа, имеют тип `char` и занимают в памяти один байт.
- ❑ Символьные константы, состоящие из двух символов, имеют тип `int` и занимают два байта.
- ❑ Последовательности, начинающиеся со знака `\`, называются управляющими, они используются:
  - Для представления символов, не имеющих графического отображения, например:
    - ✓ `\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`).

# Константы

- *Строковая константа* – это последовательность символов, заключенная в кавычки. Внутри строк также могут использоваться управляющие символы.
- Например:  
“\nНовая строка”,  
“\n\”Алгоритмические языки программирования  
высокого уровня \”””

# Именованные и неименованные константы

- ❑ Тип **неименованной константы** неявно определяется ее значением: 0X2F56, -0x2A13B, True, „2“, “124”, {red, yellow, green}, NULL...
- ❑ **Именованную константу** можно описать, присвоив ей идентификатор (имя), который можно будет затем использовать в программе вместо того, чтобы непосредственно записывать значение константы:  

```
const float Pi = 3.1415926;  
const int N = 20;  
const string Text = “Hello, world”;
```

и т.д.
- ❑ Попытка где-то в тексте изменить значение константы приведет к ошибке компиляции!

# ТИПЫ ДАННЫХ

- В языках C и C++ есть шесть основных типов данных:
  - **int** (целый);
  - **char** (символьный);
  - **wchar\_t** (расширенный символьный);
  - **bool** (логический);
  - **float** (вещественный);
  - **double** (вещественный с двойной точностью).
- Первые четыре типа называют целочисленными (целыми), последние два – типами с плавающей точкой.
- Существует четыре спецификатора типа, уточняющих внутреннее представление и диапазон значений стандартных типов:
  - **short** (короткий);
  - **long** (длинный);
  - **signed** (знаковый);

# ТИПЫ ДАННЫХ

- Размер типа **int** не определяется стандартом, а зависит от компьютера и компилятора. Для 32-разрядного процессора под величины этого типа отводится 4 байта.
  - Спецификатор **short** перед именем типа указывает компилятору, что под число требуется отвести 2 байта независимо от разрядности процессора.
  - По умолчанию все целочисленные типы считаются знаковыми, то есть спецификатор **signed** можно опускать.
  - Типы **short int**, **long int**, **signed int** и **unsigned int** можно сокращать до **short**, **long**, **signed** и **unsigned** соответственно.

# ТИПЫ ДАННЫХ

- Под величину символьного типа **char** отводится количество байт, достаточное для размещения любого символа из набора символов для данного компьютера, что и обусловило название типа.
- Как правило, это 1 байт. Тип **char**, как и другие целые типы, может быть со знаком или без знака. В величинах со знаком можно хранить значения в диапазоне от -128 до 127.
- При использовании спецификатора **unsigned** значения могут находиться в пределах от 0 до 255. Этого достаточно для хранения любого символа из 256-символьного набора ASCII (**unsigned char**). Величины типа **char** применяются также для хранения целых чисел, не превышающих границы указанных диапазонов.

# ТИПЫ ДАННЫХ

- ❑ Расширенный символьный тип **wchar\_t** предназначен для работы с набором символов, для кодировки которых недостаточно 1 байта, например, Unicode (2 байта).
- ❑ Размер этого типа зависит от реализации; как правило, он соответствует типу **short**. Строковые константы типа **wchar\_t** записываются с префиксом **L**, например, **L"Ġates"**.
- ❑ Величины логического типа **bool** могут принимать только значения **true** и **false**, являющиеся зарезервированными словами. Внутренняя форма представления значения **false** – 0 (нуль). Любое другое значение интерпретируется как **true**. При преобразовании к целому типу **true** имеет значение 1.

# Типы данных

- Типы данных с плавающей точкой хранятся в памяти компьютера иначе, чем целочисленные. Внутреннее представление вещественного числа состоит из двух частей — мантиссы и порядка.
- 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.
- Для величин типа **double**, занимающих 8 байт, под порядок и мантиссу отводится 11 и 52 разряда соответственно. Длина мантиссы определяет точность числа, а длина порядка — его диапазон.
- Спецификатор **long** перед именем типа **double** указывает, что под его величину отводится 10 байт.

# Типы данных

## Тип `void`

- Кроме перечисленных, к основным типам языка относится тип **`void`**, но множество значений этого типа пусто. Он используется для определения функций, которые не возвращают значения, для указания пустого списка аргументов функции, как базовый тип для указателей и в операции приведения типов.

- Пример:

```
void main()  
{  
...  
}
```

# Переменная

- *Переменная это именованная величина, значение которой может меняться*
- **Имя (идентификатор) переменной**
  - используется для ссылки на переменную: **a + b**
  - может содержать символы из числа
    - ✓ строчные и прописные символы латиницы
    - ✓ цифры
    - ✓ символ «подчерк» — «\_»
  - начинается не с цифры
  - **строчные и прописные символы различаются:**

**MyName** ≠ **myname**

# Переменная

- Перед использованием любая переменная должна быть описана. Примеры:

```
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. Если класс памяти не указан явно, то компилятор определяет его исходя из контекста объявления.
- **Const** – показывает, что эту переменную нельзя изменять (именованная константа).

# Классы памяти переменных

- ❑ **auto** – автоматическая локальная переменная. Спецификатор `auto` может быть задан только при определении объектов блока, например, в теле функции. 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, она существует только в пределах того файла, где определена переменная.
- ❑ **register** – аналогичны `auto`, но память под них выделяется в регистрах процессора. Если такой возможности нет, то переменные обрабатываются как `auto`.

# Блок

- *Группа операторов в фигурных скобках, составляющее некоторое целое (тело цикла, варианты у оператора **if**, тело функции и т.д.):*

```
{  
    a = b + c;  
    ...  
}
```

- В блоке могут быть объявлены (и инициализированы) переменные:

```
{  
    int c = a + b;  
    ...  
}
```

# Переменные

## Пример

```
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
```

# Время жизни и области видимости

- ❑ Объявленные в блоке переменные являются *локальными* — они «видны» (и существуют) только внутри блока
- ❑ Блоки могут вкладываться, а объявленные переменные — перекрываться:

```
{  
    int a = 3;  
    double b = 2.0;  
    {  
        int a, b; // Эти a и b перекрывают внешние  
        a = 5;   // присваивание локальной a  
    }  
    // Здесь опять внешние a и b, a = 3, b = 2.0  
}
```

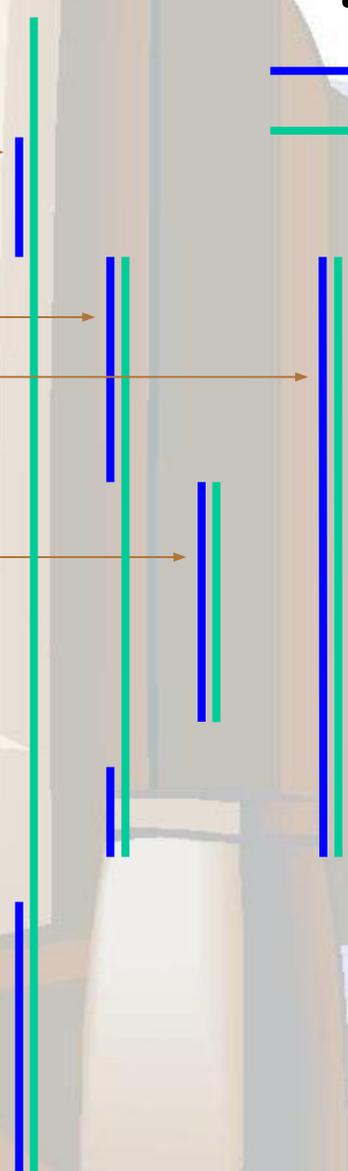
# Время жизни и области видимости

- Время жизни переменной — это участок программы от момента ее объявления и до конца блока
- Область видимости переменной — это время ее жизни минус области перекрытия имени

# Время жизни и области видимости

```
.....  
int i = 2; // файловая  
int f1(int a, int b)  
{  
    int i;  
    int j = 3;  
    i = j * 2;  
    {  
        int i;  
        int k;  
        i = j * 3;  
    }  
    return b;  
}  
int f2(int p)  
{  
}
```

— область видимости  
— время жизни



# И еще о блоках

## □ C vs. C++

- в C переменные могут объявляться только в начале блока
- в C++ переменные могут объявляться в любом месте блока

## □ В C++ переменные можно объявлять в заголовках циклов **for**, **while**, в **if** и **switch**:

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

```
while ( double s = sin(alpha) != 0.0 )...
```

```
if ( int sign = value >= 0 ? 1 : -1 )...
```

```
switch( int b = a + 1 )...
```

# Объявления namespace и using

- Ключевое слово **namespace** позволяет разделить глобальное пространство имен путем создания некоторой декларативной области. По сути, пространство имен определяет область видимости.
- Общая форма задания пространства имен имеет следующий вид:

```
namespace имя  
{  
    //Объявления  
}
```

# Объявления namespace и using

- Пример объявления и использования:

```
namespace MyNameSpace {  
    int i, k;  
    void myfunc(int j) { ... }  
}
```

- Поскольку пространство имен определяет область видимости, для доступа к определенным внутри нее объектам необходимо использовать оператор разрешения видимости. Например, чтобы присвоить значение 10 переменной *i*, необходимо использовать следующий оператор:

```
MyNameSpace::i = 10;
```

# Объявления namespace и using

- Если члены пространства имен будут использоваться часто, то для упрощения доступа к ним можно применить директиву **using**. Оператор **using** имеет две общие формы записи:

**using namespace** имя;

**using** имя::член;

- Пример:

```
using MyNameSpace::k; //только переменная k сделана видимой
```

```
k = 10; //ОК, поскольку переменная k видима
```

```
using namespace MyNameSpace; //все члены MyNameSpace  
ВИДИМЫ
```

```
i = 10; //ОК, поскольку все члены MyNameSpace сейчас видимы
```

# Преобразование типов в C/C++

- Преобразования типов выполняются, если операнды, входящие в выражения, имеют различные типы. Ниже приведена последовательность преобразований.
  - Если один из операндов имеет тип **long double**, то другой преобразуется к типу `long double`.
  - Если один из операндов имеет тип **double**, то другой преобразуется к типу `double`.
  - Если один из операндов имеет тип **float**, то другой преобразуется к типу `float`.
  - Иначе, если один из операндов имеет тип **unsigned long**, то другой преобразуется к типу `unsigned long`.
  - Иначе, если один из операндов имеет тип **long**, то другой преобразуется к типу `long`.
  - Иначе, если один из операндов имеет тип **unsigned**, то другой преобразуется к типу `unsigned`.

# Структура программы. Основная функция

- ❑ Основной функцией программы в C++ называют функцию, выполняющуюся при запуске программы.
- ❑ Основная функция имеет имя **main**. Как правило этой функции не нужны аргументы и выводимый тип.
- ❑ **void main() {}**
- ❑ Иногда среды разработки предлагают в начале работы собственный вид данной функции.
- ❑ Таким образом, рекомендуемая заготовка для написания программы на языке C++ имеет следующий вид:
- ❑ **#include <iostream.h>**
- ❑ **#include <stdlib.h>**
- ❑ **void main()**
- ❑ **{**
- ❑ **// текст программы**
- ❑ **cin.get(); // приостановка выполнения**
- ❑ **}**

# Операции языка C++

- Основные арифметические операции:  $+$ ,  $-$ ,  $*$ ,  $/$ .
- В случае деления целых чисел результат будет частным двух чисел. Для получения остатка используется операция  $\%$ .
- Для сокращения записи вместо операций типа  $a=a+b$  применяют также запись  $a+=b$ . Аналогичный смысл имеют записи:  $a-=b$ ,  $a*=b$ ,  $a/=b$ ,  $a\%=b$ .
- Две операции  $++$  и  $--$  введены для увеличения (уменьшения) переменных ровно на 1, что часто встречается при циклических вычислениях, индексации массивов и в ряде других случаев.
- $a++$ ; // эквивалент операции  $a = a + 1$ ; или  $a += 1$ ;
- $a--$ ; // эквивалент операции  $a = a - 1$ ; или  $a -= 1$ ;

# Операции языка C++

- Если символы ++ (--), расположенные слева от переменной, то сначала переменная увеличивается (уменьшается) на 1, а затем производится вычисление выражения.
- И наоборот, если символы сложения (вычитания) расположены справа от идентификатора, первоначально вычисляется арифметическое выражение, а затем переменная увеличивается (уменьшается) на 1.
- Примеры использования префиксного и постфиксного использования операторов инкремента (++) и декремента (--):

```
int m=1,n=2;
```

```
int a=(m++)+n;
```

```
int b=m+(++n);
```

# Операции языка C++

□ Пример:

```
main()
{ int n, b, c;
  n=1;
  b=10*(n++); // сначала вычисляется  $b = 10 * 1 = 10$ , а затем  $n = n + 1 = 2$ ;
  c=10*(++n); // сначала вычисляется  $n = n + 1 = 3$ , а после этого  $c = 10 * 3 = 30$ ;
}
```

□ Логические операции:

□ В C++ в качестве FALSE используется значение целого типа (int), равное 0, в качестве TRUE - любое ненулевое значение целого типа.

==	равно	<, >, <=, >=	Больше, меньше
!=	не равно	&&	логическое И (and)
!	отрицание (not)		логическое ИЛИ (or)

# Операторы:

- ❑ Оператор присваивания
- ❑ Оператор ветвления **if**
- ❑ Переключатель **switch**
- ❑ Цикл с предусловием **while**
- ❑ Цикл с постусловием **do-while**
- ❑ Цикл **for**
- ❑ Операторы **goto, break; continue**

# Операторы языка C++

## 1. Оператор (или операция) присваивания =.

Примеры:

```
a=b=2; //присваивание числового значения
```

```
b=a+c; // присваивание результата действия
```

□ Простой пример:

```
int a, b, c;
```

```
a = b = c * 3;
```

□ «Непростой» пример:

```
int a, b, c;
```

```
a = (b = 5) + (c = 3);
```

2. **Составной оператор** — последовательность любых операторов, заключенная в фигурные скобки { }, которые являются в C и C++ аналогом слов **begin** и **end**.

# Операторы языка C++

## 3. Условные операторы

### 3.1. Логический оператор присваивания (тернарная операция):

**переменная=(логическое\_выражение)?выр-ие1:выр-ие2;**

выполняется следующим образом: если логическое\_выражение истинно, то переменной присваивается результат выражения1, а в противном случае - результат выражения2.

**x=(x>0)? x: -x;**

### 3.2. Условный оператор if:

**if(выражение ) оператор;**

Пример:

```
if (f==0 || f==1) { g=1; h=2; i=3; }
```

Другая форма оператора if – оператор ветвления:

**if(выражение) оператор\_1; else оператор\_2;**

Если истинно выражение, то выполняется оператор\_1, иначе - оператор\_2.

# Пример: решение квадратного ур-я

```
double a, b, c;
double d, x1, x2, re, im;
d = b * b - 4.0 * a * c;
if ( d > 0.0 )
{
    d = sqrt(d);
    x1 = (-b + d) / (2.0 * a);
    x2 = (-b - d) / (2.0 * a);
}
else if ( d < 0.0 )
{
    d = sqrt(-d);
    re = -b / (2.0 * a);
    im = d / (2.0 * a);
}
else
    x1 = x2 = -b / (2.0 * a);
```

# Операторы языка C++

## 3.3. Оператор переключения switch: switch(выражение)

```
{ case    const_1: оператор_1; break;
  case    const_2: оператор_2; break;
  //      . . .
  default: оператор_N; break;
}
```

- ❑ выражение — целочисленное арифметическое выражение;
- ❑ `const_i` - символьная или целая константа;
- ❑ `break` - оператор прерывания, обеспечивающий выход из оператора `switch`.

# Переключатель switch

- Пример:

```
switch ( n )  
{  
    case 1: cout << "Понедельник";  
    case 2: cout << "Вторник";  
    case 3: cout << "Среда";  
    case 4: cout << "Четверг";  
    case 5: cout << "Пятница";  
    case 6: cout << "Суббота";  
    case 7: cout << "Воскресенье";  
    default: cout << "Неверный номер";  
}
```

- Этот пример при  $n = 1$  выведет:

ПонедельникВторникСредаЧетвергПятницаСубботаВоскресе  
ньеНеверный номер

# Переключатель switch

- Выход из блока — оператор **break**
- Пример:

```
switch ( n )  
{  
    case 1: cout << "Понедельник"; break;  
    case 2: cout << "Вторник";      break;  
    case 3: cout << "Среда";        break;  
    case 4: cout << "Четверг";      break;  
    case 5: cout << "Пятница";     break;  
    case 6: cout << "Суббота";     break;  
    case 7: cout << "Воскресенье"; break;  
    default: cout << "Неверный номер";  
}
```

# Операторы языка C++

## 4. Операторы цикла

### 4.1. Оператор цикла for:

**for (инициализация; условие; изменение) оператор;**

**Инициализация** – определение начальных значений, меняющихся при выполнении цикла величин.

**Оператор** – повторяющиеся операции, на которые распространяется оператор цикла; в случае, когда необходимо циклическое выполнение группы операторов, их следует заключать в операторные скобки.

**Изменение** – правила изменения переменных величин по окончании очередного цикла.

**Условие** – логическое выражение, определяющее условие выполнения цикла.

Поля **инициализация** и **изменение** могут содержать список операций, перечисляемых через запятую. Следует отметить, что любое из указанных полей может отсутствовать; при этом разделители ; должны сохраняться.

# Операторы языка C++

Пример: Программа вычисления суммы первых ста натуральных чисел.

```
#include <iostream.h>  
void main()  
{ int sum, count;  
  for (count=1, sum=0; count<=100; count++)  
    sum+=count;  
  cout<<"Summa равна " <<sum; //вывод на экран  
  cin.get();  
}
```

# Цикл for: пример

- Суммирование отрезка ряда:  $1+a-a^2+a^3+\dots+(-1)^{n-1}a^n$ :

```
int n, i;
double a, da, sum;

cout << "a, n = ";
cin >> a >> n;

da = 1.0;
sum = 1.0;
for ( i = 1; i <= n; i++ )
{
    da = da * a;
    if ( i % 2 ) sum = sum - da;
    else       sum = sum + da;
}
```

# Операторы языка C++

## 4.2. Оператор цикла while:

**while (выражение) оператор;**

Пока справедливо логическое **выражение**, будет выполняться **оператор**.

```
#include <iostream.h>
void main()
{
    sum = 0;
    while ( n )
    {
        sum+ = n % 10;
        n = n / 10;
    }
    cout<<"Summa ravna " <<sum;
    cin.get(); }
```

# Операторы языка C++

## 4.3. Оператор цикла do-while:

**do оператор; while(выражение);**

**Оператор** будет выполняться, пока справедливо логическое **выражение**.

Пример:

```
#include <iostream.h>  
void main()  
{  
    int sum=0, count=1;  
    do sum+=count++; while (count<=100);  
    cout<<"Summa равна "<<sum;  
    cin.get();  
}
```

В отличие от языка Паскаль оператор цикла с постусловием задает условие продолжения, а не окончания цикла.

# Ввод-вывод данных

- В языке C++ появились функции ввода-вывода **cin** и **cout**, находящиеся в модуле **iostream**.
- Для вывода данных на экран используется функция **cout<<**:
- Настройка вывода осуществляется при помощи команд: **cout.width(n)**, где **n** – ширина поля вывода и **cout.precision(m)**, где **m** – точность вывода вещественных чисел.
- Пример:

```
#include <iostream.h>
void main()
{ float pi=3.1415926;
  cout.precision(5);
  cout<<"Chislo pi:" << '\n' <<pi;
  cin.get();
}
```

Результат: Chislo pi:  
3.1416

# ВВОД-ВЫВОД ДАННЫХ

Для ввода данных с клавиатуры используется процедура **cin>>**.

Пример: решение квадратного уравнения

```
#include <iostream.h>
void main()
{ float a,b,c,d,x1,x2;
  cout<<"a*x*x+b*x+c=0"<<"\n"<<"Vvedite a:";
  cin>>a;
  cout<<"Vvedite b:"; cin>>b;
  cout<<"Vvedite c:"; cin>>c;
  d=b*b-4*a*c;
  if (d<0) cout<<"Net korney";
  else {
    x1=(-b+sqrt(d))/(2*a);
    x2=(-b-sqrt(d))/(2*a);
    cout<<"x1="<<x1<<"\n"<<"x2="<<x2;
  }
  cin.get();
}
```

# Безусловный переход goto

- Синтаксис:

```
goto <идентификатор>;
```

```
...
```

```
<идентификатор>: <инструкция>
```

- Осуществляет переход к выполнению программы начиная с <инструкции>

```
    if ( a > b ) goto 13;  
    if ( b > c ) goto 12;  
11: cout << "Наибольшее число =" << c;  
    goto 14;  
12: cout << "Наибольшее число =" << b;  
    goto 14;  
13: if ( a < c ) goto 11;  
    cout << "Наибольшее число =" << a;  
14: ;
```

# Break, continue, return

- ❑ **break** - оператор прерывания цикла.
- ❑ **continue** - переход к следующей итерации цикла. Он используется, когда тело цикла содержит ветвления.
- ❑ Оператор **return** – оператор возврата из функции. Он всегда завершает выполнение функции и передает управление в точку ее вызова. Вид оператора:  
**return** [выражение];

```
{  
if  
(<выражение_условие>  
break;  
<операторы>  
}  
  
for( k=0,s=0,x=1;x!=0;)  
{  
cin>>x;  
if (x<=0) continue;  
k++;s+=x;  
}
```