Основы программирования на C++

для зумеров и примкнувшим к ним

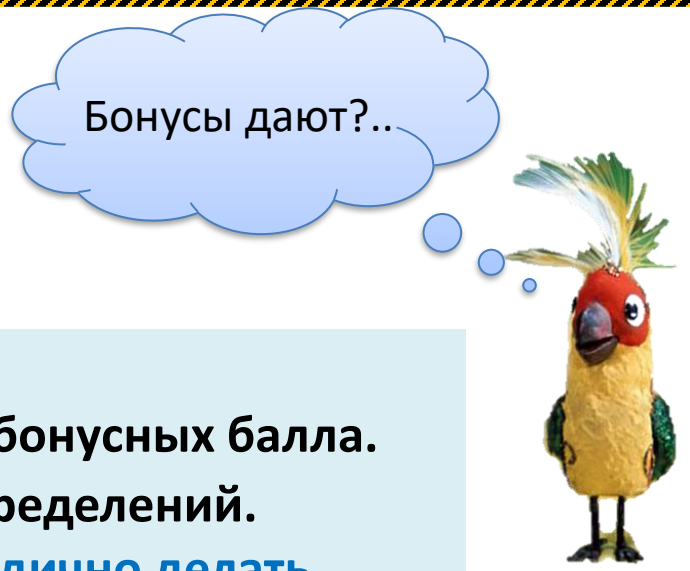
Лекция 3

База и ревизия пройденного

(часть 2)

Сергей Шершаков

22 января 2026



В конце лекции будет новый вопрос на 2 бонусных балла. Вопрос будет касаться систематизации определений. Чтобы ответить на него, необходимо методично делать заметки, которые помогут в конце набрать наибольшее число баллов и обойти конкурентов.

Рецепт (модифицированный):

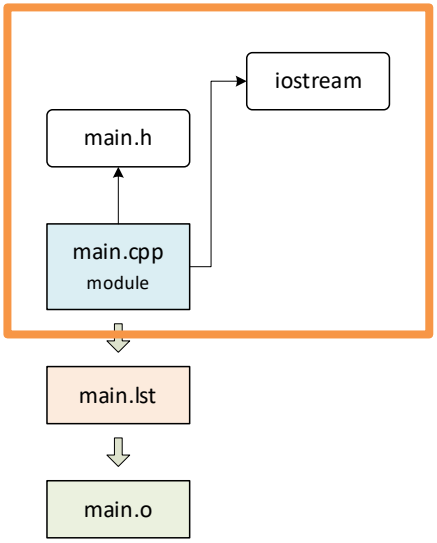
1) Тетрадка для записей (заранее заготовить из нее традиционные поллисточка — сегодня побольше! — бумаги для ответа на вопрос квиза)
2) ручка
3) написать свои ФИО полностью, группу с п/г, дату
4) написать ответ на вопрос, который появится на слайде
5) *<секретный>*
6) сдать перед покиданием аудитории

В прошлый раз…

Назначение препроцессора

- Препроцессор унаследован от C

- C++ (как и C) являются кроссплатформенными на уровне исходного кода
  - для примера: как говорят про кроссплатформенность Java?



- Задача препроцессора — подготовка текста исходного кода под конкретные условия билда
  - буквально, сборка исходника из разных фрагментов; похоже на «копипасту» при изготовлении реферата без использования ИИ(*)
  - *разные условия*: режимы построения (debug, release); использование ключевых библиотек (multithread), строковых моделей (узкие/широкие символы); разные платформы (процессоры, разрядность) и операционные системы (свои особенности для Linux, Win, Mac), и т.д.

(*) Болезненный опыт поколения, у которого интернет уже был, а адекватных поисковиков, источников информации и ИИ — еще нет

решена!

Проблема повторного косвенного включения (3)

impl_include_problem\foo.h

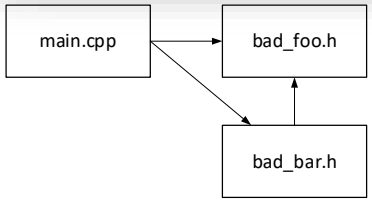
```
#ifndef FOO_H_
#define FOO_H_

// this is the foo.h content
double power(double x, unsigned int p);

int GLOBAL_I;

#endif // FOO_H_
```

bar.h

```
#ifndef BAR_H_
#define BAR_H_

#include "foo.h"

// this is the bad_foo.h content
double power2(double x, double p);


#endif // BAR_H_
```

impl_include_problem\main.cpp*

```
// example of the implicit include p
#include "foo.h"
#include "bar.h"


int main()
{
    return 0;
}
```



✅ топчик

main.lst*

```
# 0 "main.cpp"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "main.cpp"


# 1 "foo.h" 1

double power(double x, unsigned int p);

int GLOBAL_I;
# 7 "main.cpp" 2
# 1 "bar.h" 1


double power2(double x, double p);
# 8 "main.cpp" 2


int main()
{
    return 0;
}
```

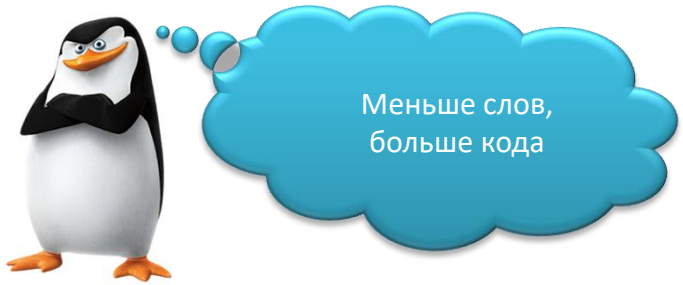
Что обычно помещают в headers?

- Заголовочные файлы — это *интерфейсная часть модулей*, содержат (как правило) только декларации
  - прототипы функций
  - декларации объектов (как это?!)
  - описания пользовательских типов
    - без реализаций методов — для нешаблонных классов
    - enum и прочие union
  - описания констант
    - тут много нюансов и, к тому же, они сильно зависят от стандарта языка и иногда даже особенностей реализации компилятора

- Шаблоны (функций и классов) в большинстве случаев размещаются в заголовочных файлов **целиком!**
  - шаблон — это не (готовый) код. А что?

Что в headers класть не стоит?

- Все то, что компилируется на уровне одного CPP
- При включении такого заголовка более, чем в один CPP, линкер распереживается при попытке собрать все вместе, так как не будет знать, кого/что использовать. Например:
  - определение функций
  - определение объектов



headers_w_unwanted problem

Ответ на

Вопрос из лекции 2 на бонусный балл

На лекции мы рассматривали примеры нескольких заголовочных файлов, демонстрирующих определенные концепции. В некоторых из них была заложена «мина» замедленного действия, которая незаметна в проекте, который рассматривался нами, но обязательно «рванет», если к этому проекту добавить еще хотя бы один модуль трансляции.

Найдите «мину» и обезвредьте ее (укажите примеры заголовков, где была проблема).

3 мин

— А где ответ?
— А ответ на следующем (или предыдущем) слайде!

решена!

Проблема повторного косвенного включения (3)

impl_include_problem\foo.h

```
#ifndef FOO_H_
#define FOO_H_

// this is the foo.h content
double power(double x, unsigned int p);

int GLOBAL_I;

#endif // FOO_H_
```

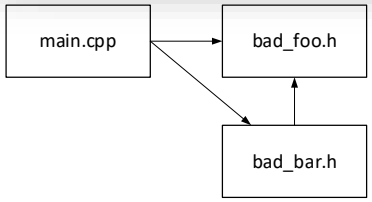
bar.h

```
#ifndef BAR_H_
#define BAR_H_

#include "foo.h"

// this is the bad_foo.h content
double power2(double x, double p);
```

Определение глобальной переменной, конечно, можно положить в заголовочный файл, но включить такой хедер получится только в один модуль трансляции.

impl_include_problem\main.cpp*

```
// example of the implicit include p
#include "foo.h"
#include "bar.h"


int main()
{
    return 0;
}
```



✅ топчик

```
# 1 "main.cpp"

# 1 "foo.h" 1

double power(double x, unsigned int p);

int GLOBAL_I;
# 7 "main.cpp" 2
# 1 "bar.h" 1


double power2(double x, double p);
# 8 "main.cpp" 2


int main()
{
    return 0;
}
```

Описание и Определение

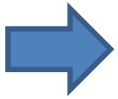
Declaration:

- *описание* создает «знание для компилятора»(*) о том, что в программе есть что-то такое, чем можно пользоваться так-то:
  - знание о типе (описание типа)
  - знание об объекте (!)
  - знание о функции (прототип)
- описание как правило НЕ влияет на ассемблерный вывод и не существует после компиляции программы
- описание — это *идеализм*!

Definition:

- *определение* имеет результатом данные либо в памяти, либо в ассемблерном коде программы:
  - *создание объекта (в памяти)*
  - *тело функции (в ассемблерном коде)*
  - …
- *определение* чего-то как правило влечет за собой и описание этого же:
  - *определение функции* невозможно без ее заголовка
- *определение* — это *материализм*!

Сегодня



- Программа на C++: основные элементы
- Объекты и типы
- Выражения и операторы
- Логические выражения и CF-инструкции

Дальше побежим ооочень быстро

ПРОГРАММА НА C++: ОСНОВНЫЕ ЭЛЕМЕНТЫ

C++ Alphabet

- Keywords and identifiers:
  - English letters: A..Z, a..z (beware of Russian “C”)
  - digits: 0, 1, … 9
  - underscore: _
- Operator symbols:
  - +, –, *, /, %, =, ==, !=, <, >, &, *, , , (,)…
- End of statement symbol: ;
- Block of statements: { }
- Preprocessor directives:
  - start with #
- Comments:
  - line comments indicated by // prefix
  - block comments framed by /* */ pairs of symbols
- Escape sequences:
  - starts with \ : \n, \t, \', \", \\, …

String literals can contain any symbols according to the code page of a source file (ANSI, UTF-8)

Hello, %username%

```
#include <iostream>
#include <string>

using std::string;
using std::cout;
using std::cin;

// Asks a user for a name and displays a greeting on the screen.
int main()
{
    string name;
    cout << "Your name: ";
    cin >> name;

    cout << "Hello, " << name << "!\n\n";

    return 0;
}
```

Keywords

up to C++11 standard

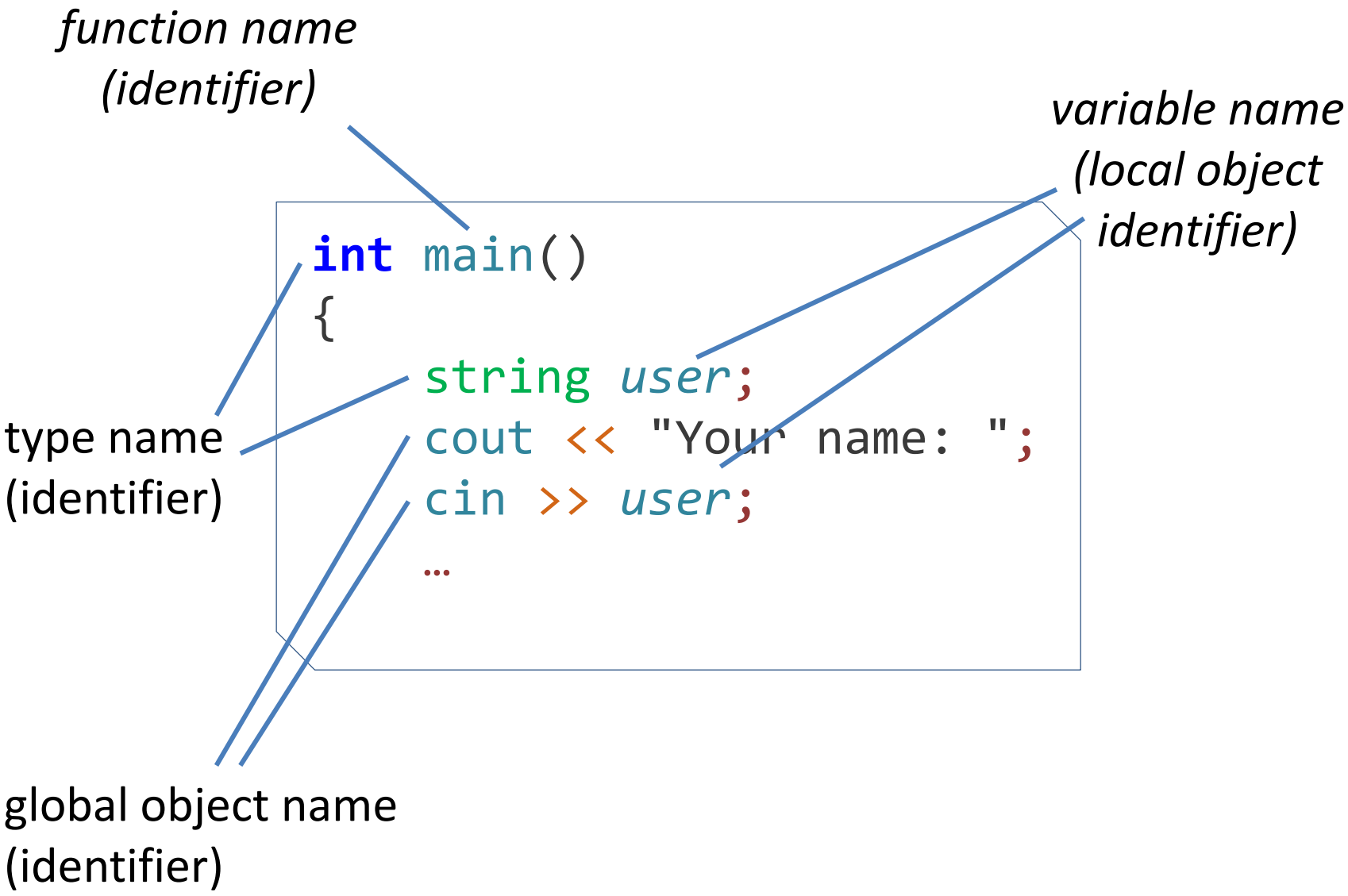
alignas (C++11)
alignof (C++11)
and
and_eq
asm
auto
bitand
bitor
bool
break
case
catch
char
char16_t (C++11)
char32_t (C++11)
class
compl
const
constexpr (C++11)
const_cast
continue
decltype (C++11)
default
delete
do
double
dynamic_cast
else
enum
explicit
export
extern
false
float
for
friend
goto
if
inline
int
long
mutable
namespace
new
noexcept (C++11)
not
not_eq
nullptr (C++11)
operator
or
or_eq
private
protected
public
register
reinterpret_cast
return
short
signed
sizeof
static
static_assert (C++11)
static_cast
struct
switch
template
this
thread_local (C++11)
throw
true
try
typedef
typeid
typename
union
unsigned
using
virtual
void
volatile
wchar_t
while
xor
xor_eq

https://en.cppreference.com/w/cpp/keyword

Identifiers

- Used for naming types, objects, variables, functions and so on
  - The only characters one can use in the names are alphabetic characters (A..Z, a..z), numeric digits (0… 9), and the underscore (_) character.
  - The first character in a name cannot be a numeric digit.
  - Uppercase characters are considered distinct from lowercase characters.
  - One can’t use a C++ keyword for a name.
  - There are no limits on the length of a name, but a reasonable size is expected.

Identifiers



Identifiers: examples

```
int foo;            // valid
int Foo;            // valid and distinct from Foo
int FOO;            // valid and even more distinct
Int bar;            // invalid(*) – has to be int, not Int
int my_class1;      // valid
int _Myclass2;      // valid but reserved – starts with underscore
int 2object;        // invalid because starts with a digit
int double;         // invalid – double is a C++ keyword
int begin;          // valid – begin is a Pascal keyword
int __circle;       // valid but reserved – starts with two underscores
int the_simple_integer_variable_version_2; // valid
int walkie-talkie; // invalid – no hyphens allowed
```

(*) `Int` here is invalid ID *conditionally*.

Statements (Инструкции)

Statement (инструкция) — правильная запись (предложение) на языке C++, оканчивающееся символом точка с запятой ;

Можно (условно) рассматривать в качестве инструкции последовательность инструкций в фигурных скобках { }.

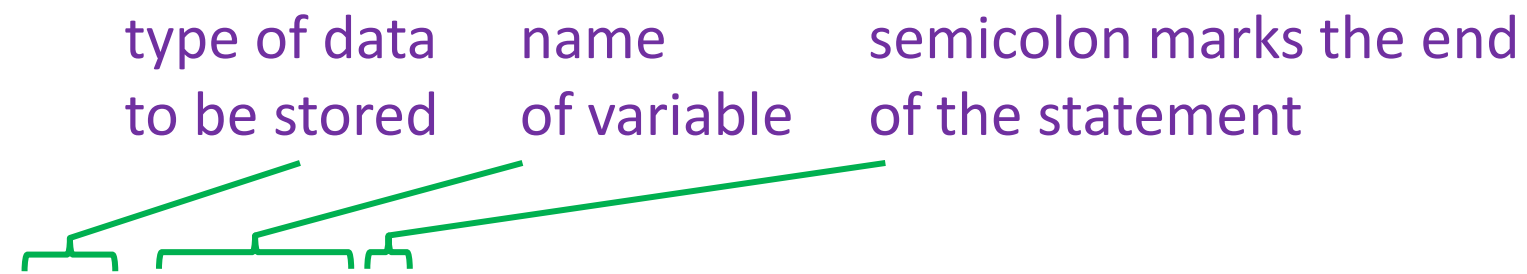
```
#include <iostream>
using namespace std;          // using statement

int main()
{
   string user;               // definition and declaration statement
   cout << "Your name: ";     // expression statement
   cin >> user;               // expression statement
   if (user.length() > 30)
   {
      cout << "Name is too long!\n";
      cout << "Your shortname: \n";
      cin >> user;
   }
   if (user.length() == 0)
      cout << “You didn’t specify a name!\n";
   cout << "Hello, " << user << "!\n"; // expression (complex)
   return 0;                  // return statement
}
```

Инструкции: *сорта и специи*

- *Прототип функции — …*

Инструкции определения и описания объекта

- *Object Definition* (определение объекта) означает его создание.
- *Object Declaration* (описание объекта) обозначение объекта, существующего *где-то там*, в текущей области видимости.




```
int number; // definition and declaration

extern int anotherNumber;  // declaration only
```

Ключевое слово обозначает, что объект *определен* (создан) где-то в другом месте

«объект типа int с именем Степан где-то там на галёрке…»

Инструкции: *сорта и специи*

- *Прототип функции* — инструкция *описания* (*definition*) контракта функции
  - `double foo(double x, int p);`

Инструкции: *сорта и специи*

- *Прототип функции* — инструкция *описания* (*definition*) контракта функции
  - `double foo(double x, int p);`

- Инструкция *определения* и *описания* объекта
  - `int a;`

Инструкции: *сорта и специи*

- *Прототип функции* — инструкция *описания* (*definition*) контракта функции
  - `double foo(double x, int p);`

- Инструкция *определения* и *описания* объекта
  - `int a;`

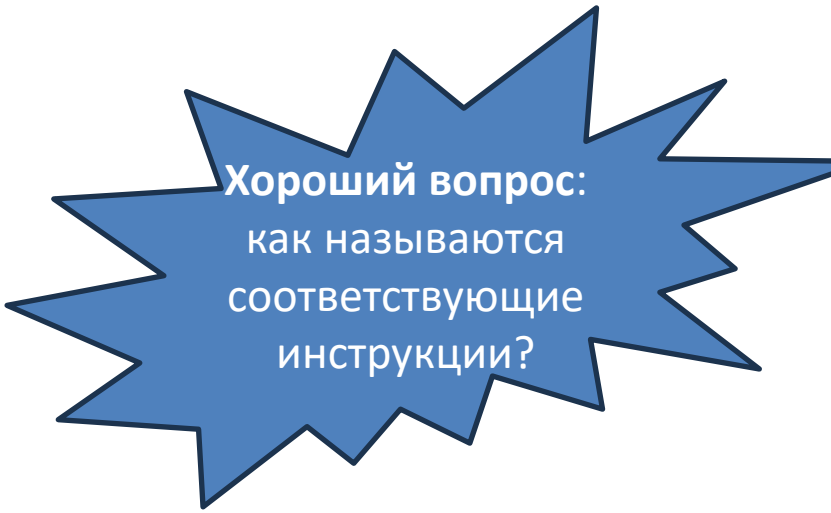
- Инструкция *(только)* описания объекта:
  - `extern int a;`

Инициализация и присваивание

- *Инициализация* — задание некоторого значения объекта в момент его конструирования (работает *конструктор копирования*):

❓ зачем?

```
int num = 42;
int num2(42);
int num42 {42};
int num43 = {42};
```



- *Присваивание* — это перезадание текущего значения для уже существующего объекта на другое значение
(работает *оператор копирующего присваивания*):

```
num = 14;
```

Инструкции: *сорта и специи*

- *Прототип функции* — инструкция *описания* (*definition*) контракта функции
  - `double foo(double x, int p);`

- Инструкция *определения* и *описания* объекта
  - `int a;`
  - разновидность: *с инициализацией*: `int a = 42;`

- Инструкция *(только)* описания объекта:
  - `extern int a;`

Block of Statements

- Block of statements { } allows putting a set of statements in a place where the only one statement is expected.
- Block of statements introduces an inner *scope* for objects declared in the block:
  - an object in the inner scope is not visible (accessible) in the outer scope;
  - the lifetime of such an object is limited by the block boundaries

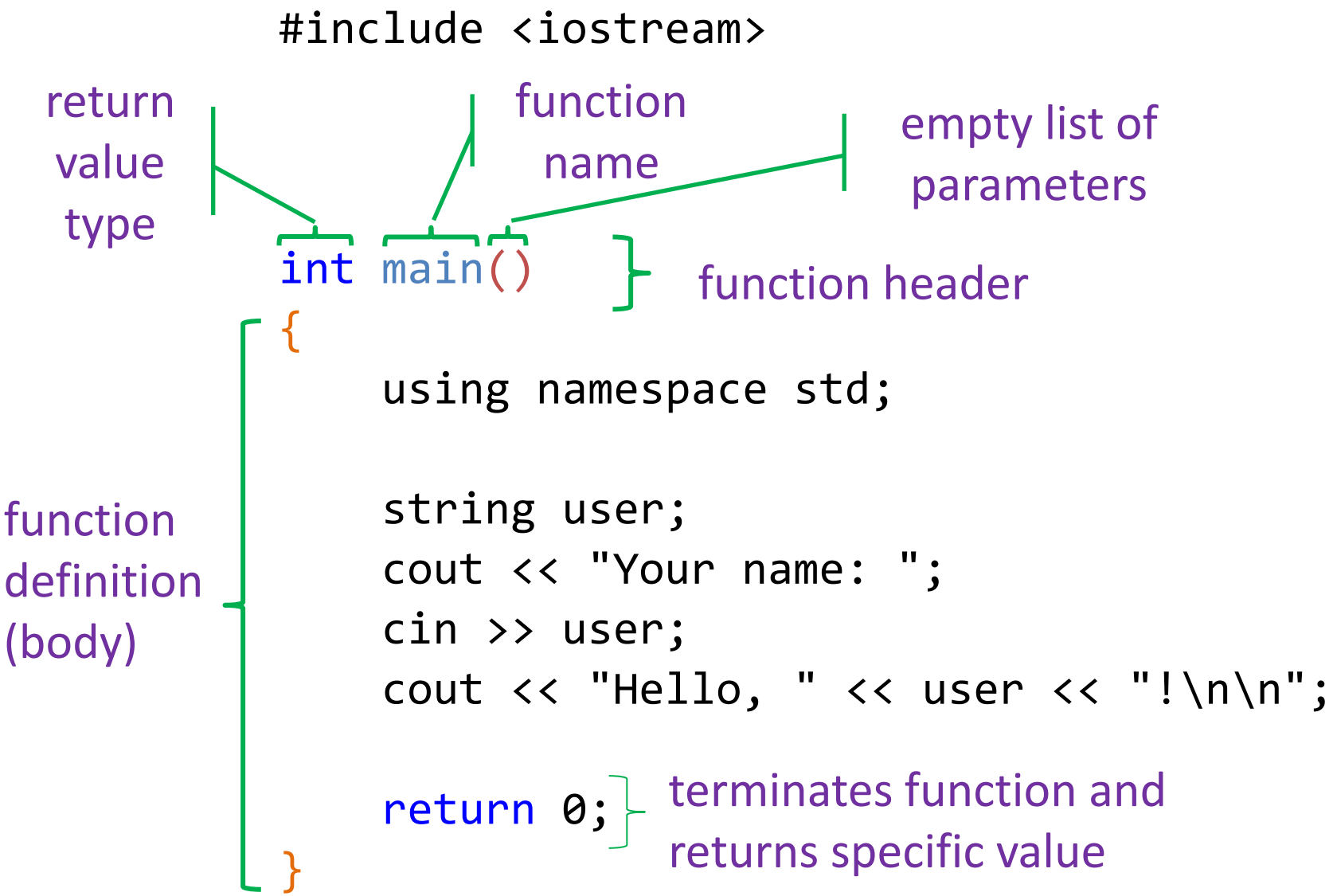
Block of Statements

```
#include <iostream>

int main()
{
    int a = 0;              // visible by the end
                            // of the function

    {
        int b = 42;         // visible only by the end
                            // of the current block
        int c = 13;         // visible only by the end
                            // of the current block
    }

    std::cout << a;         // 0
    // std::cout << b;      // ERROR: b does not exist
                            // anymore
}
```

main() scope

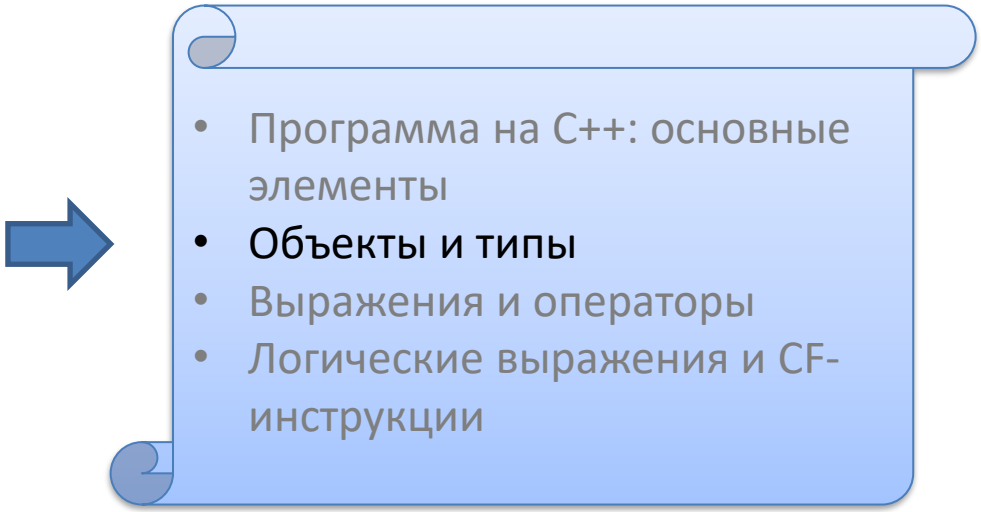
i.b. scope

Функция — как блок инструкций



А функция сама по себе (вместе с телом) — это инструкция или нет?

Функция — это *конструкция*!



В мире C++ есть два ключевых начала…

ОБЪЕКТЫ И ТИПЫ

Объект и Тип

- *Объект* — *типизированный* фрагмент памяти для хранения данных.
  Характеристики объекта:
  - *тип данных* объекта; в C++ объект без типа или *интерпретации* типа не имеет;
  - *имя* (одно, несколько или ни одного); как правило, манипулировать объектом можно через имя;
    - как создать новое имя для существующего объекта? — см. *ссылки* `&`
  - уникальный признак *сущности* (*entity*) — что это?
    - ***адрес*** в оперативной памяти (32- или 64-битное целое число); как правило, манипулировать объектом можно через адрес;
    - как получить адрес по имени? — оператор `&`
  - *значение* объекта:
    - объект-*переменная* — можно изменять значение объекта в процессе исполнения программы
    - *константный* объект — значение задается при инициализации и затем не меняется
    - *литерал* — константный объект, значение которого совпадает с его именем

`a`

`42`

- *Тип* — основная характеристика *объекта*, которая определяет:
  - *размер* объекта (в байтах);
  - *структуру* объекта — способ кодирования в оперативной памяти;
  - операционную *семантику* — множество допустимых *операций*, применимых к некоторому объекту
  - значение по умолчанию

`int`

Объект и Тип

- *Объект — типизированный* фрагмент памяти для хранения данных. Характеристики объекта:
  - *тип данных* объекта; в C++ объект без типа или *интерпретации* типа не имеет;
  - *имя* (одно, несколько или ни одного); как правило, манипулировать объектом можно через имя;
    - как создать новое имя для существующего объекта? — см. *ссылки* &
  - уникальный признак *сущности* (*entity*) — что это?
    - ***адрес*** в оперативной памяти (32- или 64-битное целое число); как правило, манипулировать объектом можно через адрес;
    - как получить адрес по имени? — оператор &
  - *значение* объекта:
    - объект-*переменная* — можно изменять значение объекта в процессе программы
    - *константный объект* — значение задается при инициализации и затем не меняется

А почему амперсандики разноцветные?

Важная договоренность:

- зелёным фломастером мы будем обозначать *модификаторы типов*:
  * — указатель, & — ссылка, [ ] — массив…
- красным фломастером мы будем обозначать *операторы*:
  * — разыменование, & — взятие адреса, [ ] — обращение к элементу массива с разыменованием…

Домашнее задание

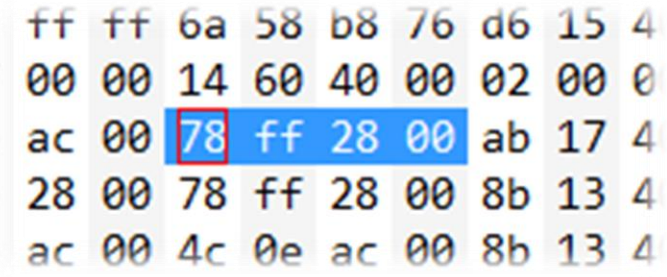
- Научиться вводить с клавиатуры:
  - зеленые звездочки * и красные звездочки *
  - зеленые амперсанды & и красные амперсанды &
  - …

Важная договоренность:

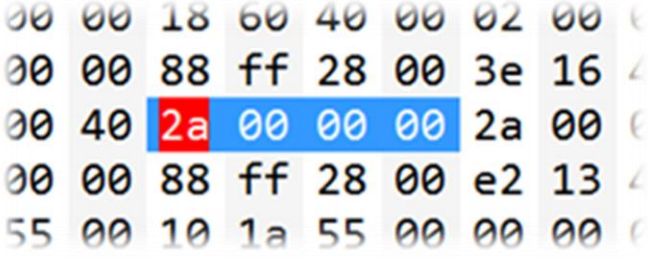
- зелёным фломастером мы будем обозначать *модификаторы типов*:
  * — указатель, & — ссылка, [ ] — массив…
- красным фломастером мы будем обозначать *операторы*:
  * — разыменование, & — взятие адреса, [ ] — обращение к элементу массива с разыменованием…

Объект и Тип: пример

```
int a;
```




```
int b = 42;
```




```
b = b + 1;                  // b == 43
a = b * 10;                 // a == 430
cout << a << ", " << b;     // 430, 43
```

Абстрактный Тип Данных (ADT)

- *Тип* — основная характеристика *объекта*:
  - *размер* объекта (в байтах);
  - *структура* объекта — способ кодирования в оперативной памяти;
  - операционная *семантика* — множество допустимых *операций*, применимых к некоторому объекту
  - значение по умолчанию

Абстрактный Тип Данных (ADT)

- *Тип* — основная характеристика *объекта*:
  - ~~*размер* объекта (в байтах);~~
  - ~~*структура* объекта — способ кодирования в оперативной памяти;~~
  - операционная *семантика* — множество допустимых *операций*, применимых к некоторому объекту
  - ~~значение по умолчанию~~

- Примеры ADT (известные успевающим[*куда?*] студентам):
  - *стек*: определяется операциями…
  - *очередь*: определяется операциями…
  - …

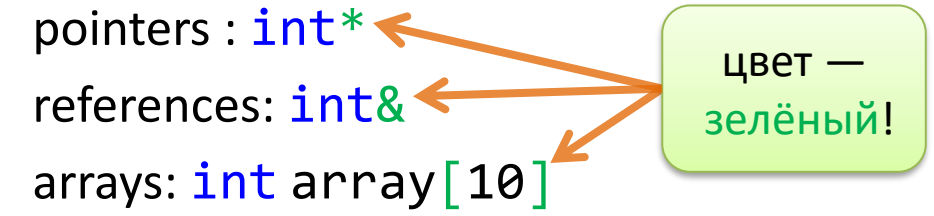
Типы C++

Фундаментальные типы

арифметические типы:
`int, double`
символьные типы: `char`
логический тип: `bool`
~~*псевдотип*~~: `void`

Производные (составные)

pointers : `int*`
references: `int&`
arrays: `int array[10]`



Пользовательские

`class`
`struct`
`enum`
функции

Функция — это тип или объект?

И то, и другое. Объявление (прототип) — про тип. Сама функция (тело) — это безусловно объект (*какими характеристиками обладает*?)

Type classification

The C++ type system consists of the following types:

- fundamental types (see also `std::is_fundamental`):
  - the type `void` (see also `std::is_void`);
  - the type `std::nullptr_t` (since C++11) (see also `std::is_null_pointer`);
  - arithmetic types (see also `std::is_arithmetic`):
    - floating-point types (`float`, `double`, `long double`) (see also `std::is_floating_point`);
    - integral types (see also `std::is_integral`):
      - the type `bool`;
      - character types:
        - narrow character types:
          - ordinary character types (`char`, `signed char`, `unsigned char`)
          - the type `char8_t`
        - wide character types (`char16_t`, `char32_t`, `wchar_t`);
      - signed integer types (`short int`, `int`, `long int`, `long long int`);
      - unsigned integer types (`unsigned short int`, `unsigned int`, `unsigned long int`, `unsigned long long int`);
  - compound types (see also `std::is_compound`):
    - reference types (see also `std::is_reference`):
      - lvalue reference types (see also `std::is_lvalue_reference`):
        - lvalue reference to object types;
        - lvalue reference to function types;
      - rvalue reference types (see also `std::is_rvalue_reference`):
        - rvalue reference to object types;
        - rvalue reference to function types;
    - pointer types (see also `std::is_pointer`):
      - pointer-to-object types;
      - pointer-to-function types;
    - pointer-to-member types (see also `std::is_member_pointer`):
      - pointer-to-data-member types (see also `std::is_member_object_pointer`);
      - pointer-to-member-function types (see also `std::is_member_function_pointer`);
    - array types (see also `std::is_array`);
    - function types (see also `std::is_function`);
    - enumeration types (see also `std::is_enum`);
    - class types;

Реальная жизнь, как обычно, немного сложнее…

https://en.cppreference.com/w/cpp/language/type.html

Integer Types

- Vary from system to system. The standard says:
  - A `short` integer is at least 16 bits wide.
  - An `int` integer is at least as big as short.
  - A `long` integer is at least 32 bits wide and at least as big as int.
  - A `long long` integer is at least 64 bits wide and at least as big as long.
  - `int16_t`, `int32_t`, `int64_t` are special aliases for types with guaranteed size.

```
#include <cstdint>
```

- Can investigate real size by:
  - using `sizeof` operator;
  - using `<climits>` header for limit constants;

HW: explore size of the following types for your individual platform and fill the table up

- Unsigned types have the same size as signed ones, but other ranges:
  - `unsigned short`
  - `unsigned int`
  - …

Type	Size (bytes)	Range
short		
int		
long		
long long		
unsigned short		
unsigned int		

Пределы числовых типов: C-подход <climits>

Symbolic Constant	Represents
CHAR_BIT	Number of bits in a `char`
CHAR_MAX	Maximum `char` value
CHAR_MIN	Minimum `char` value
SCHAR_MAX	Maximum `signed char` value
SCHAR_MIN	Minimum `signed char` value
UCHAR_MAX	Maximum `unsigned char` value
SHRT_MAX	Maximum `short` value
SHRT_MIN	Minimum `short` value
USHRT_MAX	Maximum `unsigned short` value
INT_MAX	Maximum `int` value
INT_MIN	Minimum `int` value
UINT_MAX	Maximum `unsigned int` value
LONG_MAX	Maximum `long` value
LONG_MIN	Minimum `long` value
ULONG_MAX	Maximum `unsigned long` value
LLONG_MAX	Maximum `long long` value
LLONG_MIN	Minimum `long long` value
ULLONG_MAX	Maximum `unsigned long long` value

num_limits problem, Lec3

Пределы числовых типов: C++-подход <limits>

```
template<> class numeric_limits<bool>;
template<> class numeric_limits<char>;
template<> class numeric_limits<signed char>;
template<> class numeric_limits<unsigned char>;
template<> class numeric_limits<wchar_t>;
template<> class numeric_limits<unsigned short>;
template<> class numeric_limits<int>;
template<> class numeric_limits<unsigned int>;
template<> class numeric_limits<long>;
template<> class numeric_limits<unsigned long>;
template<> class numeric_limits<long long>;          (since C++11)
template<> class numeric_limits<unsigned long long>; (since C++11)
template<> class numeric_limits<float>;
template<> class numeric_limits<double>;
template<> class numeric_limits<long double>;
```

```
std::cout << "Limits defined by numeric_limits<> type:\n";
std::cout << "  int MIN value is " << std::numeric_limits<int>::min()
          << ", and its MAX value is "
          << std::numeric_limits<int>::max() << "\n";
```

Member constants

is_specialized [static]

is_signed [static]

has_infinity [static]

has_quiet_NaN [static]

Member functions

min [static] return norma (public

lowest [static] (C++11) return 0 fo (public

max [static] return (public

Подход основан на частично определенных шаблонах. Разберемся с этим позже.

num_limits problem, Lec3

https://en.cppreference.com/w/cpp/types/numeric_limits.html

Целые типы фиксированного размера <cstdint>

- Для целых типов char, short, int, long и др. стандарт не определяет их точного размера в байтах, вместо этого даются некоторые гарантированные отношения вида:
  - char < short <= int <= long…
- Иногда необходимо зафиксировать точный размер: 8 бит, 16 бит, 4 байта…

defined in header <cstdint>

int8_t **int16_t** (optional) **int32_t** **int64_t**	signed integer type with width of exactly 8, 16, 32 and 64 bits respectively with no padding bits and using 2's complement for negative values (provided if and only if the implementation directly supports the type) (typedef)
int_fast8_t **int_fast16_t** **int_fast32_t** **int_fast64_t**	fastest signed integer type with width of at least 8, 16, 32 and 64 bits respectively (typedef)
int_least8_t **int_least16_t** **int_least32_t** **int_least64_t**	smallest signed integer type with width of at least 8, 16, 32 and 64 bits respectively (typedef)
intmax_t	maximum-width signed integer type (typedef)
intptr_t (optional)	signed integer type capable of holding a pointer to void (typedef)
uint8_t **uint16_t** (optional) **uint32_t** **uint64_t**	unsigned integer type with width of exactly 8, 16, 32 and 64 bits respectively (provided if and only if the implementation directly supports the type) (typedef)
uint_fast8_t	

Initialization of Numbers

```
int nInt = INT_MAX;

int apples = 3;       // initializes uncles to 3
int pears = apples;   // initializes pears to 3
int peaches = apples + pears + 6; // initializes peaches to 10

int dogs = 101;       // traditional C initialization, sets dogs to 101
int cats(667);        // alternative C++ syntax, sets cats to 667

int boys{9};
int girls = {10};
```

C++11

```
int a = 42;           // decimal integer literal
int b = 0x42;         // hexadecimal integer literal
int c = 042;          // octal integer literal

double pi = 3.1415925;          // decimal point form
double avogadro = 6.022e+23;    // exponential form 6.022 * 10^23
```

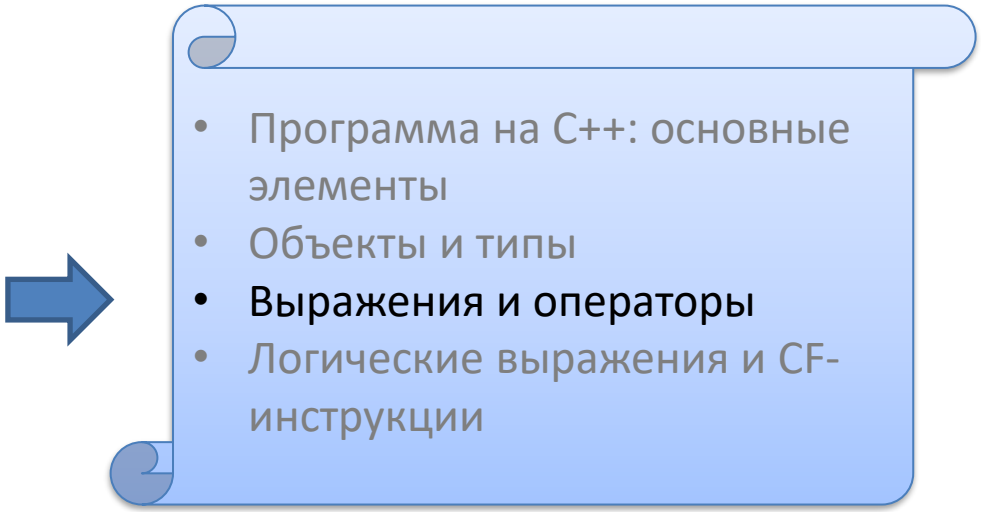
Other Primitive Types

- `char` is for 8-bit small integer or a 1-byte character
  - can be signed
- `bool` is for boolean type:
  - `true` and `false` constants;
- `double` is for floating-point numbers:
  - at least 48 bits (generally, 8 bytes) for representation;
  - do not use `float` instead, never!

Для справки…

Plain Old Datatype (POD; старые-добрые тёплые-ламповые типы данных) — скаляры или старомодные структуры (из C), у которых *нет* (?) *конструкторов*, *базовых классов*, *закрытых данных*, *виртуальных функций* и проч.;

POD — что-то такое, что «безопасно» копировать побитно, «как есть»



«Мясо» любой программы на любом языке…

ВЫРАЖЕНИЯ И ОПЕРАТОРЫ

Expressions

- *Expression* is a valid C++-sentence containing operands and operators;
  - *operands* are objects: variables, constants, literals;
  - *operators* are represented by single characters (+, –, *, /, %), double characters (++, --, ==, !=, ? :) or even by keywords (sizeof, new, delete , …)
  - an individual operator and its operands form a subexpression;
  - an expression has a type inferred from types of individual operands;
  - the expression is evaluated by putting specific values for all operands;

```
2 + 3 * 2              // arithmetic expression
a = a * sqrt(4)        // expression calling a function
2. == sqrt(4)          // logical expression
```

Operators

Operator is a special symbol (pair of symbols or a keyword), which performs an operation on its operands.

- Operators have *arity*:
  - *unary* (`!`, `-`, `~`, `&`, `*`, …),
  - *binary* (`+`, `-`, `++`, `!=`, `==`, `||`, `+=`, …),
  - *ternary* (`?:`)

Здесь мы, конечно, уже используем рыжий цвет для обозначения операторов, но это то же самое, что и красный, просто выцветший под сиянием ~~теслонда~~ солнца

- Order of evaluation in an expression is determined by the operators' precedence:

```
2 + 3 * 4     // 14
(2 + 3) * 4   // 20
```

Operators

Basic arithmetic:

- The `+` operator adds its operands.
- The `-` operator subtracts the second operand from the first.
- The `*` operator multiplies its operands.
- The `/` operator divides its first operand by the second.
- The `%` operator finds the modulus of its first operand with respect to the second.

Compound assignment:

- `+=`, `-=`, `*=`, `/=`, `%=`

Increment and decrement (*prefix* and *suffix* versions):

- `i++` , `++i`, `i--`, `--i`

Precedence of Operators

Level	Precedence group	Operator	Description	Grouping
1	Scope	`::`	scope qualifier	Left-to-right
2	Postfix (unary)	`++ --`	postfix increment / decrement	Left-to-right
		`()`	functional forms	
		`[]`	subscript	
		`. ->`	member access	
3	Prefix (unary)	`++ --`	prefix increment / decrement	Right-to-left
		`~ !`	bitwise NOT / logical NOT	
		`+ -`	unary prefix	
		`& *`	reference / dereference	
		`new delete`	allocation / deallocation	
		`sizeof`	parameter pack	
		`(type)`	C-style type-casting	
4	Pointer-to-member	`.* ->*`	access pointer	Left-to-right
5	Arithmetic: scaling	`* / %`	multiply, divide, modulo	Left-to-right
6	Arithmetic: addition	`+ -`	addition, subtraction	Left-to-right
7	Bitwise shift	`<< >>`	shift left, shift right	Left-to-right
8	Relational	`< > <= >=`	comparison operators	Left-to-right
9	Equality	`== !=`	equality / inequality	Left-to-right
10	And	`&`	bitwise AND	Left-to-right
11	Exclusive or	`^`	bitwise XOR	Left-to-right
12	Inclusive or	`|`	bitwise OR	Left-to-right
13	Conjunction	`&&`	logical AND	Left-to-right
14	Disjunction	`||`	logical OR	Left-to-right
15	Assignment-level expressions	`= *= /= %= += -= >>= <<= &= ^= |=`	assignment / compound assignment	Right-to-left
		`?:`	conditional operator	
16	Sequencing	`,`	comma separator	Left-to-right

http://www.cplusplus.com/doc/tutorial/operators/

Дерево разбора

```
-2 + 3 * (18 / sqrt(4) * (2 + pow(4, 2 + 1)))
```

Задание на дом (HW): нарисовать дерево разбора для этого выражения.

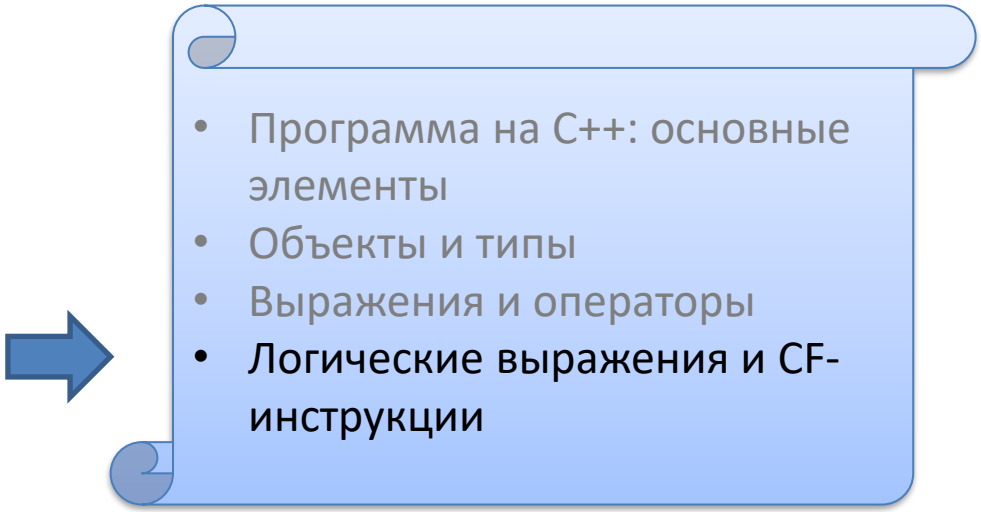
Инструкции: *сорта и специи*

- *Прототип функции* — инструкция *описания* (*definition*) контракта функции
  - `double foo(double x, int p);`

- Инструкция *определения* и *описания* объекта
  - `int a;`
  - разновидность: *с инициализацией*: `int a = 42;`

- Инструкция *(только)* описания объекта:
  - `extern int a;`

- Инструкция-выражение:
  - `y = x + 1;`

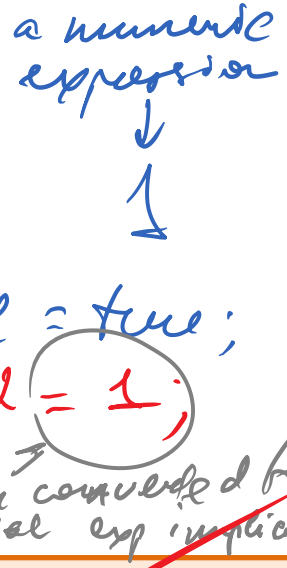


Для CF statements нужны

ЛОГИЧЕСКИЕ ВЫРАЖЕНИЯ И CF-ИНСТРУКЦИИ

Logical Expressions

- A *logical expression* is an expression evaluated as the boolean type
  - contains logical operators;
  - types other than `boolean` are implicitly converted to `bool`:
    - numbers: 0 → `false`, otherwise `true`
    - pointers: `nullptr` → `false`, otherwise `true`
    - …
- Predicates:
  - (in)equality: `==`, `!=`
  - comparison `<`, `<=`, `>`, `>=`
- Logical Operators:
  - `||` is for logical *OR*
  - `&&` is for logical *AND*
  - `!` is for logical *NOT*




```
%:include <iostream>

int main(int argc, char *argv<::>)
<%
    if (argc > 1 and argv<:1:> not_eq '\0') <%
        std::cout << "Hello " << argv<:1:> << '\n';
    %>
%>
```

The Comparison Operators: ==, !=, <, <=, >, >=

```
bool l1 = (2 == 2);
bool l2 = (2. == 2);
bool l3 = ("2" == 2);    //
bool l4 = (2. != 22);
bool l5 = (18 < 42);

bool l6 = ("Abc" < "abc");   //

string s1("Abc"), s2("abc");
bool l7 = s1 < s2;          // strings are compared lexicographically
```

Logical Functions (some examples)

```
bool l10 = isdigit('A'); ✗
bool l11 = isdigit('1'); ✓
bool l12 = isalpha('A'); ✓
bool l13 = isalpha('1'); ✗

string s1("Abc");
bool l14 = s1.empty(); ✗
```

// empty(s1) ?

The Logical Operators: &&, ||, !

x	y	x && y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x || y
0	0	0
0	1	1
1	0	1
1	1	1

x	!x
0	1
0	1
1	0
1	0

&& — logical *AND*

|| — logical *OR*

! — logical *NOT*

The Logical Operators: &&, ||, !

- The Logical *OR* Operator: ||

```
4 == 4 || 4 == 11
4 > 3  || 4 > 10
4 > 9  || 4 < 10
4 < 9  || 4 > 2
4 > 9  || 4 < 2
```

$3 < x < 5$

- Combination of operators:

```
x > 3 && x < 5 || y > 10

(x > 5 || y > 3) && z > 10

x != 0 && 1.0 / x > 100.0
```

- The Logical *AND* Operator: &&

```
6 == 6 && 3 == 3
6 == 2 && 3 == 3
6 > 2  && 6 > 10
6 > 8  && 6 < 10
6 < 8  && 6 > 2
6 > 8  && 6 < 2
```

- The Logical *NOT* Operator: !

```
!(x > 6)  => (x <= 6)
!(x > 5)     !x > 5
```

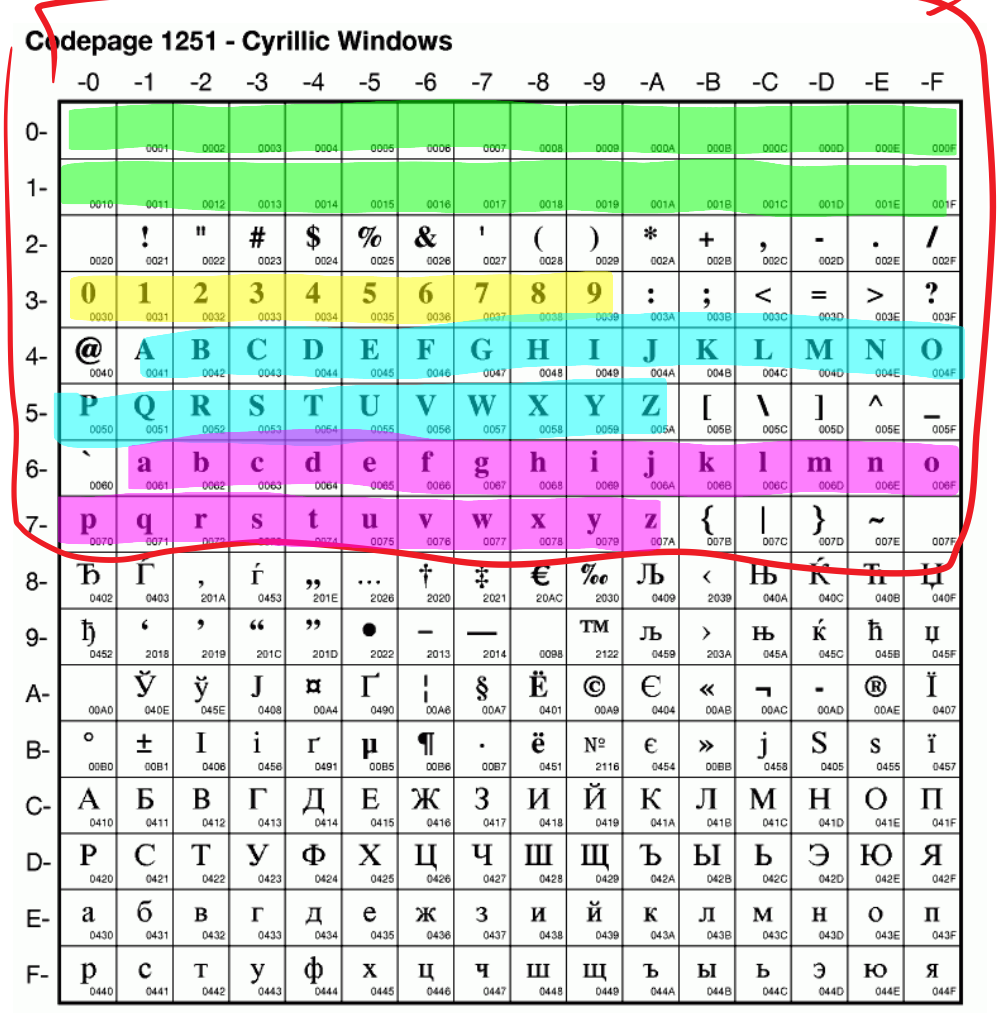
- De Morgan's law:

$!(x \,||\, y) == !x \,\&\&\, !y$

$!(x \,\&\&\, y) == !x \,||\, !y$

Setting Up Ranges with Logical Operators

```
char ch = …

if(ch < 32) …

if(ch >= '0' && ch <= '9')…

if(ch >= 'A' && ch <= 'Z'
    ||
   ch >= 'a' && ch <= 'z') …

if(!(ch >= '!' && ch <= '/'))
```



CF: Ветвление

- *CF* — **C**ontrol-Flow statements (инструкции *управления потоком исполнения программы*):
  - условие `if`
    - `if` с инициализацией
  - условие с альтернативой: `if..else`
  - многовариантное условие `switch`
    - `switch` с инициализацией

C++17

C++17

Notes on Using `else` Clause

Try to omit using else clause whenever possible!

```
if(x1 == x2 || y1 == y2)
{
    cout << "YES";
}
else if(abs(x) == abs(y))
{
    cout << "YES";
}
else
{
    cout << "NO";
}

return 0;
```

Better!

```
if(x1 == x2 || y1 == y2)
{
    cout << "YES";
    return 0;
}

if(abs(x) == abs(y))
{
    cout << "YES";
    return 0;
}

cout << "NO";

return 0;
```

Notes on Using `else` Clause

Use inversion of logic to keep the conditions of both branches close to each other.

```
if(x > 0)
{
    cout << "Lorem ipsum dolor sit"
 "amet, consectetur adipiscing"
 "elit, sed do eiusmod tempor"
 "incididunt ut labore et dolore"
 "magna aliqua. Ut enim ad minim"
 "veniam, quis nostrud exercitation"
 "ullamco laboris nisi ut aliquip"
 "ex ea commodo consequat. Duis"

 "in culp                    deserunt"
 "mollit ani                 m.";
}
else
{
    cout << "Hello World";
}
```

5 pages of code!

Better!

```
if(x <= 0)  // !(x > 0)
{
    cout << "Hello World";
}
else
{
    cout << "Lorem ipsum dolor sit"
 "amet, consectetur adipiscing"
 "elit, sed do eiusmod tempor"
 "incididunt ut labore et dolore"
 "magna aliqua. Ut enim ad minim"
 "veniam, quis nostrud exercitation"
 "ullamco laboris nisi ut aliquip"
 "ex ea commodo consequat. Duis"

 "in culpa qui officia deserunt"
 "mollit anim id est laborum.";
}
```

The ?: Operator

- Serves as a substitution for `if..else` statement when the only need is to have a different expression in one place
  - `expression1 ? expression2 : expression3`

```
int x;
cout << "Input x = ";
cin >> x;
cout << "Abs(x) is " << (x > 0 ? x : -x);
```

- Is this valid? —

```
cout << " 100 / x = "
     << (x != 0 ? 100 / x : "x can't be 0!");
```

The `if..else if..else` Construction

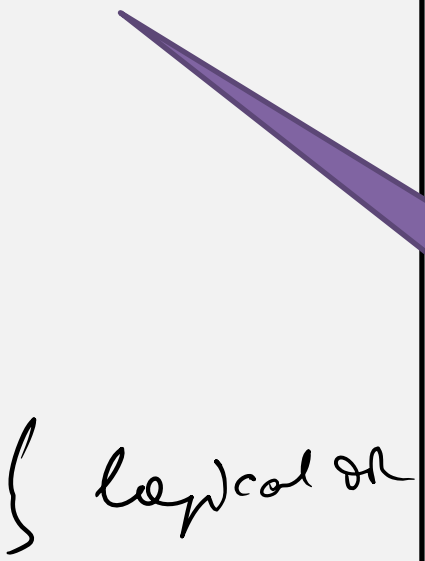
- C++ doesn't have a dedicated elseif clause but it can be implemented by using secondary `if..else` statement as a statement of `else` branch of the first `if..else` statement:

```
if(symb >= '0' && symb <= '9')
    cout << "Digit\n";
else if(symb >= 'A' && symb <= 'Z')
    cout << "Capital Latin Letter\n";
else if(symb >= 'a' && symb <= 'z')
    cout << "Small Latin Letter\n";
else
    cout << "Something else\n";
```

How to choose between the `switch` statement and the `if.. else if.. else` construction

```
if(symb >= '0' && symb <= '9')
    cout << "Digit\n";

else if(symb >= 'A' && symb <= 'Z')
    cout << "Capital Latin Letter\n";

else if(symb >= 'a' && symb <= 'z')
    cout << "Small Latin Letter\n";

else
    cout << "Something else\n";
```

```
unsigned short day;
cout << "Input day num (1..7): ";
cin >> day;
cout << "Day is ";

switch(day) {
case 1:
    cout << "Monday";
    break;
case 2:
    cout << "Tuesday";
    break;
// ...
case 7:
    cout << "Sunday";
    break;
case 8:
case 9:
case 10:
    cout << "Additional Day of a Week :)";
    break;

default:
    cout << "Wrong day number";
}
```



Инструкции: *сорта и специи*

- *Прототип функции* — инструкция *описания* (*definition*) контракта функции
  - `double foo(double x, int p);`
- Инструкция *определения* и *описания* объекта
  - `int a;`
  - разновидность: *с инициализацией*: `int a = 42;`
- Инструкция *(только)* описания объекта:
  - `extern int a;`
- Инструкция-выражение:
  - `y = x + 1;`
- CF/ветвления: `if`, `if..else`, `switch`

CF: Циклы

- Конструирование циклов:
  - нерегулярный цикл с пред/усл. `while` (инструкция)
  - регулярный цикл с пред/усл. `for` (инструкция)
  - регулярный цикл перебора элементов коллекции range-based `for` (инструкция)
  - нерегулярный цикл с пост/усл. `do..while` (инструкция)

C++11

- Управление поведением циклов:
  - инструкция `break`;
  - инструкция `continue`;

В западной традиции CS это называют *петлями* (*loops*), а понятие *цикл* (*cycle*) чаще встречается в теории графов

Цикл Range-Based `for`

- Итерирует по коллекции элементов от первого до последнего
- Может модифицировать коллекцию, если делать привязку по ссылке (поговорим об этом подробнее позже)

```
double koefs[] = {1.12, 2.13, 3.14, 4.15, 5.16};

for (double x : koefs)
    cout << x << std::endl;



for (int x : {1, 1, 2, 3, 5})
    cout << x << " ";
```

Инструкции: *сорта и специи*

- *Прототип функции* — инструкция *описания* (*definition*) контракта функции
  - `double foo(double x, int p);`
- Инструкция *определения* и *описания* объекта
  - `int a;`
  - разновидность: *с инициализацией*: `int a = 42;`
- Инструкция *(только)* описания объекта:
  - `extern int a;`
- Инструкция-выражение:
  - `y = x + 1;`
- CF/ветвления: `if`, `if..else`, `switch`
- CF/циклы: `while`, `for`, RB- `for`, `do..while`; `break`; `continue`;

The `return` Statement

The **return** statement is a beautiful way to break a logical condition or a loop and leave the function immediately.

The best!!

```
if(x <= 0)  // !(x > 0)
{
    cout << "Hello World";
    return;
}

cout << "Lorem ipsum dolor sit"
  "amet, consectetur adipiscing"
  "elit, sed do eiusmod tempor"
  "incididunt ut labore et dolore"
  "magna aliqua. Ut enim ad minim"
  "veniam, quis nostrud exercitation"
  "ullamco laboris nisi ut aliquip"
  "ex ea commodo consequat. Duis"
  "in culpa qui officia deserunt"
  "mollit anim id est laborum.";
```

Better!

```
if(x <= 0)  // !(x > 0)
{
    cout << "Hello World";
}
else
{
    cout << "Lorem ipsum dolor sit"
 "amet, consectetur adipiscing"
 "elit, sed do eiusmod tempor"
 "incididunt ut labore et dolore"
 "magna aliqua. Ut enim ad minim"
 "veniam, quis nostrud exercitation"
 "ullamco laboris nisi ut aliquip"
 "ex ea commodo consequat. Duis"
 "in culpa qui officia deserunt"
 "mollit anim id est laborum.";
}
```

Инструкции: *сорта и специи*

- *Прототип функции* — инструкция *описания* (*definition*) контракта функции
  - `double foo(double x, int p);`
- Инструкция *определения* и *описания* объекта
  - `int a;`
  - разновидность: *с инициализацией*: `int a = 42;`
- Инструкция *(только)* описания объекта:
  - `extern int a;`
- Инструкция-выражение:
  - `y = x + 1;`
- CF/ветвления: `if`, `if..else`, `switch`
- CF/циклы: `while`, `for`, RB- `for`, `do..while`; `break`; `continue`;
- CF*/`return`

To be continued…


2 BP
QUIZ
TIME
5min

Вопрос на бонусный балл

На лекции мы рассматривали различные сорта (разновидности) инструкций. Приведите столько примеров различных сортов (при необходимости пояснения — с примерами) **инструкций**, сколько сможете: из лекции, или из С-шной *прошлого*, или из С++-го *настоящего*.

Будьте осторожны: каждая запись, не являющаяся настоящей инструкцией (пусть и корректная с т.з. С++), будет уменьшать общее число правильных ответов.

Авторы наибольшего числа правильных ответов получает пару… бонусов!

 **5 мин**

Спасибо за внимание!