HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

Introduction to Programming

C++ Basics

Sergey Shershakov

#2/15 Jan 2019

C++ Alphabet

- Keywords and identifiers:
  - English letters: A..Z, a..z (beware of Russian “С”)
  - 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

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.

```
int main()
{
    string name;
    cout << "Your name: ";
    cin >> name;
    …
```

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_stars3;  // valid
int _Mystars3;  // valid but reserved – starts with underscore
int 4ever;      // invalid because starts with a digit
int double;     // invalid – double is a C++ keyword
int begin;      // valid – begin is a Pascal keyword
int __fools;    // valid but reserved – starts with two underscores
int the_very_best_variable_i_can_be_version_112; // valid
int honky-tonk; // invalid – no hyphens allowed
```

Excerpts from Naming Conventions

- Use *PascalStyle* for naming types:

```
class MyNewClass {…}
```

- Use *camelStyle* for naming local objects:

```
int varName;

void funcName(int param)
{ …
}
```

- Start an object name with an underscore for private and protected members of a class:

```
class Foo
{
private:
    int _privNumber;
}
```

- Use *CAPITALS_WITH_UNDERSCORES* for naming constants:

```
const int PI_NUMBER = 3.1415926;
```

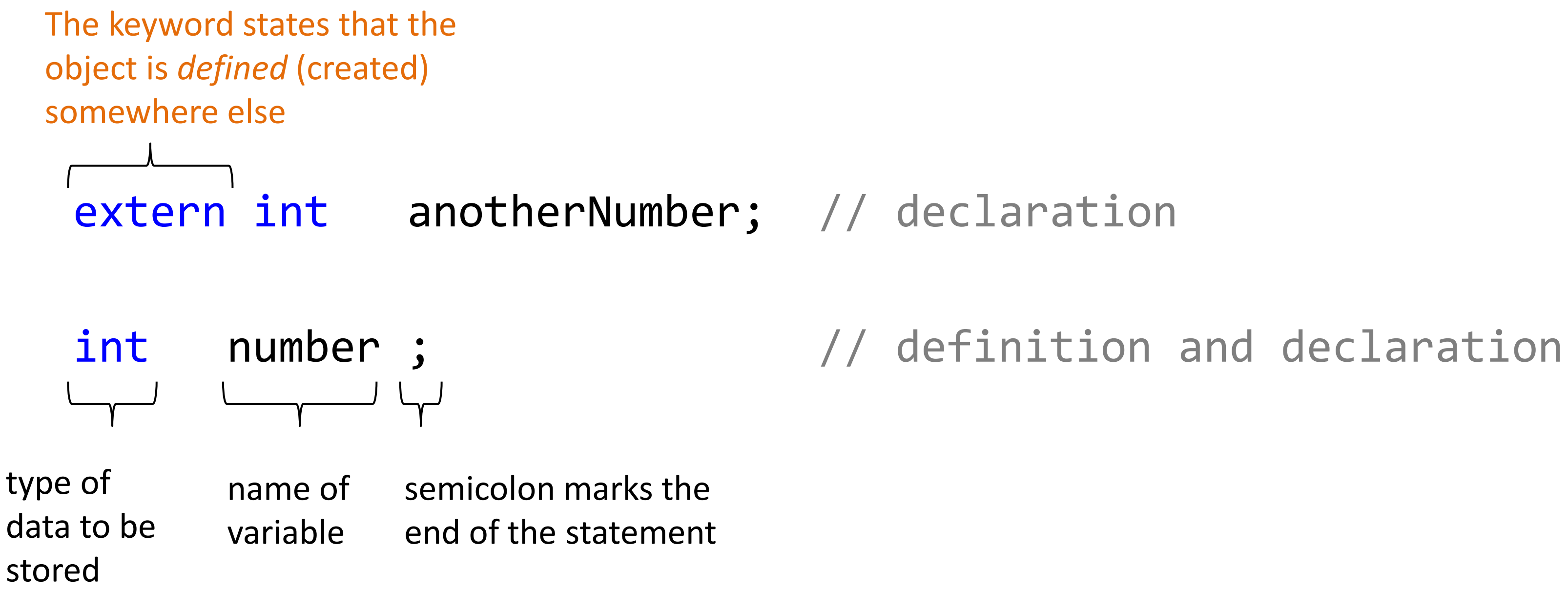
Statements

- *Statement* is a sentence ending with a semicolon or a set of sentences enclosed in curly brackets { }.

```
#include <iostream>

int main()
{
    using namespace std;       // using statement

    string name;               // definition and declaration statement
    cout << "Your name: ";     // expression statement
    cin >> name;               // expression statement

    name = "Rostislav";        // assignment statement

    cout << "Hello, " << name
         << "!\n\n";           // expression (complex)

    return 0;                  // return statement
}
```

Definition and Declaration Statements

- *Definition* is the creation of an object.
- *Declaration* is the designation (unleashing) of an object in the current scope.



The keyword states that the object is *defined* (created) somewhere else

```
extern int   anotherNumber;  // declaration

int    number ;              // definition and declaration
```

type of data to be stored

name of variable

semicolon marks the end of the statement

Initialization and Assignment Statements

- *Initialization* is putting a default value to an object when creating (copy constructor works):

```
    int num = 42;
```

- *Assignment* is rewriting a current value of an object to another value (copy operator works):

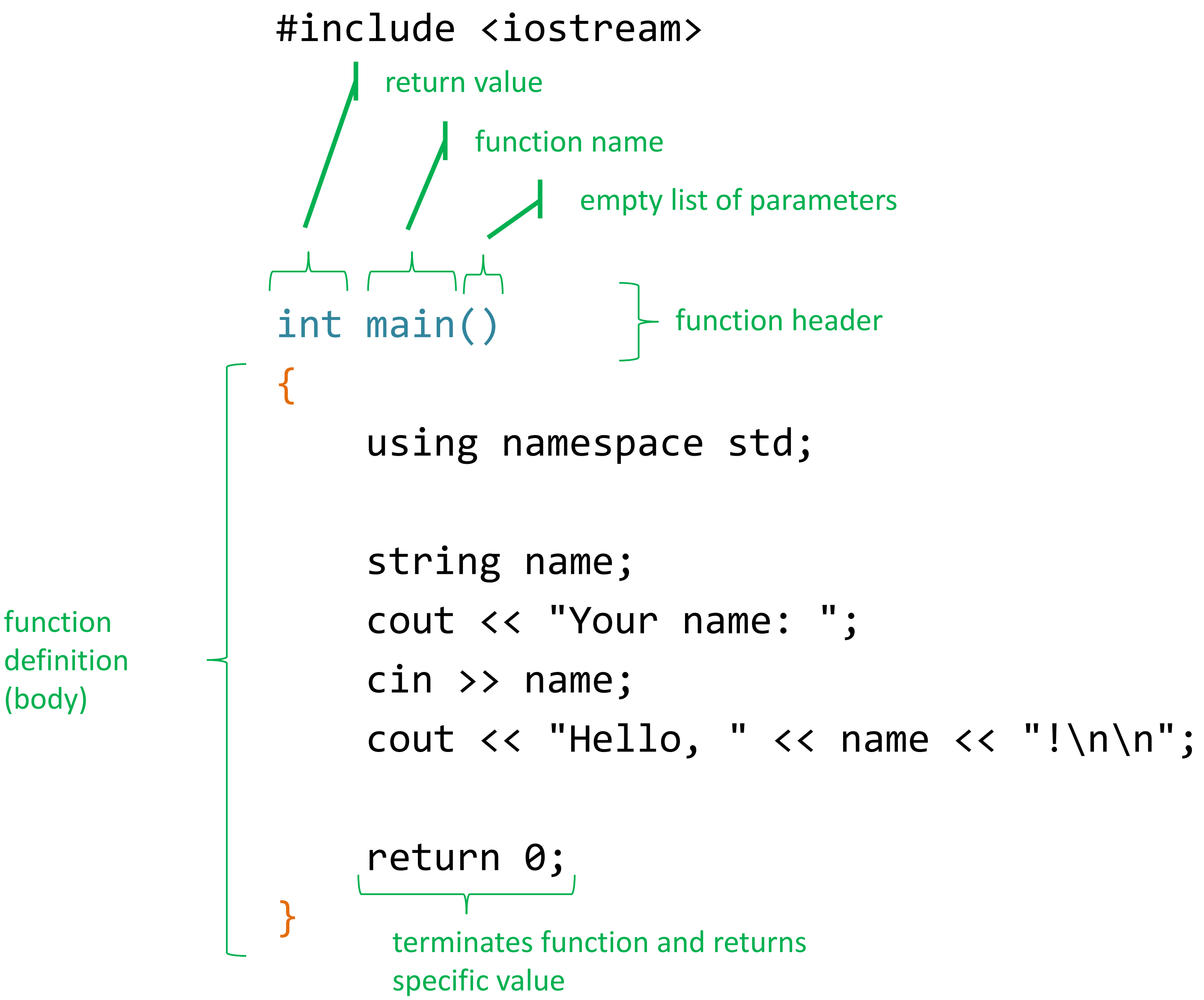
```
    num = 13;
```

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

```
#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
}
```

Function as a Block of Statements



```
#include <iostream>

int main()
{
    using namespace std;

    string name;
    cout << "Your name: ";
    cin >> name;
    cout << "Hello, " << name << "!\n\n";

    return 0;
}
```

return value

function name

empty list of parameters

function header

function definition (body)

terminates function and returns specific value

OBJECTS AND TYPES

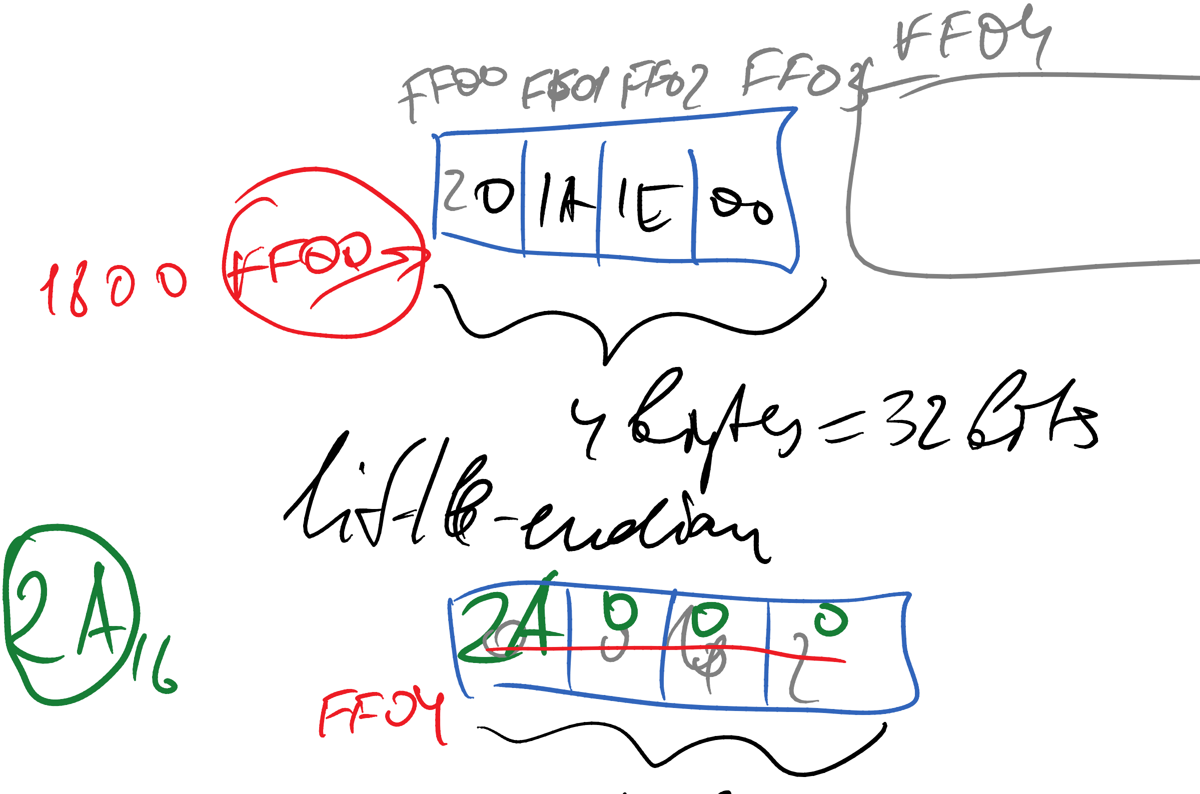
Object and Type

- *Object* is a *typed* piece of memory for storing data:
  - can have a *name* (not mandatory);
  - has an *address* in memory; can be manipulated by using the address;
  - *variable* is an object that can change its value during a program execution;

- *Type* is the main characteristic of an *object*:
  - determines the size of an object;
  - determines an object’s structure;
  - determines all possible operations that are applicable for the object (its semantics).

Object and Type: Example

```
int a;
```



```
int b = 42;
```
$= \boxed{2A}_{16}$

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

Types in C++

Fundamental types

arithmetic types: `int`, `double`, `char`

`bool`

`void`

Compound types

pointers `int*`

references `int&`

arrays `int array[10]`

`class struct`

`enum`

function

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.

Explore for your own platform!

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

- 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`		

Symbolic Constants from <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

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 = O42;         // octal integer literal
```

Other Primitive Types

- *Plain Old Datatype (POD)* is a scalar type or an old-fashioned structure with no *constructors*, *base classes*, *private data*, *virtual functions*, and so on;
  - POD is something for which it's safe to make a byte-by-byte copy

- `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!

EXPRESSIONS AND OPERATORS

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            // arithmetical 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:
  - has *arity*: *unary* (`!`, `-`, `~`, `&`, `*`, …), *binary* (`+`, `-`, `++`, `!=`, `==`, `||`, `+=`, …) and one *ternary* (`? :`)
  - order of evaluation in an expression is determined by the operators' precedence:

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

- Arithmetic operators:
  - 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.

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/]

Expression Tree

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

Try to build it yourself!

https://goo.gl/forms/6VDVnnH12S8778pI3