

CSC 211: Computer Programming

Basic C++ Concepts and Syntax

Dr. Michael Conti

Department of Computer Science and Statistics
University of Rhode Island

Spring 2026



C++ Basics

Basics

- Everything in C++ is **case sensitive**
- Curly braces are used to denote **code blocks**

```
int main() {  
    // body ...  
}
```

- All statements end with a **semicolon** (can use multiple lines)

```
int a;  
a = 100;  
a = a + 111;  
  
int a;  
a = 100;  
a = a  
+  
111;
```

The main function

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

```
int main (int argc, char *argv[]) {  
    // body  
    return 0;  
}
```

The `main` function

- Automatically called at program startup
 - designated entry point to a program that is executed in a hosted environment (operating system)
- Prototype cannot be modified
- Cannot be used anywhere in the program
 - cannot be overloaded
 - cannot be called recursively
- Its address cannot be taken

https://en.cppreference.com/w/cpp/language/main_function

5

The `main` function

- Does not need to contain the `return` statement
 - if control reaches the end of `main` without encountering a `return` statement, the effect is that of executing `return 0;`
- Execution of the `return` (or the implicit `return`) is equivalent to:
 - leaving the function normally (which destroys local objects)
 - calling `std::exit` with the same argument as the argument of the `return`
 - `std::exit` destroys static objects and terminates the program

https://en.cppreference.com/w/cpp/language/main_function

6

Comments

- **Comments** can be single-line or multi-line
 - comments are ignored by the compiler

```
int a;  
// ignore the following line  
// a = 100;  
a = 200;
```

```
int a;  
// ignore this block  
a = 100;  
/*  
a = a  
+  
111;  
*/
```

7

C++ keywords

This is a list of reserved keywords in C++. Since they are used by the language, these keywords are not available for re-definition or overloading.

| | | |
|--------------------------------------|---------------------------------------|--|
| <code>alignas</code> (since C++11) | <code>default</code> (1) | <code>register</code> (2) |
| <code>alignof</code> (since C++11) | <code>delete</code> (1) | <code>reinterpret_cast</code> |
| <code>and</code> | <code>do</code> | <code>requires</code> (since C++20) |
| <code>and_eq</code> | <code>double</code> | <code>return</code> |
| <code>asm</code> | <code>dynamic_cast</code> | <code>short</code> |
| <code>atomic_cancel</code> (TM TS) | <code>else</code> | <code>signed</code> |
| <code>atomic_commit</code> (TM TS) | <code>enum</code> | <code>sizeof</code> (1) |
| <code>atomic_noexcept</code> (TM TS) | <code>explicit</code> | <code>static</code> |
| <code>auto</code> (1) | <code>export</code> (1)(3) | <code>static_assert</code> (since C++11) |
| <code>bitand</code> | <code>extern</code> (1) | <code>static_cast</code> |
| <code>bitor</code> | <code>false</code> | <code>struct</code> (1) |
| <code>bool</code> | <code>float</code> | <code>switch</code> |
| <code>break</code> | <code>for</code> | <code>synchronized</code> (TM TS) |
| <code>case</code> | <code>friend</code> | <code>template</code> |
| <code>catch</code> | <code>goto</code> | <code>this</code> |
| <code>char</code> | <code>if</code> | <code>thread_local</code> (since C++11) |
| <code>char8_t</code> (since C++20) | <code>inline</code> (1) | <code>throw</code> |
| <code>char16_t</code> (since C++11) | <code>int</code> | <code>true</code> |
| <code>char32_t</code> (since C++11) | <code>long</code> | <code>try</code> |
| <code>class</code> (1) | <code>mutable</code> (1) | <code>typedef</code> |
| <code>compl</code> | <code>namespace</code> | <code>typeid</code> |
| <code>concept</code> (since C++20) | <code>new</code> | <code>typename</code> |
| <code>const</code> | <code>noexcept</code> (since C++11) | <code>union</code> |
| <code>constexpr</code> (since C++11) | <code>not</code> | <code>unsigned</code> |
| <code>constexpr</code> (since C++11) | <code>not_eq</code> | <code>using</code> (1) |
| <code>constinit</code> (since C++20) | <code>nullptr</code> (since C++11) | <code>virtual</code> |
| <code>const_cast</code> | <code>operator</code> | <code>void</code> |
| <code>continue</code> | <code>or</code> | <code>volatile</code> |
| <code>co_await</code> (since C++20) | <code>or_eq</code> | <code>wchar_t</code> |
| <code>co_return</code> (since C++20) | <code>private</code> | <code>while</code> |
| <code>co_yield</code> (since C++20) | <code>protected</code> | <code>xor</code> |
| <code>decltype</code> (since C++11) | <code>public</code> | <code>xor_eq</code> |
| | <code>reflexpr</code> (reflection TS) | |

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

8

Identifiers

- Names given to entities such as data types, objects, references, variables, functions, macros, class members, data types, etc.
- Identifiers** cannot be the same as any of the reserved words
- A valid **identifier** is a sequence of one or more letters, digits, and underscore characters
 - cannot begin with a digit
 - some compilers may impose limits on length (e.g. 2048 characters Microsoft C++)
- Examples:

<https://en.cppreference.com/w/cpp/language/identifiers>

9

Basic Data Types

- Void **void**
- Boolean **bool**
- Integer **int**
- Floating Point **float, double**
- Character **char**

10

Variables

- A **variable** is a named location in memory
 - store values during program execution
 - memory location irrelevant (we use names for access)
- C++ type system keeps track of the size of the memory block and how to interpret its contents
- Declaration:
 - Parenthesis will initialize the values as well (optional)

```
<type> <identifier> [= <initializer>];  
<type> <list of identifiers>;
```

11

Examples

12

Literals

- Tokens that represent constant values explicitly embedded in the source code
 - integers, characters, floating point, strings, boolean, user-defined
- Examples:

Escape Sequences

| Escape sequence | Description | Representation |
|--------------------------|--|-----------------------------|
| \' | single quote | byte 0x27 in ASCII encoding |
| \" | double quote | byte 0x22 in ASCII encoding |
| \? | question mark | byte 0x3f in ASCII encoding |
| \\ | backslash | byte 0x5c in ASCII encoding |
| \a | audible bell | byte 0x07 in ASCII encoding |
| \b | backspace | byte 0x08 in ASCII encoding |
| \f | form feed - new page | byte 0x0c in ASCII encoding |
| \n | line feed - new line | byte 0x0a in ASCII encoding |
| \r | carriage return | byte 0x0d in ASCII encoding |
| \t | horizontal tab | byte 0x09 in ASCII encoding |
| \v | vertical tab | byte 0x0b in ASCII encoding |
| \nnn | arbitrary octal value | byte nnn |
| \Xnn | arbitrary hexadecimal value | byte nn |
| \Unnnn (since C++11) | universal character name (arbitrary Unicode value); may result in several characters | code point U+nnnn |
| \Unnnnnnnn (since C++11) | universal character name (arbitrary Unicode value); may result in several characters | code point U+nnnnnnnn |

Statements

- Fragments of code that are executed in sequence
- Types of statements:
 - expression statements
 - compound statements
 - brace-enclosed sequences of statements
 - selection statements
 - iteration statements
 - jump statements
 - declaration statements
 - try blocks

Examples

```
int main() {  
    int n = 1;           // declaration  
    n = n + 1;           // expression  
    std::cout << "n = " << n << '\n'; // expression  
    return 0;           // jump  
}  
  
if (x > 5)               // start of if statement  
{                       // start of block  
    int n = 1;           // declaration statement  
    std::cout << n;      // expression statement  
}                       // end of block, end of if statement
```

Expressions

- An **expression** is a sequence of operators and their operands
 - it can also be a literal or a variable name, etc.
- Expression evaluation may produce a result (has a type)
 - e.g., evaluation of **2+2** produces the result **4**
- Expression evaluation may generate side-effects
 - e.g., output of a **std::cout** expression

<https://en.cppreference.com/w/cpp/language/expressions>

17

Arithmetic Expressions

Mathematical Formula

C++ Expression

$$b^2 - 4ac$$

$$b*b - 4*a*c$$

$$x(y + z)$$

$$x*(y + z)$$

$$\frac{1}{x^2 + x + 3}$$

$$1/(x*x + x + 3)$$

$$\frac{a + b}{c - d}$$

$$(a + b)/(c - d)$$

from: Problem Solving with C++, 10th Edition, Walter Savitch

18

Common operators

| assignment | increment decrement | arithmetic | logical | comparison | member access | other |
|--|--|---|---|--|---|--|
| <code>a = b</code> <code>a += b</code> <code>a -= b</code> <code>a *= b</code> <code>a /= b</code> <code>a %= b</code> <code>a &= b</code> <code>a = b</code> <code>a ^= b</code> <code>a <<= b</code> <code>a >>= b</code> | <code>++a</code> <code>--a</code> <code>a++</code> <code>a--</code> | <code>+a</code> <code>-a</code> <code>a + b</code> <code>a - b</code> <code>a * b</code> <code>a / b</code> <code>a % b</code> <code>~a</code> <code>a & b</code> <code>a b</code> <code>a ^ b</code> <code>a << b</code> <code>a >> b</code> | <code>!a</code> <code>a && b</code> <code>a b</code> | <code>a == b</code> <code>a != b</code> <code>a < b</code> <code>a > b</code> <code>a <= b</code> <code>a >= b</code> <code>a <=> b</code> | <code>a[b]</code> <code>*a</code> <code>&a</code> <code>a->b</code> <code>a.b</code> <code>a->*b</code> <code>a.*b</code> | <code>a(...)</code> <code>a, b</code> <code>? :</code> |

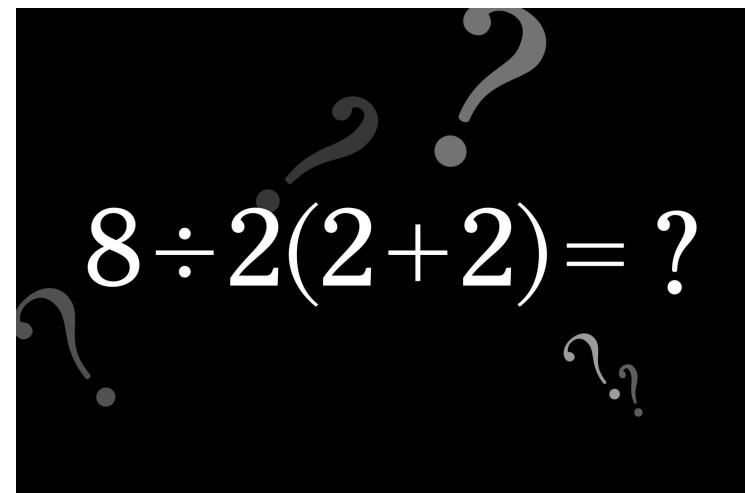
Special operators

static_cast converts one type to another related type
dynamic_cast converts within inheritance hierarchies
const_cast adds or removes **cv** qualifiers
reinterpret_cast converts type to unrelated type
C-style cast converts one type to another by a mix of **static_cast**, **const_cast**, and **reinterpret_cast**
new creates objects with dynamic storage duration
delete destructs objects previously created by the **new** expression and releases obtained memory area
sizeof queries the size of a type
sizeof... queries the size of a **parameter pack** (since C++11)
typeid queries the type information of a type
noexcept checks if an expression can throw an exception (since C++11)
alignof queries alignment requirements of a type (since C++11)

<https://en.cppreference.com/w/cpp/language/expressions>

19

Operator Precedence / Associativity



20

Operator Precedence / Associativity

- Operator precedence determines which operator is performed first in an expression with more than one operators with different precedence
- Operators Associativity is used when two operators of same precedence appear in an expression. Associativity can be either Left to Right or Right to Left.
- For example: '*' and '/' have the same precedence and their associativity is Left to Right, so the expression "100 / 10 * 10" is treated as "(100 / 10) * 10".

<https://www.geeksforgeeks.org/operator-precedence-and-associativity-in-c/>

21

Operator Precedence / Associativity

| Precedence | Operator | Description | Associativity | | |
|------------|-----------------|--|---------------|---|---------------|
| 1 | :: | Scope resolution | Left-to-right | | |
| 2 | a++ a-- | Suffix/postfix increment and decrement | | | |
| | type() type{} | Functional cast | | | |
| | a() | Function call | | | |
| | a[] | Subscript | | | |
| | . | Member access | | | |
| 3 | ++a --a | Prefix increment and decrement | Right-to-left | | |
| | +a -a | Unary plus and minus | | | |
| | ! ~ | Logical NOT and bitwise NOT | | | |
| | (type) | C-style cast | | | |
| | *a | Indirection (dereference) | | | |
| | &a | Address-of | | | |
| | sizeof | Size-of ^[note 1] | | | |
| | co_await | await-expression (C++20) | | | |
| | new new[] | Dynamic memory allocation | | | |
| | delete delete[] | Dynamic memory deallocation | | | |
| | 4 | .* ->* | | Pointer-to-member | Left-to-right |
| | 5 | a*b a/b a%b | | Multiplication, division, and remainder | |
| 6 | a+b a-b | Addition and subtraction | | | |
| 7 | << >> | Bitwise left shift and right shift | | | |
| 8 | <<= >>= | Three-way comparison operator (since C++20) | | | |
| 9 | < <= | For relational operators < and ≤ respectively | | | |
| | > >= | For relational operators > and ≥ respectively | | | |
| | == != | For relational operators == and ≠ respectively | | | |
| 11 | & | Bitwise AND | | | |
| 12 | ^ | Bitwise XOR (exclusive or) | | | |
| 13 | | Bitwise OR (inclusive or) | | | |
| 14 | && | Logical AND | | | |
| 15 | | Logical OR | | | |

https://en.cppreference.com/w/cpp/language/operator_precedence

22

Operator Precedence / Associativity

| | | | |
|----|----------|---|---------------|
| 16 | a?b:c | Ternary conditional ^[note 2] | Right-to-left |
| | throw | throw operator | |
| | co_yield | yield-expression (C++20) | |
| | = | Direct assignment (provided by default for C++ classes) | |
| | += -= | Compound assignment by sum and difference | |
| | *= /= %= | Compound assignment by product, quotient, and remainder | |
| | <<= >>= | Compound assignment by bitwise left shift and right shift | |
| 17 | &= ^= = | Compound assignment by bitwise AND, XOR, and OR | Left-to-right |
| | , | Comma | |

https://en.cppreference.com/w/cpp/language/operator_precedence

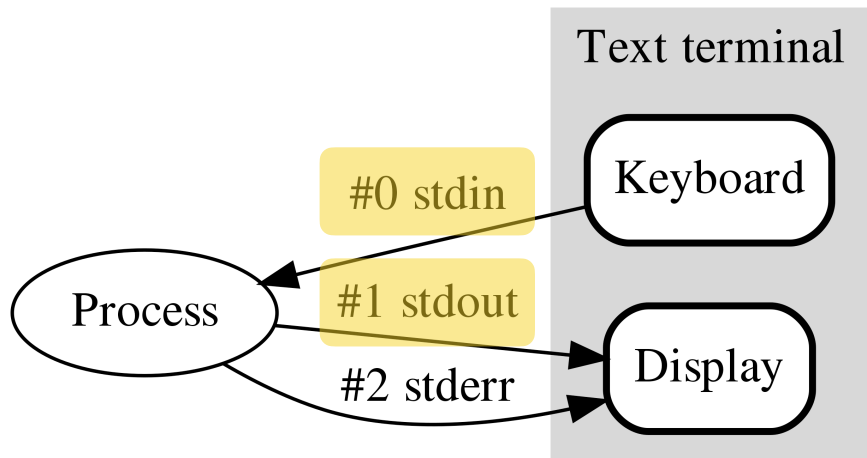
23

Basic Input / Output

- Data streams are just sequences of data
- Input Stream**
 - data passed to programs
 - typically originates from keyboard or files
- Output Stream**
 - output from programs
 - typically goes to the terminal / monitor or files

24

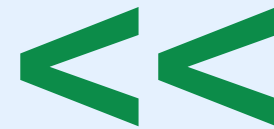
Basic Input/Output



25

`std::cout`

the output stream

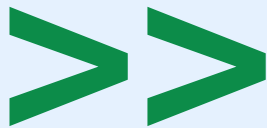


the insertion operator

26

`std::cin`

the input stream



the extraction operator

27

Include directives

- Required to add **library** files to programs
- For standard **input** and **output** use:

```
#include <iostream>
```

28