CSC 211: Computer Programming

Basic C++ Concepts and Syntax

Michael Conti

Department of Computer Science and Statistics
University of Rhode Island

Spring 2025

Original design and development by Dr. Marco Alvarez

Administrative Notes

Administrative notes

- A00
- Micro-Programming Challange#01
- Make sure you're in contact with group for A00
  - If you fail to respond and your group moves forward without you, you won't be listed for credit in A00 submission

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

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

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;
*/
```


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.

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

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

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

Basic Data Types

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

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>;
```


Examples

Literals

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

https://en.cppreference.com/w/cpp/language/expressions#Literals

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

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

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

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

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

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

Common operators

assignment	increment decrement	arithmetic	logical	comparison	member access	other

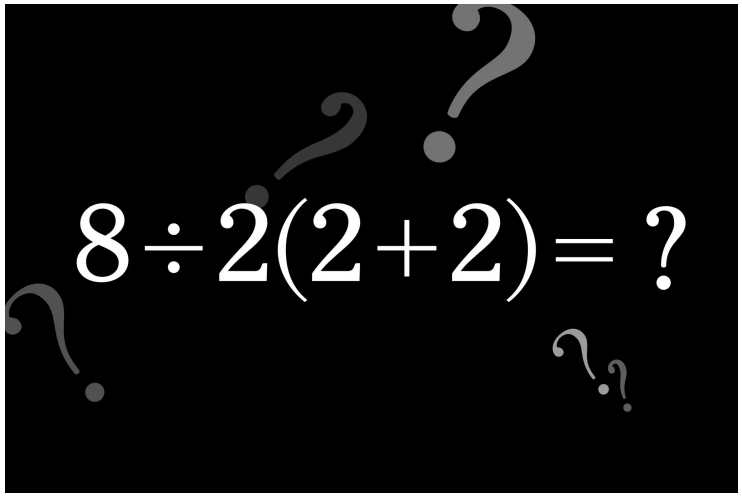
`a = b`
`a += b`
`a -= b`
`a *= b`
`a /= b`
`a %= b`
`a &= b`
`a |= b`
`a ^= b`
`a <<= b`
`a >>= b`
	`++a`
	`--a`
	`a++`
	`a--`
		`+a`
		`-a`
		`a + b`
		`a - b`
		`a * b`
		`a / b`
		`a % b`
		`~a`
		`a & b`
		`a | b`
		`a ^ b`
		`a << b`
		`a >> b`
			`!a`
			`a && b`
			`a || b`
				`a == b`
				`a != b`
				`a < b`
				`a > b`
				`a <= b`
				`a >= b`
				`a <=> b`
					`a[b]`
					`*a`
					`&a`
					`a->b`
					`a.b`
					`a->*b`
					`a.*b`
						`a(...)`
						`a, b`
						`? :`

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

Operator Precedence / Associativity



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/

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	
10	`== !=`	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

Operator Precedence / Associativity

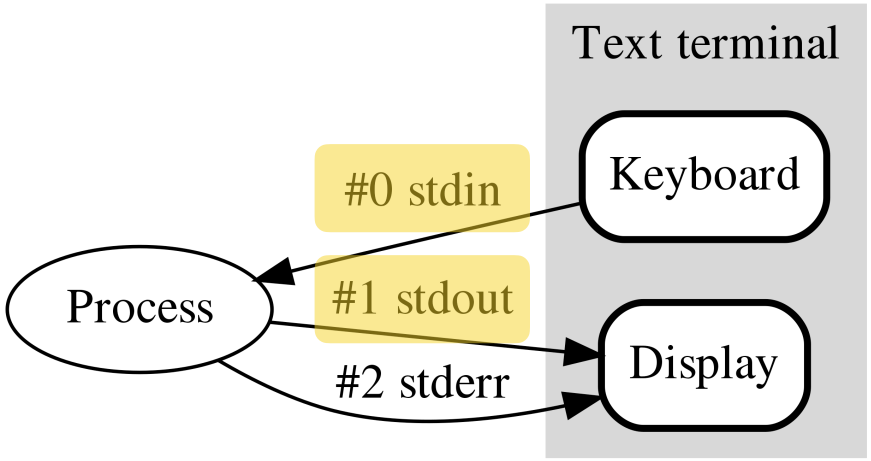
16	a?b:c	Ternary conditional[note 2]	Right-to-left
	throw	throw operator	
	co_yield	yield-expression (C++20)	
	=	Direct assignment (provided by default for C++ classes)	
	+= -=	Compound assignment by sum and difference	
	*= /= %=	Compound assignment by product, quotient, and remainder	
	<<= >>=	Compound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

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

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

Basic Input/Output



`std::cout`

the output stream

`<<`

the insertion operator

`std::cin`

the input stream

`>>`

the extraction operator

Include directives

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

```
#include <iostream>
```