MEC510: Object-Oriented Programming for Scientists and Engineers

(Spring 2024)

Amin Fakhari, Ph.D.

Department of Mechanical Engineering

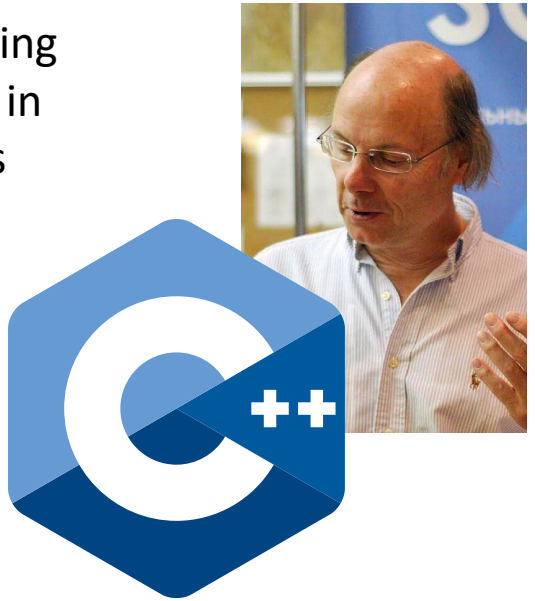
Stony Brook University

Ch1: Introduction, Input-Output, and Operators

Introduction to C++

C++

C++, an extension of C, is a high-level general-purpose programming language created by Danish computer scientist Bjarne Stroustrup in 1979 at Bell Laboratories. Originally called "C with Classes," it was renamed to C++ in the early 1980s. C++ provides capabilities for object-oriented programming.



- C++ is standardized by the ISO:

Year	C++ Standard
1998	C++98
2003	C++03
2011	C++11, C++0x
2014	C++14, C++1y
2017	C++17, C++1z
2020	C++20, C++2a
2023	C++23

https://isocpp.org
http://cppreference.com

- Filename Extensions:
  .cpp, .cxx, .C (uppercase), .cc, .c++
  .h, .hpp, .hxx, .H, .hh, .h++

- C++ programs consist of pieces called classes and functions. You can program each piece yourself, but most C++ programmers take advantage of the rich collections of classes and functions in the **C++ Standard Library**.

Machine Languages, Assembly Languages, and High-Level Languages

Machine Languages: Languages that computers can directly understand/process without a previous transformation. Machine languages generally consist of numbers (ultimately reduced to 1s and 0s).

A function in hexadecimal representation to calculate the nth Fibonacci number:

```
8B542408  83FA0077  06B80000  0000C383
FA027706  B8010000  00C353BB  01000000
B9010000  008D0419  83FA0376  078BD989
C14AEBF1  5BC3
```

Assembly (or Low-Level) Languages: Languages that use English-like abbreviations for performing elementary operations. Translator programs called assemblers convert these languages directly to machine language.

The same code in assembly language:

```
_fib:
        movl $1, %eax
        xorl %ebx, %ebx
.fib_loop:
        cmpl $1, %edi
        jbe .fib_done
        movl %eax, %ecx
        addl %ebx, %eax
        movl %ecx, %ebx
        subl $1, %edi
        jmp .fib_loop
.fib_done:
        ret
```

Machine Languages, Assembly Languages, and High-Level Languages (cont.)

High-Level Languages: These languages (such as C, C++, Python, Java) look more like everyday English and contain commonly used mathematical notations. Single statements can be written to accomplish substantial tasks. Two modes of execution for high-level languages:

- **Compiled**

  A translator program called **compiler** convert high-level language programs into machine language. *Ex.*: C/C++.

- **Interpreted**

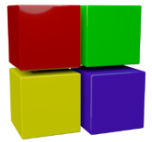
  A program called an **interpreter** reads each program statement and execute it directly, with no compilation stage. *Ex.*: Python, MATLAB.

```
unsigned int fib(unsigned int n) {
  if (!n)
     return 0;
  else if (n <= 2)
     return 1;
  else {
     unsigned int a, c;
     for (a = c = 1; ; --n) {
        c += a;
        if (n <= 2) return c;
          a = c - a;
     }
  }
}
```

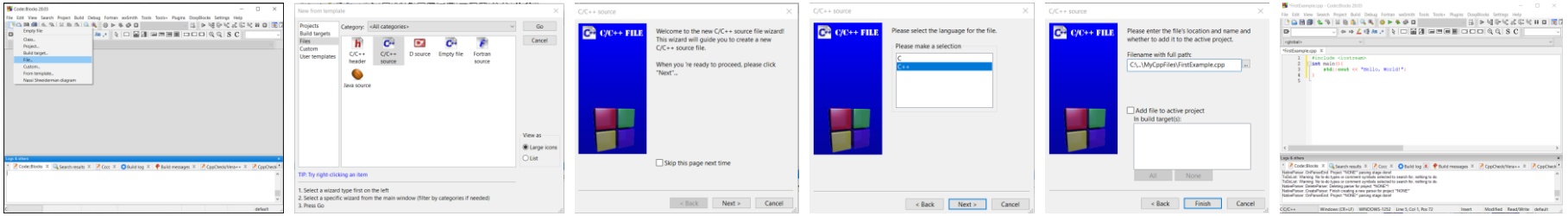
(The same code in C language)

C++ Integrated Development Environments (IDEs)

There are many C++ IDEs to write/compile your source code and you can choose among them. **Code::Blocks** is a free, open-source, cross-platform IDE.



In Code::Blocks: “New” > “File...” > “C/C++ source” > “Go” > “Next” > “C++” > choose or make a folder and name your file > “Finish” > write your code > click “Build and run”



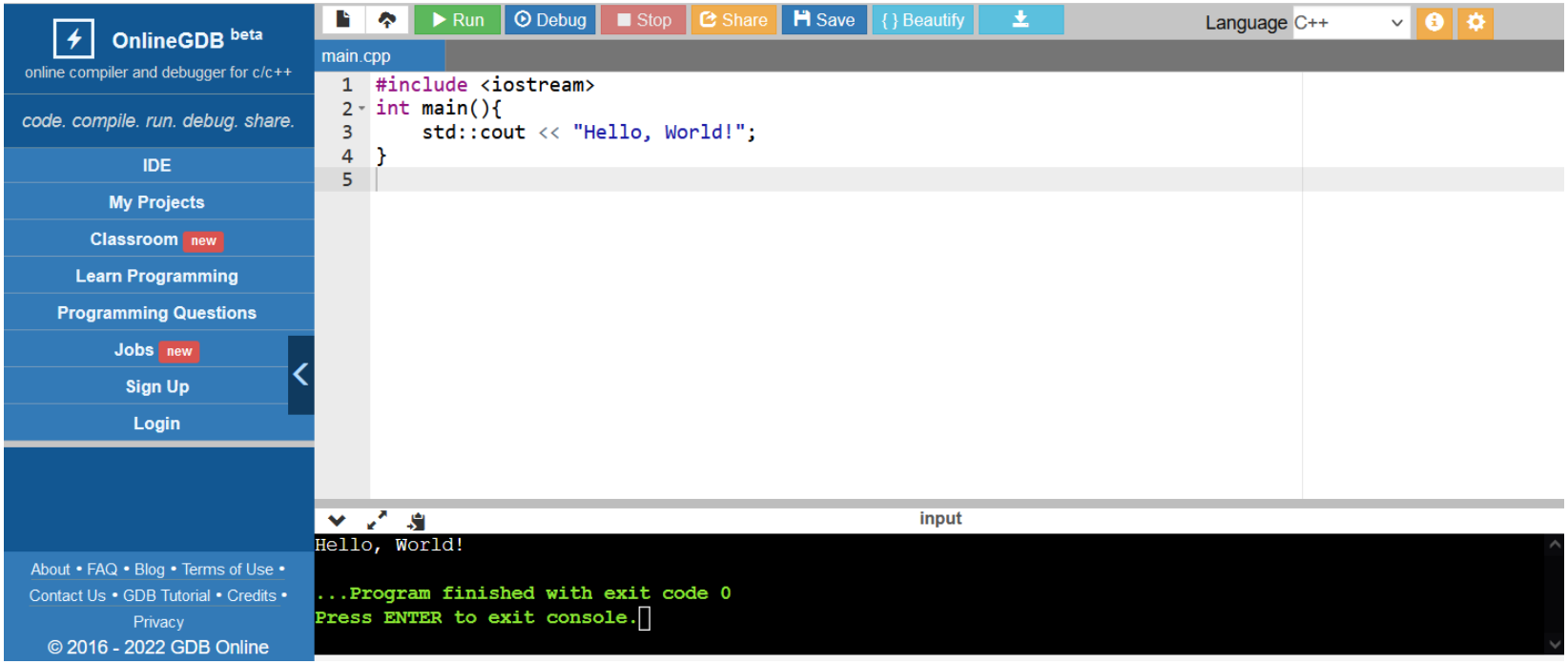
Learn how to set up your Linux-based, Windows-based, or Apple OS-based computer to run the following code to print "Hello, World!".

```
#include <iostream>
int main(){
    std::cout << "Hello, World!";
}
```

C++ Integrated Development Environments (IDEs) (cont.)

You can also use online IDEs like [OnlineGDB](https://www.onlinegdb.com/).

In OnlineGDB, you can write your code, choose “C++” language, and Run. Preferably, create a free account to save your codes.



Comments, Strings, Output Stream

Sample Program: Printing a Text

A simple program that prints a text:

```
/* First Program in C++
Text-printing program. */
#include <iostream> // enables program to output data to the screen

// function main begins program execution
int main() {
  std::cout << "Welcome to C++!\n"; // display message

  std::cout << "Welcome to C++!" << std::endl; // display message; end line

  // Printing a line of text with multiple statements.
  std::cout << "Welcome ";
  std::cout << "to C++!\n";

  // Printing multiple lines of text with a single statement.
  std::cout << "Welcome\nto\n\nC++!\n";

  return 0; // indicate that program ended successfully
} // end function main
```

Comments

- Comments are used to document the programs and to help other people read and understand them.
- It is always advised to begin a program with a comment that describes the purpose of the program.
- They are ignored by the compiler.

Single-Line Comment: "//" in each line indicates that the remainder of the line is a comment.

```
// Text-printing program.
#include <iostream> // enables program to output data to the screen
```

Multi-Line Comment: Line(s) enclosed in "/*" and "*/" are comments.

```
/* First Program in C++
Text-printing program. */
```

Preprocessor Directive #include

```
#include <iostream>
```

Preprocessor directives are lines of code that begin with # and do not end with a semicolon. These lines are not program statements but directives for the preprocessor, and they are processed by the preprocessor before the program is compiled.

- #include notifies the preprocessor to insert (copy and paste!) the contents of another file (C++ Standard Library headers or User-defined headers) into the source code at the point where the #include directive is found.

- iostream is the input/output stream header (from C++ Standard Library headers) which is required when compiling any C++ program that outputs data to the screen or inputs data from the keyboard.

White Space

```
// Text-printing program.
#include <iostream>

      // function main begins program execution
```

- **White space** (e.g., blank lines, space characters, and tab characters) are used to make programs easier to read.
- White-space characters are ignored by the C++ compiler.

main Function

- **function** main is a necessary part of every C++ program.
- C++ programs begin executing at function main, even if main is not the first function defined in the program.
- Exactly one function in every program must be named main.
- Body of the function is written inside braces {}.
- Indent the body of each function one level within the braces {} (e.g., three spaces per level of indent). This makes the program easier to read.

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

No semicolon here.

- **Keyword** int indicates that main "returns" an integer value.

Note: Files ending with the .cpp filename extension are **source-code files**. These define a program's main function, other functions, and more. Headers are included into source-code files.

C++ Keywords

A **keyword** is a word in code that is reserved by C++ for a specific use and must appear in all lowercase letters.

C++-only keywords

and	and_eq	bitand	bitor	bool
catch	class	compl	const_cast	delete
dynamic_cast	explicit	export	false	friend
mutable	namespace	new	not	not_eq
operator	or	or_eq	private	protected
public	reinterpret_cast	static_cast	template	this
throw	true	try	typeid	typename
using	virtual	wchar_t	xor	xor_eq

Keywords common to the C and C++ programming languages

asm	auto	break	case	char
const	continue	default	do	double
else	enum	extern	float	for
goto	if	inline	int	long
register	return	short	signed	sizeof
static	struct	switch	typedef	union
unsigned	void	volatile	while	

C++11 keywords

alignas	alignof	char16_t	char32_t	constexpr
decltype	noexcept	nullptr	static_assert	thread_local

Strings

```
std::cout << "Welcome to C++!\n";
```

- The entire line is called a **Statement**. Most C++ statements end with a semicolon (;).
- The quotation marks " " and the characters between them are called a **String**.
- White-space characters in strings are not ignored by the compiler.
- In a string of characters, the backslash (\) is called an **escape character**. \ and its next character are not printed, and they form an **escape sequence**.

Escape sequence	Description
\n	Newline. Position the screen cursor to the beginning of the next line.
\t	Horizontal tab. Move the screen cursor to the next tab stop.
\r	Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line.
\a	Alert. Sound the system bell.
\\	Backslash. Used to print a backslash character.
\'	Single quote. Used to print a single-quote character.
\"	Double quote. Used to print a double-quote character.

Standard Output Stream

```
std::cout << "Welcome to C++!\n";
```

- cout (**standard output stream**) belongs to **namespace** std and is defined in <iostream> header file.
- The identifiers of the C++ Standard Library headers are defined in a namespace called std. The notation std:: specifies that we are using a name that belongs to namespace std.
- In the context of an output statement, the **stream insertion operator <<** send the stream of characters between quotation marks " " to std::cout object to print on the screen. The << operator points toward where the data goes.

endl Stream Manipulator

```
std::cout << "Welcome to C++!" << std::endl;
```

- endl is a (nonparameterized) **stream manipulator**.
- endl is an abbreviation for "end line". It belongs to namespace std and is defined in <iostream> header file.
- endl outputs a newline, then flushes the output buffer: `std::endl` ⇔ `"\n" << std::flush`

Thus, these have the same output but the first one is probably a bit faster:

```
std::cout << "Welcome to C++!\n";
std::cout << "Welcome to C++!" << std::endl;
```

- Flushing the output buffer is very rarely necessary. Thus, you do not have to use endl except possibly for aesthetic reasons.

- The insertion operator (<<) can be used multiple times in a single statement to concatenate multiple pieces of output:

```
std::cout << "Welcome "
          << "to C++!\n";
```

Other Ways of Printing

- Printing a line of text with multiple statements:

```
std::cout << "Welcome ";
std::cout << "to C++!\n";
```

Each stream insertion resumes printing where the previous one stopped.

- Printing multiple lines of text with a single statement:

```
std::cout << "Welcome\nto\n\nC++!\n";
```

Each time the \n (newline) is encountered in the output stream, the screen cursor is positioned to the beginning of the next line.

Best practice: Output a newline whenever a line of output is complete (even for the last output).

return **Statement**

```
return 0;
```

This statement at the end of main indicates that the program has terminated successfully. According to the C++ standard, if program execution reaches the end of main without encountering a return statement, it is assumed that the program terminated successfully. Therefore, we can omit the return statement at the end of main.

Compilation: Behind the Scenes, When Running a C++ Program

helloworld.cpp > ...

```cpp
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;
}
```

Preprocessing: Preprocessing directives are executed to include (i.e., copy into the program file) other text files (libraries) and/or perform various text replacements. The extension of the generated file is *.i.

helloworld.i 9+

helloworld.i > main()

```
42393   extern __attribute__ ((__visibility__("default"))) wostream wcout;
42394
42395   extern __attribute__ ((__visibility__("default"))) ostream cerr;
42396   extern __attribute__ ((__visibility__("default"))) wostream wcerr;
42397   extern __attribute__ ((__visibility__("default"))) ostream clog;
42398   extern __attribute__ ((__visibility__("default"))) wostream wclog;
42399
42400   } }
42401   # 2 "helloworld.cpp" 2
42402
42403   int main() {
42404       std::cout << "Hello World" << std::endl;
42405
42406   }
```

Compilation: Behind the Scenes, When Running a C++ Program (cont.)

Compiling: The compiler translates the program into machine-language code (known as object code). The extension of the generated file is *.o.

helloworld.o

```
                ...
1328: f8 0b 00 00    udf     #3064
132c: 00 00 00 00    udf     #0
1330: 88 00 00 00    udf     #136
1334: 00 00 00 44    <unknown>
                ...
1340: 98 0d 00 00    udf     #3480
1344: 00 00 00 00    udf     #0
1348: 80 0c 00 00    udf     #3200
134c: 00 00 00 00    udf     #0
1350: 2c 00 00 00    udf     #44
1354: 00 00 00 04    <unknown>
                ...
1368: ac 0c 00 00    udf     #3244
136c: 00 00 00 00    udf     #0
1370: 3c 00 00 00    udf     #60
1374: 00 00 00 04    <unknown>
                ...
1388: e8 0c 00 00    udf     #3304
138c: 00 00 00 00    udf     #0
1390: 38 00 00 00    udf     #56
1394: 00 00 00 04    <unknown>
```

Linking: C++ programs typically contain references to functions and data defined elsewhere. The object code produced by the compiler typically contains “holes” due to these missing parts. A linker links the object code with the code for the missing functions to produce a single executable program *.exe file that the computer can understand and run the program.

HelloWorld.exe

Variables, Input Stream, Operators

Sample Program: Adding Two Integers

A program that obtains two integers typed by a user at the keyboard, computes their sum and print the result:

```
// Addition program that displays the sum of two integers.
#include <iostream>

int main() {
  // declaring and initializing variables
  int number1{0}; // first integer to add (initialized to 0)
  int number2{0}; // second integer to add (initialized to 0)
  int sum{0}; // sum of number1 and number2 (initialized to 0)

  std::cout << "Enter first integer: "; // prompt user for data
  std::cin >> number1; // read first integer from user into number1

  std::cout << "Enter second integer: "; // prompt user for data
  std::cin >> number2; // read second integer from user into number2

  sum = number1 + number2; // add the numbers; store result in sum

  std::cout << "Sum is " << sum << std::endl; // display sum; end line
}
```

Variable Declarations & Variable Assignment

All variables must be declared with a name (**identifier**) and a data type (and optionally initialized) almost anywhere in a program, but before they are used in a program.

```
int x;    // define an integer variable named x
int y, z; // define two integer variables, named y and z
```

These **variable declarations** specify that the data type of identifiers x, y, z is int (holding integer values).

After a variable has been defined, you can give it a value (in a separate statement) using the = operator. This process is called **Variable Assignment**, and the = operator is called the assignment operator.

```
int width; // define an integer variable named width
width = 5; // assignment of value 5 into variable width
```

Variable Initialization

There are some basic ways to initialize variables in C++:

```
int a;       // no initializer (default initialization), "a" may have an indeterminate value.
int b = 5;    // initializer after equals sign (copy initialization)
int c( 6 );   // initializer in parenthesis (direct initialization)

// List initialization methods (introduced in C++11) (preferred method)
int d { 7 };  // direct list initialization
int e = { 8 }; // copy list initialization
int f {};     // value initialization (zero initialization)
```

In most cases, value initialization will initialize the variable to zero (or empty, if that's more appropriate for a given type).

Best Practice: Initialize your variables upon creation.

- Several variables of the same type may be declared and initialized in one declaration by using a **comma-separated** list:

```
int a = 5, b = 6;         // copy initialization
int c( 7 ), d( 8 );       // direct initialization
int e { 9 }, f { 10 };    // direct list initialization
int g = { 9 }, h = { 10 }; // copy list initialization
int i {}, j {};           // value initialization
int a, b = 5; // Note: "a" is not initialized!
```

Remarks: Fundamental Data Types

By order of increasing memory requirements in each type:

Integral types	Floating-point types
`bool` (for holding true and false)	`float`
`char`	`double`
`signed char`	`long double`
`unsigned char`	
`short int` (syn. with short)	
`unsigned short int` (syn. with unsigned short)	
`int`	
`unsigned int` (syn. with unsigned)	
`long int` (syn. with long)	
`unsigned long int` (syn. with unsigned long)	
`long long int` (syn. with long long)	
`unsigned long long int` (syn. with unsigned long long)	
`char16_t`, `char32_t`, `wchar_t` (for representing Unicode characters)	

Note: Each **signed** integer type has a corresponding **unsigned** integer type (that cannot represent negative values) with the same memory requirements. The unsigned type can represent approximately **twice** as many positive values as the signed integer types.

Note: The exact ranges of values for the fundamental types are machine-dependent, and they can be specified by using the header files <climits> (for the integral types) and <cfloat> (for the floating-point types). For example, the max. and min. values stored in an int:

```
#include<iostream>
#include<climits>
int main() {
  std::cout << "INT_MIN: " << INT_MIN << "\n";
  std::cout << "INT_MAX: " << INT_MAX << "\n";
  std::cout << "UINT_MAX: " << UINT_MAX;
}
```

Fundamental Data Type Sizes

The size of a given data type is dependent on the compiler and/or the computer architecture. C++ only guarantees that each fundamental data types will have a minimum size:

Category	Type	Minimum Size
boolean	bool	1 byte
character	char	1 byte
	wchar_t	1 byte
	char16_t	2 bytes
	char32_t	4 bytes
integer	short	2 bytes
	int	2 bytes
	long	4 bytes
	long long	8 bytes
floating point	float	4 bytes
	double	8 bytes
	long double	8 bytes

An object with n bits can hold 2^n unique values. Therefore, with an 8-bit byte, a byte-sized object can hold 2^8 (256) different values.

- For maximum compatibility, you should not assume that variables are larger than the specified minimum size.

sizeof Operator

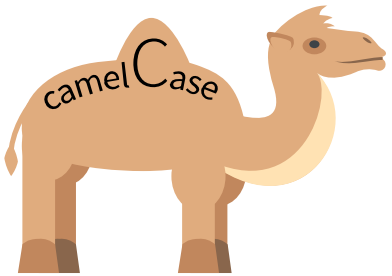
In order to determine the size of data types on a particular machine, C++ provides an operator named sizeof. The sizeof operator is a unary operator that takes either a **type** or a **variable**, and returns its size in bytes.

```
#include <iostream>
int main() {
   std::cout << "bool:\t\t" << sizeof(bool) << " bytes\n";
   std::cout << "char:\t\t" << sizeof(char) << " bytes\n";
   std::cout << "wchar_t:\t" << sizeof(wchar_t) << " bytes\n";
   std::cout << "char16_t:\t" << sizeof(char16_t) << " bytes\n";
   std::cout << "char32_t:\t" << sizeof(char32_t) << " bytes\n";
   std::cout << "short:\t\t" << sizeof(short) << " bytes\n";
   std::cout << "int:\t\t" << sizeof(int) << " bytes\n";
   std::cout << "long:\t\t" << sizeof(long) << " bytes\n";
   std::cout << "long long:\t" << sizeof(long long) << " bytes\n";
   std::cout << "float:\t\t" << sizeof(float) << " bytes\n";
   std::cout << "double:\t\t" << sizeof(double) << " bytes\n";
   std::cout << "long double:\t" << sizeof(long double) << " bytes\n";
}
```

```
#include <iostream>
int main() {
   int x{0};
   std::cout << "x is " << sizeof(x) << " bytes\n";
}
```

Remarks: Identifiers

- An identifier (variable name) is a series of characters consisting of letters, digits, and underscores (_) that does not begin with a digit, and it cannot be a C++ keyword.
- C++ is case sensitive (uppercase and lowercase letters are different).
- Use identifiers of 31 characters or fewer to ensure portability (and readability).
- Choosing meaningful identifiers helps make a program self-documenting.
- Do not use identifiers that begin with underscores and double underscores, because C++ compilers use names like that for their own purposes internally.
- By convention, variable-name identifiers begin with a lowercase letter, and every word in the name after the first word begins with a capital letter, e.g., firstNumber starts its second word, Number, with a capital N. This naming convention is known as **camel case**.



Standard Input Stream

```
std::cout << "Enter first integer: ";
```

A message that directs the user to take a specific action is known as a **prompt**. We prompt the user to enter something.

```
std::cin >> number1;
```

- cin (**standard input stream**) belongs to **namespace** std and is defined in <iostream> header file.
- This statement uses cin and the **stream extraction operator >>** to obtain a value from the keyboard and assign it to variable number1.

When the computer executes this statement, it waits for the user to enter a value for variable number1. After typing characters and pressing the Enter (Return) key, the computer converts the character to an integer and assigns it to the variable number1 (the cursor also moves to the beginning of the next line on the screen).

Note: The extraction operation uses the **type** of the variable after the >> operator to determine how it interprets the characters read from the input; if it is an integer, the format expected is a series of digits, if a string a sequence of characters, etc.

Remarks

- Instead of

```
std::cout << "Enter first integer: ";
std::cin >> number1;
std::cout << "Enter second integer: ";
std::cin >> number2;
```

you could also use **cascaded stream extraction operations** to input two integers (the first value is read into number1, the second value is read into number2). A space, a tab, or Enter can be used to separate the consecutive input operations:

```
std::cout << "Enter two integers to add: ";
std::cin >> number1 >> number2;
```

- (Lengthy) Statements may be spaced and split over several lines according to your preferences.

```
int

   number1{0};
```

(acceptable)

```
std::cout << number1 <<
    " == " << number2 << std::endl;
```

(acceptable)

However, it is a syntax error to split identifiers, strings, and constants (such as the numbers) over several lines.

Assignment and Arithmetic Operators

```
sum = number1 + number2;
```

"=" is called **Assignment Operator**

Important Note: Integer division (where both operands are integers) yields an integer quotient, i.e., any fractional part of the calculation is truncated not rounded (19 ÷ 5 = 3.8 where 19 / 5 truncates to 3, rather than rounding to 4).

% operator is used only with integer operands and yields the remainder after integer division (19 % 5 → 4).

Operation	Arithmetic Operator
Addition	+
Subtraction	-
Multiplication	* (asterisk)
Division	/
Remainder	% (percent sign)

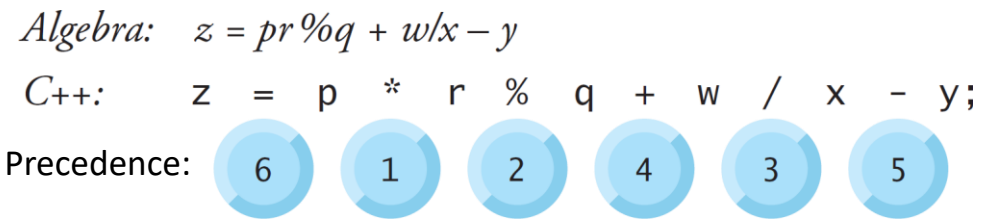
- C++ has three types of operators: **Unary**, **Binary**, and **Ternary**, which take one, two, and three operands, respectively. The assignment and arithmetic operators are **binary** operators. C++ also supports **unary** versions of the plus (+) and minus (-) operators, so that you can write such expressions as -7 or +5.
- Place spaces on either side of a Binary or Ternary operator to make the program more readable.
- You could eliminate the need for the variable sum by

```
std::cout << "Sum is " << number1 + number2 << std::endl;
```

Assignment and Arithmetic Operations

C++ applies the operators in arithmetic expressions based on Rules of Operator Precedence and Associativity, which are generally the same as those in algebra.

1. Operators in expressions contained within "()" are evaluated first (starting from innermost pair).
2. *, /, % operations are evaluated on the same level of precedence (starting from left to right).
3. +, - operations are evaluated on the same level of precedence (starting from left to right).



- There is no arithmetic operator for **Exponentiation** in C++, but there is a standard library function pow ("power") that performs exponentiation.

- Using **Redundant Parentheses** in an expression to make it clearer is acceptable.

```
y = (a * x * x) + (b * x) + c;
```

Algorithm and Pseudocode

Before writing a program to solve a problem, you should have a thorough understanding of the problem and a carefully planned approach to solving it.

A procedure for solving a problem in terms of the **actions** (statements) to execute and the **order** in which these actions execute (program control) is called an **algorithm**.

- **Pseudocode** is an informal plain language description of the key principles and steps in an **algorithm**, that helps developing algorithms without having to worry about the strict details of a programming language syntax.
- It is intended for human reading and understanding rather than machine reading.
- Pseudocode typically does not include details such as variable declarations and language-specific code, but includes input, output, assignments, compact mathematical notation, and calculations.
- It is commonly used in textbooks and scientific publications to document algorithms.
- A carefully prepared pseudocode program can easily be converted to a corresponding program in any language.

Pseudocode: Addition Program

- Prompt the user to enter the first integer
- Input the first integer

- Prompt the user to enter the second integer
- Input the second integer

- Add first integer and second integer, store result
- Display result

```
// Addition program that displays the sum of two integers.
#include <iostream>

int main() {
  // declaring and initializing variables
  int number1{0}; // first integer to add (initialized to 0)
  int number2{0}; // second integer to add (initialized to 0)
  int sum{0}; // sum of number1 and number2 (initialized to 0)

  std::cout << "Enter first integer: "; // prompt user for data
  std::cin >> number1; // read first integer from user into number1

  std::cout << "Enter second integer: "; // prompt user for data
  std::cin >> number2; // read second integer from user into number2

  sum = number1 + number2; // add the numbers; store result in sum

  std::cout << "Sum is " << sum << std::endl; // display sum; end line
}
```

if Statement, Relational Operators

Sample Program: Comparing Integers Using if Statements

```
// Comparing integers using if statements, relational/equality operators.
#include <iostream>

using std::cout; // program uses cout
using std::cin; // program uses cin
using std::endl; // program uses endl

int main() {
  int number1{0};
  int number2{0};

  cout << "Enter 1st integer to compare: ";
  cin >> number1;

  cout << "Enter 2nd integer to compare: ";
  cin >> number2;

  if (number1 == number2){
    cout << number1 << " == " << number2 << endl;
  }

  if (number1 < number2) {
    cout << number1 << " != " << number2 << endl;
    cout << number1 << " < " << number2 << endl;
  }

  if (number1 > number2) {
    cout << number1 << " != " << number2 << endl;
    cout << number1 << " > " << number2 << endl;
  }
}
```

using **Declarations and** using **Directive**

```
using std::cout;
using std::cin;
using std::endl;
```
(1)

- using **declarations** eliminate the need to repeat the std:: prefix. We can now write cout instead of std::cout, cin instead of std::cin and endl instead of std::endl, respectively, in the remainder of the program.

- Instead of the above method, one can use using **directive** as a single statement

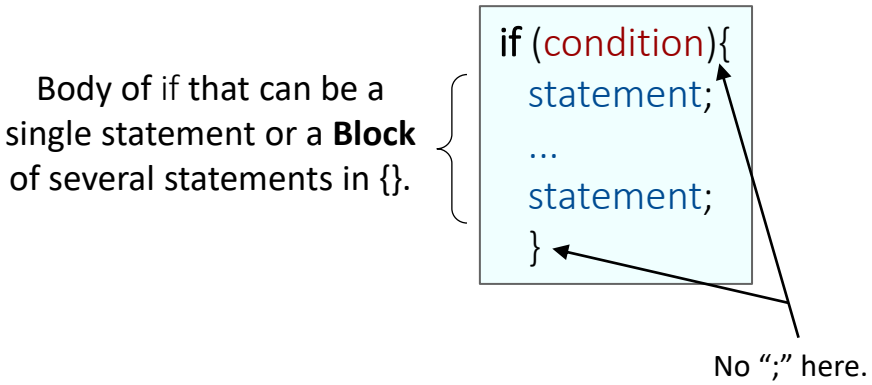
```
using namespace std;
```
(2)

This enables a program to use <u>all</u> the names defined in any Standard C++ header, such as <iostream> (i.e., for any name the compiler cannot find, it tries looking it up in std).

Note: The best practice is to repeat the std:: prefix for the names that belong to namespace std (C++ Standard Library headers). If you do not want to repeat, method (1) is usually a better practice compared with method (2).

if Statement (or Single-Selection Statement)

if statement (or single-selection statement) performs (selects) an action (or group of actions) in its body, if a condition is true, or skips it, if the condition is false.



- You do not need to use braces, { }, around single-statement bodies. However, it is always recommended to enclose all the statement bodies in braces to avoid logic errors.

```
if (condition)
  statement;
```

- The **indentation** of the statement(s) in the body of an if statement, which enhances readability, is optional, but recommended. Many IDEs do indentation automatically.

Relational and Equality Operators

Conditions are usually formed by using the **relational and equality operators**.

Algebraic relational or equality operator	C++ relational or equality operator	Sample C++ condition	Meaning of C++ condition
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y
Equality operators			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y

- The expressions containing relational or equality operators always result in a **Boolean** value, true (1) or false (0). The data type bool (Boolean variables) can hold these values.

```
if (4 > 2){
  statements;
}
```

```
bool x{true}
if (x){
  statements;
}
```

```
bool x{4 > 2}
if (x){
  statements;
}
```

- In C++, a decision in the if statement can be also based on any expression that evaluates to **zero** or **nonzero**. A zero value corresponds to false, and any nonzero value corresponds to true.

```
bool x{0}
if (x){
  statements;
}
```
(condition is false)

```
int x{4}
if (x){
  statements;
}
```
(condition is true)

== (equality) Operator vs = (assignment) Operator

Accidentally swapping the operators == (equality) and = (assignment) usually does not cause syntax errors, but often generates incorrect results through runtime logic errors.

```
if (x == 4) {
   statement(s);
}
```

vs

```
if (x = 4) {
   statement(s);
}
```


Since assignments produce a value (the value assigned to the variable), the condition always evaluates as true. Moreover, the value of x has changed when it was only supposed to be examined!

Operator Precedence and Associativity

decreasing order of precedence

Operators				Associativity	Type
()				starting from innermost pair	grouping parentheses
*	/	%		left to right	multiplicative
+	-			left to right	additive
<<	>>			left to right	stream insertion/extraction
<	<=	>	>=	left to right	relational
==	!=			left to right	equality
=				right to left	assignment

C++ Operator Precedence

- If you are uncertain about the order of evaluation in a complex expression, break the expression into smaller statements or use parentheses to force the order of evaluation, exactly as you would do in an algebraic expression.

Errors in C++

Errors in C++

In C++, errors can be categorized into four different types:

1. Syntax Errors (Compile-Time Errors): These errors occur when the rule of C++ writing techniques or syntax has been broken. These types of errors are typically flagged by the compiler prior to compilation.

```
#include <iostream>
int main() {
  std::cout << "Hello, World!" // missing semicolon
}
```

2. Runtime Errors (Execution-Time Errors): These errors occur while the program is running (after the compilation). For example, when a number is divided by 0.

```
#include <iostream>
int main() {
   int x = 52;
   int y = 0;
   std::cout << x / y << "\n";
   std::cout << x % y;
}
```

Errors in C++

3. Logical Errors: These errors occur due to logical issues (such as an incorrect calculation or putting a semicolon in a wrong place) and result in incorrect results/output.

```
#include <iostream>
int main() {
  int grade{50};
  if (grade >= 60); {
    std::cout << "Passed";
  }
}
```

```
#include <iostream>
int main() {
  int i;
  for (i = 0; i < 5; i++); {
    std::cout << "Hello, World!, ";
  }
}
```

4. Linker Errors: These errors occur when the program is successfully compiled and trying to link the different object files with the main object file, but the executable is not generated. For example, when an incorrect header file is used or main() is written as Main().

```
#include <iostream>
int Main() {
  int x = 52;
  std::cout << x;
}
```