

# Introduction to Object-Oriented Programming

## COMP2011: C++ Basics I

Dr. Cindy Li  
Dr. Brian Mak  
Dr. Dit-Yan Yeung

Department of Computer Science & Engineering  
The Hong Kong University of Science and Technology  
Hong Kong SAR, China



- Just like different **human languages** (e.g. Chinese, English, Japanese, French, etc.), each **programming language** (e.g. Pascal, C++, Java, etc.) has its own
  - **vocabulary** = the set of legal “words”
  - grammar or **syntax**: how “words” are put together to make a legal “sentence”
- A **program** consists of a sequence of **statements** (c.f. sentence in human language)
- Some parts of a **statement** are called **expressions** (c.f. phrase in human language). e.g.
  - **logical expression**:  $x > y$
  - **arithmetic expression**:  $5 + 4$

# Part I

## A Simple C++ Program

# Example: Hello World!

```
/*  
 * File: hello-world.cpp  
 * A common program used to demo a new language  
 */  
  
#include <iostream>           // Load info of a Standard library  
using namespace std;         // Standard C++ namespace  
  
int main( )                   // Program's entry point  
{  
    cout << "Hello World!" << endl; // Major program codes  
    return 0;                 // A nice ending  
}
```

# Write, Compile, and Run a Program in a Terminal

STEP 1 : Write the program using an **editor**.

e.g., **Eclipse**, **vi** (Unix/Linux), **MS Word** (Windows)

STEP 2 : Save the program into a file called **hello-world.cpp**.

STEP 3 : Compile the program using **g++** compiler.

```
g++ -o hello-world hello-world.cpp
```

If you don't specify the output filename using the “**-o**” option, the default is **a.out**.

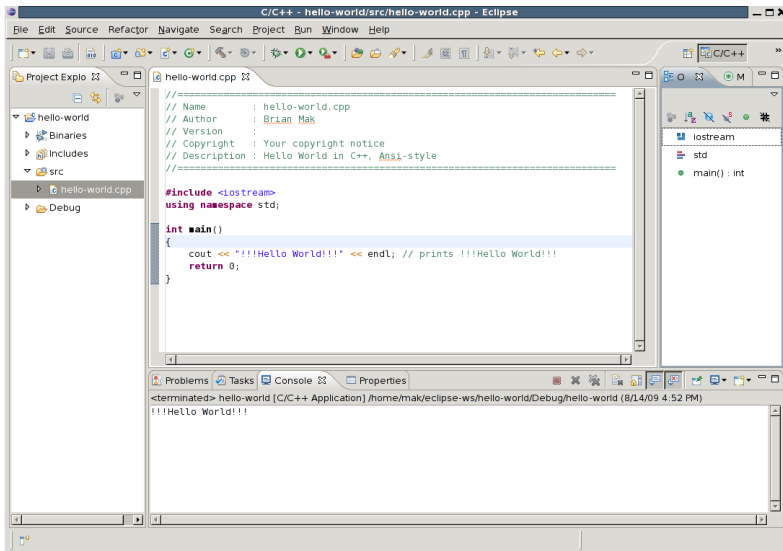
```
g++ hello-world.cpp
```

STEP 4 : Run the program in a terminal (command window):

```
linux:: hello-world  
Hello World!
```




# Eclipse IDE for C/C++

In the lab, you will use Eclipse (similar to MS Visual Studio).



# Program Development using Eclipse



Step 1	<a href="#">create</a> a new project	Eclipse menu
Step 2	<a href="#">write</a> program	Eclipse built-in editor
Step 3	<a href="#">save</a> program	
Step 4	<a href="#">compile</a> program	
Step 5	<a href="#">run</a> program	

## Example: Addition of 2 Numbers

```
/* File: add.cpp */  
#include <iostream>           // Load info of a Standard C++ library  
using namespace std;         // Standard C++ namespace  
  
int main( )                   // Program's entry point  
{  
    /* Major program codes */  
    cout << "123 + 456 = " << 123 + 456 << endl;  
  
    return 0;                 // A nice ending  
}
```



# Main: the Entry Point

- Every program must have exactly one and only one `main()` function.

## Simple Form of the `main` Function

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

## General Form of the `main` Function

```
int main (int argc, char** argv) { ... }
```

(We'll talk about `argc` and `argv` later.)

- Between the braces “{” and “}” are the program codes consisting of zero or more program **statements**.
- Each simple C++ statement ends in a semicolon “;”.

- Use `/* ... */` for **multiple-line comments**.

```
/*  
 * A common program used to demo a new language  
 */
```

- **Single-line comments** start with `//`.

```
// Program's entry point
```

- Comments are just for human to read.
- They will not be translated by the compiler into machine codes.

# #include and Standard C++ Libraries

- `#include` will include information of a **library** — a collection of sub-programs. e.g. `#include <iostream>` gets the information of the **standard C++ library** called **iostream** that deals with I/O:
  - **cin**: to read, e.g., from the keyboard or file
  - **cout**: to print out, e.g., to the screen or file
  - **cerr**: to print error message, e.g., to the screen or file

## Examples

*// endl means "end of a line"*

```
cout << "Einstein:  God does not play dice." << endl;
```

*// You may also break down the message in several lines*

```
cerr << "Error:  "  
    << "There is no stress and tension in HKUST!"  
    << endl;
```

- These library information files are called **header files**.

# #include and User-defined Libraries

- You may also define your *own* library.
- Again you need to use `#include` to include its information into your sub-programs.
- Example:

```
#include "drawing.h"
```

gets the information of a **user-defined C++ library** about drawing.

- By convention, the **header file** of a **user-defined library** ends in `".h"` or `".hpp"`, while **Standard C++ library** header files have **no** file suffix.
- Also by convention, the **header file** of a **user-defined library** is delimited using double-quotes `"..."`, while **Standard C++ library** header files use `< ... >`.

# C++ is a Free Format Language

- Extra **blanks**, **tabs**, **lines** are **ignored**.
- Thus, codes may be indented in any way to enhance **readability**.
- More than one statement can be on one line.
- Here is the same Hello World program:

```
#include <iostream>           /* File: hello-world-too.cpp */  
using namespace std; int main (int argc, char** argv) {  
    cout<<"Hello World!"<<endl;return 0;}
```

- On the other hand, a single statement may be spread over several lines.

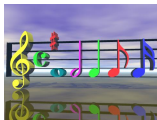
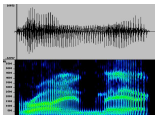
```
cout << "Hello World!"  
    << endl;
```

# Good Programming Style

- Place **each** statement on a line by itself.
- For **long** statements
  - if possible, break it down into several shorter statements.
  - wrap it around with proper indentation (since extra space doesn't matter!)
- Use blank lines to **separate** sections of related codes that together perform some action.
- **Indent consistently**. Use the same indentation for the same block of codes.

# Part II

## Simple C++ Data Types: Integers, Characters, and Strings



# Data Types: Introduction

- The Web has to deal with different **multimedia data**: text, sound/music, image, video, etc., and they can only be read/viewed with different softwares such as MS Notepad, Acrobat Reader, RealPlayer, etc.
- Similarly, a **computer program** has to deal with different **types** of data. In a **programming language**, data are categorized into different **types**.
- Each **data type** comes with a set of **operations** for manipulating its **values**. **Operations** on **basic data types** are built into a **programming language**.



# Integers, Characters, Character Strings

- **Integers**

- Examples: ..., -2, -1, 0, 1, 2, ...
- C++ type name: **int**

- **Characters**

- Examples: 'a', 'b', '4'
- Represent a single character by delimiting it in **single quotes**.
- For special characters, use the escape character **\**. e.g.

<b>'\t'</b>	=	tab	<b>'\n'</b>	=	newline
<b>'\b'</b>	=	backspace	<b>'\0'</b>	=	null character

- C++ type name: **char**

- **Character Strings**

- Examples: "hkust", "How are you?", "500 dollars"
- Character strings are **not** a basic data type in C++.
- They are **sequences** of basic **char** data.

**Note:** There is a **string** library that defines **string** objects which are more than a character string. (More about it later.)

# How Numbers are Represented in Computers: Binary Numbers

- Computer uses **binary** numbers (base 2) to represent data.
- In the **decimal** system:  $423_{10} = 4 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$ .
- In the **binary** system:
  - A digit has only **2** possibilities:  $\{0,1\}$ .
  - Example:  $101_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
  - Thus, the maximum  $N$ -digit number in base 2 =
  - A **binary digit** is aka **bit**.
  - 8 bits = 1 **byte**.  
(smallest amount of data that a computer can “bite” at once.)

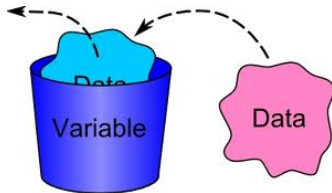
# Relation between Characters and Integers

- In C++, a **char** datum is represented by 1 byte (8 bits).
- **Question**: How many different characters can 8 bits represent?
- Put it in another way, a char datum is encoded by one of the possible 8-bit patterns.
- The most common **encoding scheme** is called **ASCII** (American Standard Code for Information Interchange).
- Since a computer only recognizes bits, a **char** datum may also be **interpreted** as an **integer**!

CHARACTER	ASCII CODE	Integral Value
'0'	00110000	48
'1'	00110001	49
'9'	00111001	57
'?'	00111111	63
'A'	01000001	65
'B'	01000010	66
'Z'	01011010	90
'a'	01100000	97
'b'	01100001	98
'z'	01111010	122

## Part III

# C++ Variables



# Motivation Example: Addition of 2 Numbers Again

## Example: Add 2 Numbers

```
/* File: add.cpp */
#include <iostream>           // Load info of a Standard C++ library
using namespace std;         // Standard C++ namespace

int main( )                  // Program's entry point
{
    /* Major program codes */
    cout << "123 + 456 = " << 123 + 456 << endl;

    return 0;                // A nice ending
}
```

- In this old example, the 2 numbers to be added are hard-coded into the program file.
- Can we write a program that takes 2 arbitrarily numbers to add?

$$f(x) = x^2 + c$$

where

$f$  : name of a function

$x$  : name of a variable

$c$  : name of a constant

In programming languages, these “names” are called **identifiers**.

# Rules for Making up Identifier Names

- Only the following characters may appear in an identifier:

0–9, a–z, A–Z, \_

- The **first** character cannot be a digit (0–9).
- C++ keyword — **reserved words** — are not allowed.
- Examples: amount, COMP2011, \_myname\_
- C++ identifiers are **case-sensitive**: lowercase and uppercase letters are considered **different**.

⇒ hkust, HKUST, HkUst, HKust are **different** identifiers.

- Examples of illegal C++ identifiers:
- Guidelines:
  - use meaningful names. e.g. **amount** instead of **a**
  - for long names consisting of several words, use '\_' to separate them or **capitalize** them. e.g. **num\_of\_students** or **numOfStudents** instead of numofstudents.
  - usually identifiers starting with '\_' are used for **system** variables.

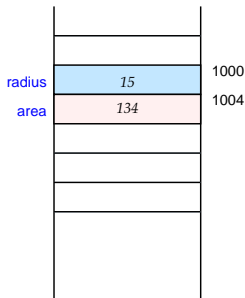
# Reserved Words in C++

asm	auto	bool	break	case
catch	char	class	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	extern
false	float	for	friend	goto
if	inline	int	long	mutable
namespace	new	operator	private	public
protected	register	reinterpret	return	short
signed	sizeof	static	static_cast	struct
switch	template	this	throw	true
try	typedef	typeid	typename	union
unsigned	using	virtual	void	volatile
wchar_t	while			



# Variables

A **variable** is a **named memory location** for a value that we can write to, retrieve from, and manipulate.



- It can be thought of as a container/box for a value.
- A variable must be **declared** and/or **defined** before it can be used.

## Syntax: Variable Definition

<data-type> <identifier> ;

## Examples

```
int radius; // Radius of a circle
int area;   // Area of a circle
```

# Variable Declaration/Definitions

Syntax: Defining **Several** Variables of the **Same** Type at Once

```
<data-type> <identifier1>, <identifier2>, ... ;
```

## Examples

```
int radius, num_of_words;  
char choice, gender, pass_or_fail;
```

- When a variable is **defined**, the compiler **allocates memory** for it.
- The amount of memory is equal to the size of its data type.

**\*\*** *Some books will call this variable declaration. Actually there is a big difference between variable declaration and variable definition. We'll talk about that later. When a variable is defined, it is also declared. The other way is not true.*

## Syntax: Initialize Variables While they Are Defined

`<data-type> <identifier> = <value> ;`

- Several variables of the same type may also be initialized at the same time. e.g.

```
int radius = 10, sum = 0;
```

- A variable may also be initialized by a separate **assignment statement** after it is defined: e.g.

```
int radius;           // Variable definition
radius = 5;           // Initialization by assignment
```

- ANSI C++ does not require compilers to initialize variables.
- Thus, in general, if you do not explicitly initialize variables while you are defining them, their initial contents may be **garbage**.

# Example: Addition of 2 Numbers Using Variables

```
#include <iostream>                                /* File: add-var.cpp */
using namespace std;

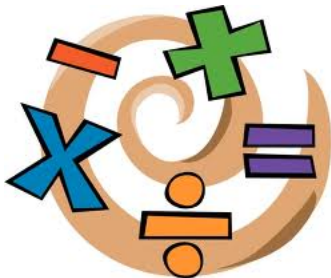
int main( )                                         // Program's entry point
{
    int x, y; // Define 2 variables that will hold the integer values to add

    cin >> x;                                     // You may also shorten the 2 statements into one:
    cin >> y;                                     // cin >> x >> y;

    cout << x << " + " << y << " = " << x+y << endl;
    return 0;                                     // A nice ending
}
```

## Part IV

# Operators



# Assignment Operator

## Syntax: Assignment

<variable> = <value> ;

- In C++, the “=” sign is used to assign a value to a variable; it is the **assignment operator**.

## Examples

```
int a, b, x = 2, y = 3, z = 4;
```

```
a = 10*x;
```

```
b = a - (100*y - 1000*z);
```

```
a = a + b;
```

- Don't try to understand the assignment statement:  

a = a + b;

 using normal math notation, otherwise, it doesn't make sense.
- Nor should you treat it as a boolean relational “equality” sign.

# Arithmetic Operators

Assuming  $x = 100$ ,  $y = 67$ :

OPERATION	OPERATOR	int
unary minus	—	
addition	+	
subtraction	—	
multiplication	*	
division	/	
modulus	%	
increment	++	
decrement	--	

A handwritten long division of 17 by 5. The divisor 5 is on the left, and the dividend 17 is under the division bar. The quotient 3 is written above the bar, and 15 is written below 17. A horizontal line is drawn under 15, and the remainder 2 is written below that line. To the right of the division, there are two annotations: a red arrow points from the quotient 3 to the expression  $17/5$ , with the word "quotient" written in blue below it; and a green arrow points from the remainder 2 to the expression  $17 \% 5$ , with the word "remainder" written in blue above it.

- The **mod** math function is represented as the **%** operator.
- **mod** is used to get the **remainder** in an **integer division**. e.g.

$$\text{mod}(17, 5) = 17 \text{ mod } 5 = 17\%5 = 2$$

(Read as “17 modulo 5.”)

- **Question:** What are the results of  $(-17)\%5$  or  $17\%(-5)$ ?



# Pre- and Post- Increment, Decrement

- The **unary** increment operator **++** add 1 to its operand.
- The **unary** decrement operator **--** subtract 1 from its operand.
- However, there are 2 ways to call them: **pre-increment** or **post-increment**. e.g.

**++x   x++   --x   x--**

- If used **alone**, they are equivalent to:  $x = x + 1$  and  $x = x - 1$ .
- But if used **with** other operands, then there is a big difference:
  - **++x**  $\Rightarrow$  add 1 to x, and use the result for further operation.
  - **x++**  $\Rightarrow$  use the current value of x for some operation, and then add 1 to x.

```
cout << ++x;
```

```
/* same as */
```

```
x = x + 1;
```

```
cout << x;
```

```
cout << x++;
```

```
/* same as */
```

```
cout << x;
```

```
x = x + 1;
```

## Example: %, ++, --

```
#include <iostream>                                     /* File: inc-mod.cpp */
using namespace std;

int main()
{
    int x = 100, y = 100;                                // Variable definitions and initialization
    int a = 10, b = 10, c = 10, d = 10;

    cout << ++x << "\t"; cout << "x = " << x << endl;    // Pre-increment
    cout << y++ << "\t"; cout << "y = " << y << endl;    // Post-increment

    a = ++b; cout << "a = " << a << "\t" << "b = " << b << endl;
    c = d++; cout << "c = " << c << "\t" << "d = " << d << endl;

    cout << 19%4 << endl;                                // Trickiness of the mod function
    cout << (-19)%4 << endl;
    cout << 19%(-4) << endl;

    return 0;
}
```

# Shorthand Assignment Operators

SHORTHAND NOTATION	NORMAL NOTATION
$n \text{ } + = 2$	$n = n + 2$
$n \text{ } - = 2$	$n = n - 2$
$n \text{ } * = 2$	$n = n * 2$
$n \text{ } / = 2$	$n = n / 2$
$n \text{ } \% = 2$	$n = n \% 2$

# Precedence and Associativity

OPERATOR	DESCRIPTION	ASSOCIATIVITY
$-$ $++$ $--$	minus increment decrement	RIGHT
$*$ $/$ $\%$	multiply divide mod	LEFT
$+$ $-$	add subtract	LEFT
$=$	assignment	RIGHT

Example:  $1/2 + 3 * 4 = (1/2) + (3 * 4)$

because  $*$ ,  $/$  has a **higher precedence** over  $+$ ,  $-$ .

- **Precedence rules** decide which operators run first.
- In general,

$$x \text{ } P \text{ } y \text{ } Q \text{ } z = x \text{ } P \text{ } ( y \text{ } Q \text{ } z )$$

if operator  $Q$  is at a higher precedence level than operator  $P$ .

# Associativity: Binary Operators

Example:  $1 - 2 + 3 - 4 = ((1 - 2) + 3) - 4$   
because  $+$ ,  $-$  are **left associative**.

- **Associativity** decides the grouping of operands with operators of the *same* level of precedence.
- If **binary** operator  $P$ ,  $Q$  are of the **same** precedence level
  - if operator  $P$ ,  $Q$  are both **right associative**, then

$$x \ P \ y \ Q \ z = x \ P \ (y \ Q \ z)$$

- if operator  $P$ ,  $Q$  are both **left associative**, then

$$x \ P \ y \ Q \ z = (x \ P \ y) \ Q \ z$$

# Cascading Assignments

- C++ allows assigning the same value to multiple variables at once.

## Examples

```
int w, x, y, z;
```

```
y = z = 5;
```

*// Same as y = (z = 5);*

```
w = x = y + z;
```

*// Same as w = (x = (y+z));*

# Expression and Statement

- An **expression** has a value which is the result of some operation(s) on its(theirs) operands.
- Expression examples:

4          x - y          2 - a - (b \* c)

- A **statement** is a sentence that acts as a command.
  - It does not have a value.
  - It always ends in a **';**'.
- Statement examples:
  - **Input** statement: `cin >> x;`
  - **Output** statement: `cout << x;`
  - **Assignment** statement: `x = 5;`
  - **Variable** definition: `int x;`