UEE1303(1070) S'12 Object-Oriented Programming in C++

Lecture 01: *C/C++ Overview and OOP by Example*

Learning Objectives

You should be able to review/understand:

- Fundamentals of C/C++ programming
  - Datatypes and variables
  - Flow of control: condition + loops
  - Functions
  - Arrays and strings
- C++ as an object-oriented language
  - Basic concept of object-orientation
  - Your first useful C++ program

C++ Program 101: Hello, World!

```
//hello.cpp
#include <iostream>

int main(int argc, char ** argv)
{
    std::cout << "Hello, World!"
              << std::endl;
    return 0;
}
```

- The above C++ code
  - prints "Hello, World!" on the screen
  - exhibits important concepts of C++ format

Comments

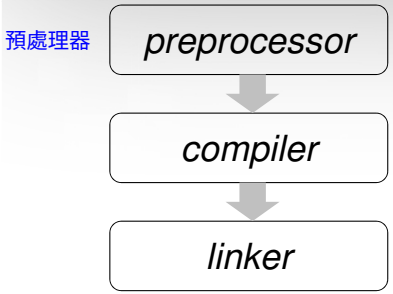
- Comments are the messages for the programmer only and ignored by the compiler
- Two ways in C++ to delineate a comment:
  - *Line comments*: two slashes (`//`) followed by whatever on that line

    Ex: `//program-01.cpp`
  - *Block comments*: embraced with `/*` and `*/` and may (usually) span multiple lines

    Ex:
    ```
    /* this is a multiline
     * C-style comment.
     */
    ```

Preprocessor Directives

- Building a C++ program is a 3-step process



– recognize *meta-information* about the code

– translate source code into *machine-dependent* object code

– link together all individual object files into an *application*

- Preprocessor aims at ***directives*** which starts with the `#` character
  - Ex: `#include <iostream>`

Most Common Directives

- `#include <[file]>`
  - inserts the specified *header* file into the code at the location of directives
- `#define [key] [value]`
  - replaces the key with the value everywhere
  - define *constants* or *macros*
- `#ifdef [key]` `#ifndef [key]` `#endif`
  - *includes*/*omits* the code within `ifdef` / `ifndef` and `endif` blocks based on if `key` has been defined before with `#define`
  - protects against *circular includes*

`main()` function

- `int main(int argc, char ** argv)`
  - where the program starts
  - An `int` is returned ⇨ indicate the result status; typically return 0
  - `argc` gives the number of arguments (integer) passed to the program
  - `argv` contains those arguments (C-Strings)
- Ex: `>./prog 5 4.4 test1.txt`
  - `argc = 4`
  - `argv[0]` is "`prog`", `argv[1]` is "`5`", `argv[2]` is "`4.4`" and `argv[3]` is "`test1.txt`"

atoi(): array to int

I/O Streams (1/2)

- `printf()` can still be used in C++, but a much better input/output facility is provided.
- `std::cout` corresponds to the user console or standard out
  - `<<` operator tosses data down to the pipe
  - allows multiple data of varying types sequentially in one or more lines

Ex:
```
std::cout << "Today is" << 2 << "-"
          << 21 << std::endl;
```

  - `std::endl` is an end of line character ⇨ output everything in the pipe and move to next line

I/O Streams (2/2)

- Common escape characters used with I/O
  - `\n`: new line
  - `\r`: carriage return
  - `\t`: tab
  - `\\`: the backslash character
  - `\"`: quotation mark
- `std::cin` accepts to the input from the user
  - Use `>>` operator with an input stream
  - User input can be tricky since you can never know what kind of data that a user input

Namespaces

- Namespaces solve the naming conflicts between different pieces of code
- Ex: Having your own `foo()` and `foo()` from a third-party library
  - Compiler does not know which to call

```
//myspace.h
namesapce mycode {
    void foo() {
        ...
    }
}
```

```
//usenamespaces.cpp
#include "myspace.h"
using namespace mycode;
int main(){
    foo();//use mycode::foo()
}
```


New Program 101

```
//newhello.cpp
#include <iostream>
using namespace std;

int main(int argc, char ** argv)
{
    cout << "Hello, World!" << endl;
    return 0;
}
```

- `using` directive can refer to a particular item , ex:

```
using std::cout;
...
cout << "Hello, World!" << std::endl;
```


Variables and Datatypes

- C++ allows variables to be *declared anywhere* and hereafter uses them in the current block
- Datatypes: a set of *values* and *operations* that can be applied to these values
- *Built-in* datatypes: an integral part in C++
  - also known as *primitive* types
  - require no external code
  - consist of basic numerical types
  - majority of operations are symbols (e.g. +,-,*,>,<…)

Built-in Datatypes

Type	Description	Usage
`int`	Positive and negative integers	`int i = 7;`
`short/long`	Short/long integers	`short s = 13;` `long l = -55;`
`unsigned`	Limits the preceding types to ≥0	`unsigned int i = 2;` `unsigned long l = 23;`
`float` `double`	Floating-point and double-precision values	`float f = 7.2` `double d = 7.2`
`char`	Single characters	`char ch = 'm';`
`bool`	True or false	`bool b = true;`

Type Casting and Coercion

- Casting: *explicitly* convert the data type of a value to another data type
  - method 1: most common used; from C

```
bool someBool = (bool)someInt;
```

  - method 2: naturally but rarely seen

```
bool someBool = bool(someInt);
```

  - method 3: verbose but clean

```
bool someBool = static_cast<bool>(someInt);
```

有時候在編譯轉型,
有時候在執行時轉型

- Coercion: *automatically* casting by the compiler
  - Ex:

```
int someInt = someDouble;
```

強制轉型

Operators

- Arithmetic: `+, -, *, /, %`
  - Shorthand: `+=, -=, *=, /=, %=`
  - Increment/decrement: `++, --`
- Relational: `==, !=, >, >=, <, <=`
  - used to compare operands
  - Format: `x <op> y`
- Logical: `&&, ||, !` ⇨ compound conditions

複合條件式

  - Bitwise: `&, &=, |, |=, <<, <<=, >>, >>=, ^, ^=`
  - Famous swap macro: //only for integers

```
#define swap(x,y) (x^=y,y^=x,x^=y)
```


Precedence of Operations

Operator	Associativity
`!(unary), -, ++, --`	right to left
`*, /, %`	left to right
`+, -`	left to right
`<, <=, >, >=`	left to right
`==, !=`	left to right
`&&`	left to right
`||`	left to right
`=, +=, -=, *=, /=`	right to left

User-defined Datatypes

- Enumerated type: the sequence of numbers
  - –Format: `enum typename {id1,id2,id3,…};`
  - –where `id1 < id2 < id3 < …`
- `struct` type: encapsulate one or more existing types into a new one 把很多不同型態的東西包成一種型態

```
struct tagname
{
    type_1   member_1;
    ...
    type_n   member_n;
};
```

name

body

  - –access members by dot operator (.)

Flow of Control: Selection (1/2)

- ***If/Else*** statements

```
if (i>4) {
    // do something
} else if (i>2) {
    // do something else
} else {
    // do something else
}
```

- ***Ternary*** operator

```
cout << ((i>2)? “yes” : “no”);
```


Flow of Control: Selection (2/2)

- ***Switch*** statements

```
switch (menu) {
    case item1:
        //do something
        break;
    case item2:
    case item3:
        //do something
        break;
    ...
    default:
        //do something
        break;
}
```

Each item needs to be a constant value

Flow of Control: Repetition (1/2)

- ***While*** loop

```
while (i < 5) {
    cout << “good!” << endl;
    i++;
}
```

- ***Do/While*** loop

```
do {
    cout << “good!” << endl;
    i++;
} while (i < 5);
```

- Loop control
  - –`break` ⇨ get out of the loop immediately
  - –`continue` ⇨ return to the top of the loop

後面不做, 從頭在開始做

Flow of Control: Repetition (2/2)

- ***For*** loop

```
for (int i = 0; i < 5; i++) {
    cout << "good!" << endl;
}
```

  - the most verbose
  - but also the most convenient
- Can convert any ***for*** loop into a ***while*** loop

```
int i = 0;
while (i < 5) {
    cout << "good!" << endl;
    i++;
}
```


Functions (1/2)

- Functions are building blocks of programs
  - Available for other code to use
  - *Declaration* (in header files) + *Definition* (in source files) + *Call* (used in the code)
- **Declaration** (a.k.a. *prototype* or *signature*)
  - how the function can be accessed
  - syntax: `<type> FnName(<parameters>);`
  - placed before any calls in *declaration space* of main() or in *global space*
- Ex:

```
double totalCost(int num, double
price);
```


Functions (2/2)

- **Definition** is the *implementation* of function
  - The *link* stage searches the right function

```
double totalCost(int num, double
price) {
    return (num*price*1.05);
}
```

- **Call**s to the function in the program
  - pass *constants* or *variables* as arguments
  - Ex 1:

```
totalCost(8, 9.5);
```

  - Ex 2:

```
totalCost(inum, 9.5);
```

  - Ex 3:

```
totalCost(inum, dprice);
```


Inline Functions

- An **inline** function 適用於小的, 用的程式
  - where compiler performs *inline expansion*
  - reduce the program execution time
  - improve over macros
- Good candidates are *small*, *frequently* called functions

```
inline float area(float len, float wid) {
    return (len*wid);
}
```

- Not accepted typically if the inline function
  - contains loops, `switch`, `goto` or static variables
  - is too large or recursive

Arrays

- An array are a *collection of data* of same type
  - in C++, the size must be a constant
  - C++ allows multidimensional arrays
  - three-dimensional or higher is rarely used
- An example of Tic-Tac-Toe board

```
char ticTacToe[3][3];
for (int idx=0; idx<3; idx++) {
    for (int jdx=0; jdx<3; jdx++) {
        ticTacToe[idx][jdx] = 'x';
    }
}
```

  - The first element is always at *position 0*
  - The last element is at *position (size-1)*

Memory in C++ (1/2)

- Memory in C/C++ applications consists of
  - *Stack*: like a deck of cards; last-in first-out
  - *Heap*: like a pile of bits
- A function has its own *stack frame*
  - isolate memory space from each other
  - if the current function `f1()` calls `f2()`, a new frame is put on top of the `f1()`'s frame
  - variable `var` inside `f1()` cannot be changed or seen by `f2()`
  - Once `f2()` is done running, all variables inside `f2()` no longer take up memory

Memory in C++ (2/2)

- Stack frame size is predetermined ⇨ cannot declare an array with a variable size
- The following code will not compile *safely*

```
int arraysize = 10;
int MyVariableSizedArray[arraysize];
```

  - because the entire array must go on the stack
- The *heap*, independent of the stack:
  - is less structured than the stack
  - has variables even when the function in which they were declared has completed
  - can add/modify variables at any time

Dynamically Allocated Arrays

- Place the array with the size specified at runtime in the heap (dynamic memory)
  - need to declare a pointer first

```
int * MyVariableSizedArray;
```

  - initialize the pointer to `new` heap memory

```
MyVariableSizedArray = new int [arraySize];
```

  - work like a regular stack-based array

```
MyVariableSizedArray[3] = 2;
```

  - `delete` the array from the heap when done

```
delete [] MyVariableSizedArray;
```

Working with Pointers

- Other uses of a pointer from heap memory
  - points to a *single value* that can be accessed by dereferencing (*)

```
int * MyIntPtr = new int;
*MyIntPtr = 8;
```

  - points to a *stack variable* or another pointer + use addressing-of (&)

```
int i = 8;
int * MyIntPtr = &i;
```

  - points to a *structure variable* + use dereferencing (*) and an arrow (->)

```
EmployeeT * Worker = getEmployee();
cout << Worker->salary << endl;
```


Strings from C

- **C-Style** Strings (a.k.a. C-Strings)
  - is a *character array* ending with '\0'

```
char cString[20] = "Hello, World!";
char * ptrString = "Hello, World!";
```

  - allocate 20 characters on stack for `cString` with *random* values in position 14 to 19
  - allocate *just enough* stack memory (14 characters) for `ptrString`
- C language provides many *standard functions* for string manipulation in `<cstring>`
  - `strcpy` (copy strings), `strlen` (return string length), `strstr` (search string), and etc.

New C++ Strings

- C++ includes a more flexible string type
  - described by the `<string>` header file living in the "`std`" package
  - Like `strcat()` in C to concatenate two C-style strings

```
string str1 = "Hello";
string str2 = "World";
string str3 = str1 + ", " + str2 + "!";
```

  - in C, == operator does not work to compare two C-style strings, but works in C++

```
cout <<(str4 == "Monday")? "yes" : "no"
     << endl;
```


Procedural Programming

- **Procedural programs:** consist of a series of procedures (building blocks, functions) that take place one after the other
- Common procedural languages:
  - COBOL
  - BASIC
  - FORTRAN
  - Pascal
  - C/C++
- Procedural programming techniques have evolved into object-oriented techniques

Object-Oriented Programming (OOP)

- C is completely *procedural* ⇔ C++ mixes both *object-oriented* + *procedural* programming
  - –C++ covers whatever C can do
- Object-oriented programming (OOP) is a programming paradigm that
  - –uses objects to design programs and
  - –features (1) *encapsulation* (2) *inheritance* and (3) *polymorphism*
- For OOP,
  - –Object = data fields + methods
  - –Program = object + object + … + object

First OO Program in C++ (1/6)

- Declare a class `AirTicket` for airline tickets

```
//AirTicket.h
#include <string>
class AirTicket {
    private: //two data members
       std::string name;
       int miles;
    public: //six methods
       AirTicket();
       ~AirTicket();
       std::string getName();
       void setName(std::string inName);
       void setMiles(int inMiles);
       int calculatePrice();
};
```


First OO Program in C++ (2/6)

- Define methods: `constructor` and `destructor`

```
//AirTicket.cpp (part 1)

#include <iostream>
#include "AirTicket.h"
using namespace std;

//constructor
AirTicket::AirTicket() {
    name = "unknown";
    miles = 0;
}
//destructor
AirTicket::~AirTicket() {
    //actually nothing to do
}
```


First OO Program in C++ (3/6)

- Define `getName`, `setName` and `setMiles`

```
//AirTicket.cpp (part 2)
//getName
string AirTicket::getName() {
    return name;
}
//setName
void AirTicket::setName(string inName) {
    name = inName;
}
//setMiles
void AirTicket::setMiles(int inMiles) {
    miles = inMiles;
}
```

First OO Program in C++ (4/6)

- Define method `calculatePrice`

```
//AirTicket.cpp (part 3)
//calculatePrice
int AirTicket::calculatePrice() {
    int rPrice = 0;

    //10% off if mileage > 10000
    if (miles > 10000)
        rPrice = (int) (miles * 0.095);
    else
        rPrice = (int) (miles * 0.1);

    return rPrice;
}
```


First OO Program in C++ (5/6)

- `main()` program to use class `AirTicket`
- create a *stack*-based object `tkt1`

```
//main.cpp (part 1)
#include <iostream>
#include "AirTicket.h"
using namespace std;

int main(int argc, char ** argv) {
    //declare a stack variable
    AirTicket tkt1;
    //initialize and compute price
    tkt1.setName("Peter Woods");
    tkt1.setMiles(25000);
    cout << tkt1.getName() << "pays"
         << tkt1.calculatePrice() << endl;
```


First OO Program in C++ (6/6)

- Create a *heap*-based object `tkt2`

```
//main.cpp (part 2)
    //declare a heap variable
    AirTicket * tkt2 = new AirTicket;
    //initialize and compute price
    tkt2->setName("Laura Clinton");
    tkt2->setMiles(3000);
    cout << tkt2->getName() << " pays"
         << tkt2->calculatePrice() << endl;
    //delete the variable when done
    delete tkt2;

    return 0;
}
```


Compiling with g++

```
>ls
AirTicket.cpp  AirTicket.h  main.cpp
>g++ -c AirTicket.cpp
>g++ -c main.cpp
>g++ -o prog AirTicket.o main.o
>ls
AirTicket.cpp AirTicket.h AirTicket.o
main.cpp main.o prog
>./prog
Peter Woods pays 2375
Laura Clinton pays 300
>
```

Simple makefile

- Prepare a file named *makefile*

```
AirTicket.o: AirTicket.cpp
 tab g++ -c AirTicket.cpp
main.o: main.cpp
 tab g++ -c main.cpp
prog: AirTicket.o main.o
 tab g++ -o prog AirTicket.o main.o
```

- Run *make* on terminal

```
>make
>./prog
Peter Woods pays 2375
Laura Clinton pays 300
>
```


Summary (1/2)

- Review basic C/C++ programming
  - Comments: line vs. block
  - Preprocessor directives
  - Datatypes: built-in and user-defined
  - Variables: local vs. global
  - Flow of control: selection (If/Else, switch ternary) + repetition (for, while, do/while)
  - Functions: declaration, definition and call
  - Arrays: stack vs. heap, pointers
  - strings: C-style strings vs. class string

Summary (2/2)

- C++ as an object-oriented language
  - Procedural vs. object-oriented
  - Definition of object-oriented programming (OOP) and three key features
- First OOP in C++
  - Declare a class `AirTicket`
  - Define six methods including constructor, destructor, `getName()`, `setName()`, `setMiles()` and `calculatePrice()`
  - Create a stack-based object and a heap-based object in `main()`