EECS 204002
Data Structures 資料結構
Prof. REN-SONG TSAY 蔡仁松 教授
NTHU

C++ QUICK REVIEW

www.tutorialspoint.com/cplusplus/cpp_quick_guide.htm

1.2.3

History of C++

- C is widely-used in industry because:
  - **Efficient**: Support low-level features which utilize hardware more efficiently.
  - **Flexible**: Can be used to solve problem in most application areas.
  - **Available**: C Compilers are readily available for most platforms.
- C++ is an enhanced version of C
- C++ = Object-oriented paradigm + C

Review of C

Auto and Static Variables

- Automatic
  - The variable will lose its storage (and value) when the program exits the block
- Static
  - The variable is initialized only once.
  - The value can be changed during run time.
  - The value remains until the program exits.

Quiz 1: Static Variables

Show the output

If take out "static" what will be the output?

```
#include <stdio.h>

/* function declaration */
void func(void);

static int count = 5; /* global variable */

main() {
   while(count--) {
      func();
   }
   return 0;
}

/* function definition */
void func( void ) {
   static int i = 5; /* local static variable */
   i++;
   printf("i is %d and count is %d\n", i,
count);
}
```


Quiz 2: Block Scope

Show the output

```
int main()
{
  {
     int x = 10, y  = 20;
     {
        printf("x = %d, y = %d\n", x, y);
        {
           int y = 40;

           x++;  // Changes the outer block variable x to
11
           y++;  // Changes this block's variable y to 41

           printf("x = %d, y = %d\n", x, y);
        }

        printf("x = %d, y = %d\n", x, y);
     }
  }
  return 0;
}
```

Example: extern global variable

```
//file :   main.cpp

int   x  = 100;

//static global only within a file
static  int  size=10;

int main()
{
   func();

   return  0;
}
```

```
// file:    func.cpp

// auto global defined in other file
extern  int  x;

void   func () {
     printf("x=%d\n", x );
}
```


typedef: a new name for a type

```
typedef  int     Integer ;
Integer  x;   // =  int x

typedef struct Books {
   char title[50];
   …
} Book;

int main( ) {
   Book book;
}
```


typedef Function Pointer

```
typedef int (*t_somefunc)(int, int);

int product(int u, int v) {
  return u*v;
}

t_somefunc afunc = &product;
...
int x2 = (*afunc)(123, 456);
// call product() to calculate
123*456
```

1.2 Object Oriented Design

From C to C++

- Data type and operations in C
  - `int i,j; x = i+j;`
- Extend to user-defined data types?

```
typedef struct {
  double long; // Longitude of building
  double lati; // Latitude of building
  char* owner; // owner of the building
} building;
building A,B;
```

  - `A+B`? (Data abstraction)
- How to describe a **house** as a **building** with an additional attribute of a number of **rooms**? (Object Inheritance)

1.2.1 C: Algorithmic Decomposition

- Software is viewed as a process and decomposed into steps.
- Steps are implemented as functions
  - e.g., in C or Pascal
- Data structures are a secondary concern
  - Data is visible and accessible to potentially all steps
  - Unclear by which function the data is manipulated
  - No way to prevent irrelevant code to access the data
- **Difficult to reuse code**

1.2.1

C++: Object-Oriented Decomposition

- Software is viewed as a set of well-defined objects that interact with each other to solve the problem.
  - Objects are entities that contain data (local state) and operations (functions) to perform computation, and are instances of some classes (vs struct in C).
  - Focus on the design of data. ← **Data Structure!**
- Benefits
  - Intuitive to develop software
  - Easy to maintain
  - High reusability and flexibility

struct V.S. class

- In C, use "struct" to define custom data types
- In C++ and OOP, introduces "class" to define custom data types
  - Each class contains NOT just data but also "operations"
    - With concrete data representation of the class object.
    - A set of operations to manipulate object data.
    - Other operations or codes cannot access the objects.

1.2.2

Object-Oriented Programming

- C++ fully supports object-oriented programming, including the four pillars of object-oriented development −
- Encapsulation
- Data hiding (abstraction)
- Inheritance
- Polymorphism (多型性)

The Object in OOP

- Contains data and procedural elements (functions)
- Is the basic unit for computation
- Is a fundamental building block
- Each object is an instance of some type (class)
- Classes are related to each other by inheritance relationships

1.3

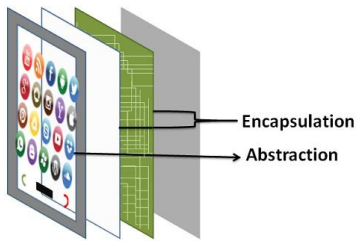
Data Abstraction and Encapsulation

Data Abstraction	Data Encapsulation
• User manual only tells what the player and buttons will do (**Specification**) • No need to know how the machine does it (**Implementation**)	• Interact only through buttons (PLAY, STOP and PAUSE) • Do not interact with internal circuitry • Internal representation (circuitry) is hidden

Abstraction & Encapsulation

- Data Abstraction: separate interface (specification) from implementation.
  - Public methods
  - Private data
    - Or **protected** data for heirs
- Data Encapsulation (Information Hiding): conceal the implementation details. E.g. "sort" can have many different implementations.
- Advantages:
  - Simplification of software development
  - Easy to test and debug

Illustrate the Difference



https://techdifferences.com/difference-between-abstraction-and-encapsulation.html

1.4

Basic of C++

C++ Basics

- **Object**: a specific dog
  - States: color, name, breed …
  - Behaviors: wagging, barking, eating …
- **Class** − describes the behaviors/states that object of its type support.
- **Methods** − each method describes a behavior (function).
- **Instance Variables** − Each variable with a value describes a state.

1.4.1

Program Organization

- Header files (*.h) store declarations
- Source files (*.cpp) store source code

```
#ifndef _HELLO_WORLD_H_
#define _HELLO_WORLD_H_

void Hello_World(void);

// insert other declarations here
// …

#endif
```

System-defined header

```
#include <iostream>
#include <Hello_World.h>

void Hello_World(void)
{
  std::cout << "Hello" << std::endl;
}
```


1.4.2

Scope in C++

Local scope

A name declared in a block

```
void Hello_World(void)
{
    int a = 2;
    {
        int b = 3;
    }
}
```

Class scope

Declaration associated with a class definition

```
class Obj
{
    Obj(void) {m_a=2; m_b=3; }

    int m_a;
    int m_b;
};
```

1.4.2

Scope in C++

- Namespace scope

```
namespace Yahoo{
  char* homePageURL;
}

//access
Yahoo::homePageURL=...
```

```
namespace Google{
  char* homePageURL;
}

//access
Google::homePageURL=...
```

http://www.cplusplus.com/doc/tutorial/namespaces/

1.4.2

Namespace

- A mechanism to group related variables and functions
- Access the members using scope operator "::"
- For the repeated usage of members, use the "using" declaration.
- Avoid namespace pollution.

```
namespace Google{
  char* homePageURL;
}

//access
Google::homePageURL=...

using namespace Google;
//access
homePageURL=...
```


1.4.2

Scope in C++

A variable is uniquely identified by its scope and its name

- Scenario 1: What if a local variable reuses the name of a global variable?

  Ans: use scope operator ::
- Scenario 2: A global variable is defined in file1.cpp, but used in file2.cpp.

  Ans: as in C, use **extern**.

1.4.2

Quiz: show the result

```
#include <iostream>
using namespace std;

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}
```

```
int main () {
  using first::x;
  using second::y;
  cout << x << '\n';
  cout << y << '\n';
  cout << first::y << '\n';
  cout << second::x << '\n';
  return 0;
}
```


1.4.3

Same as in C, except ::, new, delete, throw, …

Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right
2	++ -- () [] . ->	Suffix/postfix increment and decrement Function call Array subscripting Element selection by reference Element selection through pointer	
3	++ -- + - ! ~ (*type*) * & sizeof new, new[] delete, delete[]	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT Type cast Indirection (dereference) Address-of Size-of Dynamic memory allocation Dynamic memory deallocation	Right-to-left
4	.* ->*	Pointer to member	Left-to-right
5	* / %	Multiplication, division, and remainder	
6	+ -	Addition and subtraction	
7	<< >>	Bitwise left shift and right shift	
8	< <= > >=	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	
9	== !=	For relational = and ≠ respectively	
10	&	Bitwise AND	
11	^	Bitwise XOR (exclusive or)	
12	|	Bitwise OR (inclusive or)	
13	&&	Logical AND	
14	||	Logical OR	
15	?: = += -= *= /= %= <<= >>= &= ^= |=	Ternary conditional Direct assignment (provided by default for C++ classes) Assignment by sum and difference Assignment by product, quotient, and remainder Assignment by bitwise left shift and right shift Assignment by bitwise AND, XOR, and OR	Right-to-left
16	throw	Throw operator (for exceptions)	
17	,	Comma	Left-to-right

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

1.4.4

Initialization of Variables

```
#include <iostream>
using namespace std;

int main ()
{
  int a=5;          // initial value: 5
  int b(3);         // initial value: 3
  int result;       // initial value undetermined

  a = a + 2;
  result = a - b;
  cout << result;

  return 0;
}
```

http://www.cplusplus.com/doc/tutorial/variables/

1.4.4

Data Declaration in C++

- Constant variables (**const**):
  - Fixed values during their lifetime.
  - Must be initialized at the time declared
- Enumeration types (**enum**):
  - Declaring a series of integer constants

```
const int MAX = 500;
enum semester {SUMMER=0, FALL, SPRING};
```


1.4.4

Pointer and Reference

- Pointer (*****):
  - the memory address of an object

```
int i = 25;
int* np; np = &i;
```

- Reference type (**&**):
  - an alternative name to an object (C++ only).

```
int i=5;
int& j=i;
i = 7;
cout << j << endl;
```


1.4.5

Comments in C++

Single line comment:

```
// ooxx
```

Multiple Line comment:

```
/*
   ooxx
*/
```

1.4.6

I/O in C++

C

```
#include <stdio.h>

int main(){
int x=100, y=99;

printf("%d%d\n",x,y);
scanf("%d %d", &x , &y);

return 0;}
```

C++

```
#include <iostream>
using namespace std;
int main(){
int x=100, y=99;

cout << x << y << endl;
cin >> x >> y ;

return 0; }
```


1.4.6

I/O in C++

- Include the system-defined header file: #include <iostream>
  - need to specify namespace "**std**"
- Keyword "**cout**" + "**<<**" operator:
  - output to the standard output device.
- Keyword "**cin**" + "**>>**" operator:
  - input a value to an object.
- Keyword "**endl**" :
  - Inserts a new-line('\n') character.

http://www.cplusplus.com/reference/ostream/endl/?kw=endl

1.4.6

File I/O in C++

- Include the system-defined header file: #include <fstream>
  - Namespace is **ios**
- Declare file objects using the following keywords
  - **ofstream**: write to file
  - **ifstream**: read from file
  - **fstream**: read to/write from file
- Specify the I/O mode when opening the file
  - ios::out, ios::in, …etc
- Use shift operator "**<<**" and "**>>**" to write to and read from file.

1.4.6

File I/O in C++

```
#include <iostream>
#include <fstream>
using namespace std;

int main(){

ofstream outFile("my.out", ios::out);
if(! outFile) {
   cout << "cannot open my.out" << endl;
   return 1;
}

int n = 50; float f = 20.3;
outFile << "n: " << n << endl;
outFile << "f: " << f << endl;

return 0; }
```

```
#include <iostream>
#include <fstream>
using namespace std;

int main(){

ifstream inFile("my.out", ios::in);
if(! inFile) {
   cout << "cannot open my.out" << endl;
   return 1;
}

Int x;
while (! inFile.eof(){
    inFile >> x;
    cout << x << endl;
}
return 0; }
```


1.4.6

I/O Operator Overloading

ostream& **operator <<**(**ostream** &os, const dataType &obj);

```
Class Ferrari
{
  public:
  int W, H, L
  char* mod
}
```

```
#include <iostream>
using namespace std;
```

```
Car info:
***********
ari 250 GTO
.60
21
33
******
```


1.4.6

I/O Operator Overloading

```
using namespace std;
ostream& operator <<(ostream &os, const
Ferrari &obj)
{
    os << "***********" << endl;
    os << "Ferrari " << obj.model << endl;
    os << "W=" << obj.W << endl;
    os << "H=" << obj.H << endl;
    os << "L=" << obj.L << endl;
    os << "***********" << endl;
}
```

1.4.7

Functions in C++

- Function declaration(function prototype)

```
int sum( int , int );
```

- Function definition

```
int sum( int a, int b){
    return a+b;
}
```

- Inline function: compiler replaces each function call using its function body.

```
inline int sum( int a, int b){
    return a+b;
}
```

http://www.cplusplus.com/doc/tutorial/functions/

1.4.8

Parameter Passing by Value

- Call by **value**

```
int special_add(int a , int b)
{
        a = a+5;
        return a+b;
}
```

- Object's value is copied into function's local storage (storage overhead).
- Any change in 'a' and 'b' won't modify the original copies.

1.4.8

Parameter Passing by Pointer

- Call by **pointer**

```
void swapnum(int *a , int *b){
        int temp=*a;
        *a=*b;
        *b=temp;
}
```

- Any change in '*a' and '*b' will modify the original objects

1.4.8

Parameter Passing by Reference

- Call by **reference**

```
void swapnum(int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
}
```


1.4.8

Passing const arguments

- The referenced arguments cannot be modified
- Any attempt for modification will cause a compile error.

```
void func1(const dataType& a)
{
    a = …;  // ← Compile time error!
}
```


1.4.9

Function Overloading in C++

- In C++, we can have the following functions:

```
int  Max(int, int);
int  Max(int, int, int);
int  Max(int*, int);
int  Max(float, int);
int  Max(int, float);
```

- In C, it's impossible to define two functions of same function name.

1.4.9

Function Overloading by Signatures

- C defines a **function signature** by function name**.**
- C++ defines a **function signature** by
  1. Function name
  2. Type & number of parameters
  3. Order of parameters

Polymorphism in C++

- Occurs when there is a hierarchy of classes and they are related by inheritance.
- A **virtual** function defined in a base class is dynamically linked to the version in a derived class

An Example

```
class Shape {
  protected:
    int width, height;

  public:
    Shape( int a = 0, int b = 0){
      width = a;   height = b;
    }
    virtual int area() = 0;
};
```

```
class Rectangle: public Shape {
  public:
    Rectangle( int a = 0, int b = 0):
Shape(a, b) { }
    int area () {
        return (width * height);
    }
};

class Triangle: public Shape {
  public:
    Triangle( int a = 0, int b =
0):Shape(a, b) { }
    int area () {
        return (width * height / 2);
    }
};
```

1.4.11

Dynamic Memory Allocation in C++

- Dynamic Memory Allocation in C
  - malloc, delete, realloc, memset, memcopy
  - Causes memory leak and memory fragmentation problems
- New dynamic memory allocation mechanism
  - Use keywords "new" and "delete"
  - The "new" operator in C++ is more powerful than "malloc" in C.
  - Use 'delete' for pointer generated by 'new'.

1.4.11

Dynamic Memory Allocation in C++

- C

```
#include <cstdio>

int main () {

  int * x  = (int*) malloc ( sizeof(int) );
  free(x);

  return 0;
}
```

- C++

```
#include <iostream>

int main () {

  int * y =  new int ;
  delete   y ;

  // allocate an int array.
  int  * data  = new  int [10];
  /* make sure you use 'delete' for
  pointer generated by 'new'. */
  delete  [] data ;

  return 0;
}
```

http://www.cplusplus.com/reference/new/operator%20new[]/

1.4.12

Handling Exceptions

- **try-catch** block
  - Exceptions are thrown by a piece of code within the **try** block.
  - Each try block is followed by zero or more **catch** blocks.
  - Each **catch** block is visited sequentially until the matched block is found.
  - Each **catch** block has a parameter whose type determines the type of exception to be caught.
- **catch** (**char*** e){}
  - Catch exceptions of type char*.
- **catch** (**bad_alloc** e){}
  - Catch exceptions of type **bad_alloc** (system-defined type).
- **catch** (**…**){}
  - Catch all exceptions regardless of types.

http://www.cplusplus.com/doc/tutorial/exceptions/

1.4.12
P1.6

Example

```
int DivZero(int a, int b, int c){
  if(a <= 0 || b <=0 || c <= 0)
      throw "All parameters should be > 0";
  return a+b*c+b/c;
}

int main () {
  try {
    cout << DivZero(2, 0, 4) << endl;
  }
  catch (char* e) {
          cout << "The parameters to DivZero were 2, 0, 4" << endl;
          cout << "An exception has been thrown" << endl;
          cout << e << endl;
  } catch (…) {
          cout << "An unknown exception has been thrown" << endl;
  }
  return 0;
}
```