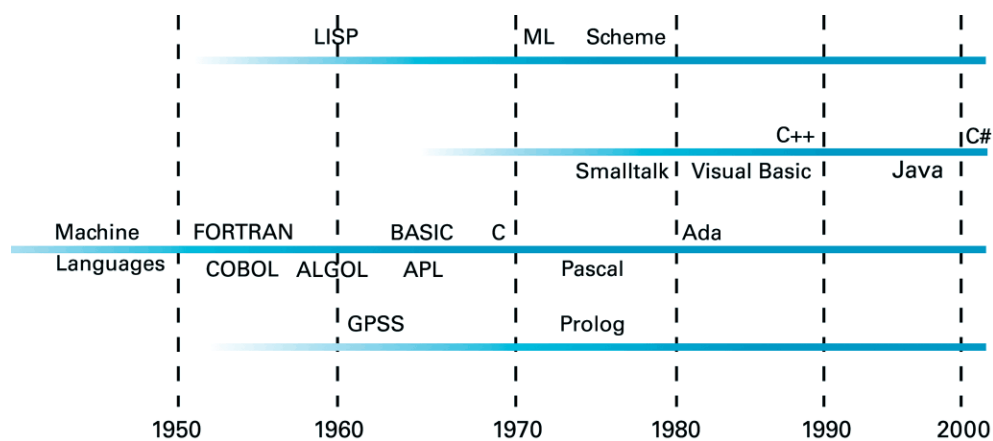# C++ Review

Yi-Shin Chen

Institute of Information Systems and Applications

Department of Computer Science

National Tsing Hua University

yishin@gmail.com

Evolution Of Programming Paradigms

Concept of Object-Oriented

Algorithmic Decomposition vs. Object-Oriented Decomposition

- Algorithmic Decomposition
  - Software decomposed into steps
  - Steps are implemented as functions
    - E.g. C or Pascal
  - Data structures are a secondary concern
    - Data is visible and accessible to all steps
    - No way to prevent irrelevant codes to access the data
- Object-Oriented Decomposition
  - Software decomposed into objects
    - E.g., C++
    - Interact with each other to solve the problem
  - High reusability and flexibility

Object-Oriented Programming (OOP)

- Object
  - Basic unit that does the computation
  - Contain data and procedural functions
- Object-Oriented Programming
  - Objects are fundamental building blocks
  - Each object is an instance of some class
  - Classes have inheritance relationships

C++ v.s. Java

- Object-Oriented Language (C++)
  - It supports objects
  - It requires objects to belong to a class
  - It supports inheritance
- Object-Based Language (Java)
  - It supports objects
  - It requires objects to belong to a class

History of C++

- Creator of C++ : Bjarne Stroustrup
- C++ is an enhanced version of C
- Standardization:

Year	C++ Standard	Informal name
1998	ISO/IEC 14882:1998	C++98
2003	ISO/IEC 14882:2003	C++03
2007	ISO/IEC TR 19768:2007	C++TR1
2011	ISO/IEC 14882:2011	C++11

Wikipedia C++11 http://www.stroustrup.com/C++11FAQ.html

Programming Language Rank

Feb 2018	Feb 2017	Programming Language	Ratings	Change
1	1	Java	14.988%	-1.69%
2	2	C	11.857%	+3.41%
3	3	C++	5.726%	+0.30%
4	5	Python	5.168%	+1.12%
5	4	C#	4.453%	-0.45%
6	8	Visual Basic .NET	4.072%	+1.25%
7	6	PHP	3.420%	+0.35%

https://www.tiobe.com/tiobe-index//

Data Types

Abstraction and Encapsulation

- Data Abstraction
  - The separation between
    - Specification of a data object
    - Implementation
- Data Encapsulation (Information Hiding)
  - Conceals the implementation details from the outside world
- Benefit to large software system design

Data Type

- A collection of objects and a set of operations
- Fundamental data type in C++:
    - Basic: char, int, float, double, and many mores
    - Modifiers: short, long, signed, unsigned
- Example: int data type
    - Objects: 0, +1, -1, +2, -2, MAXINT, MININT
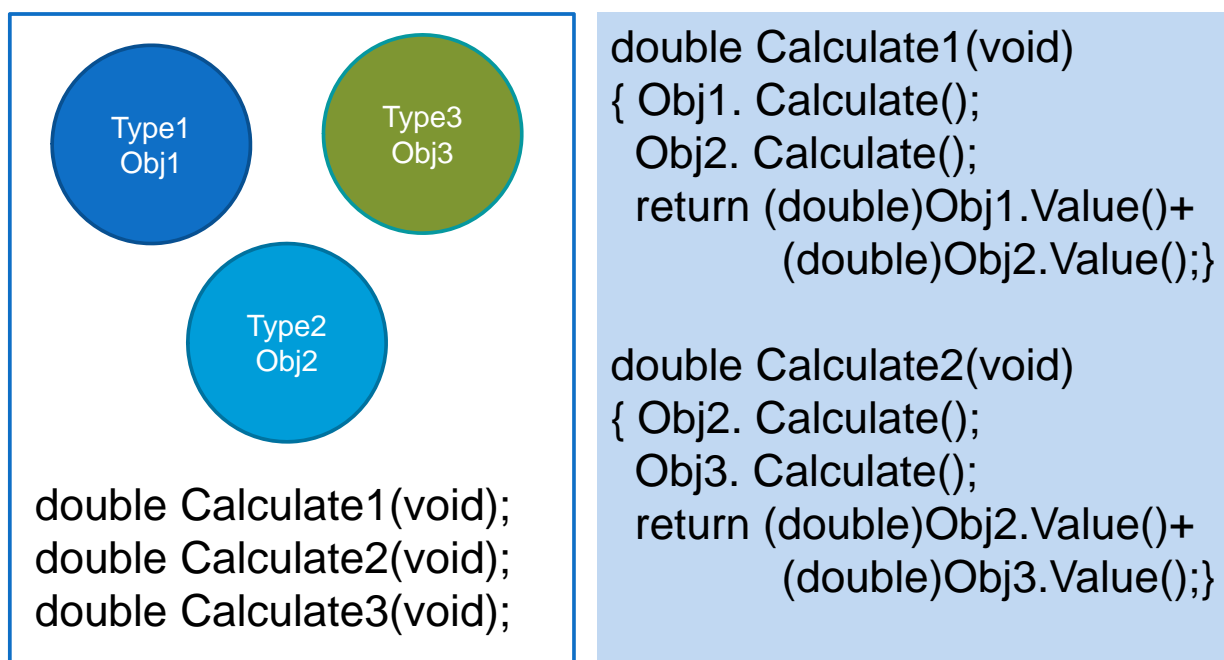   - Operations: +, -, *, /, ==, <=

Grouping Data

- Examples:
    - Arrays
        - Collection of elements of the same basic data type
    - structs (C) and classes (C++)
        - Collection of elements

Example

Data Type1	Data Type2	Data Type3

Objects:
int data;
 	
Operations:
int Value(void)
{ return data; }
 	
void Calculate(void)
{data = 100; }
	Objects:
	float data;
	
	Operations:
	float Value(void)
	{ return data; }
	
	void Calculate(void)
	{data= exp(-10); }
		Objects:
		  Type1 data;
		  int data2;
		Operations:
		int Value(void)
		{ return
		data.Value()+data2; }
		
		void Calculate(void)
		{data. Calculate();
		  data2 = 128; }

A Software that uses ADTs

Type1
Obj1

Type3
Obj3

Type2
Obj2

double Calculate1(void);
double Calculate2(void);
double Calculate3(void);

```
double Calculate1(void)
{ Obj1. Calculate();
  Obj2. Calculate();
  return (double)Obj1.Value()+
         (double)Obj2.Value();}

double Calculate2(void)
{ Obj2. Calculate();
  Obj3. Calculate();
  return (double)Obj2.Value()+
         (double)Obj3.Value();}
```

Advantages of ADTs

- Simplification

- Testing and debugging

  - Each object can be tested and debugged separately

- Reusability

- Flexibility

  - Could freely modify the internal implementation without affecting the rest of codes

Basic C++

Program Organization

■Header file (*.h)
  ■ Store declarations
■Hello_World.h

```
#ifndef _HELLO_WORLD_H_
#define _HELLO_WORLD_H_

void Hello_World(void);

// insert other declarations here
// …

#endif
```

■Source file (*.cpp)
  ■ Store source codes
■Hello_World.cpp

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

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

Scope in C++

■Local scope
  ■ A name declared in a block
■Class scope
  ■ Declaration associated with a class definition
■Namespace scope
  ■ Declaration associated with a namespace
■File scope

Example

```cpp
// using
#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 << endl;
  cout << y << endl;
  cout << first::y << endl;
  cout << second::x << endl;
  return 0;
}
```

Data Declaration in C++

- Constant values
  - E.g., 5, 'a', 4.3
- Variables
  - E.g., double income;
- Constant variables
  - The content must be fixed at declaration
  - E.g., **const int** MAX=500;
- Enumeration types
  - Declare a series of constants
  - E.g., **enum** semester {SUMMER, FALL, SPRING};

Data Declaration in C++ (Contd.)

- **Pointers**
  - Hold memory address of objects
  - E.g.,
    ```
    int i = 25;
    int* np; np = &i;
    ```
- **Reference types (C++ only)**
  - Provide a alternate name for an object
  - E.g.,
    ```
    int i=5;
    int& j=i;
    i = 7;
    cout << j << endl;
    ```

Reference v.s. Pointer

- The sematic differences between reference and pointer:
  - Pointer CAN be NULL but reference CANNOT be NULL
    - reference must bind a variable at initialization

    ```
    int * ptr = NULL;
    int & ref = NULL;
    ```

  - Pointer CAN be changed to point different target in the program but reference variable CANNOT be changed.

    ```
    int  x= 10, y=20;
    ptr = &x ;
    ptr = &y ;
    int & ref = x;
    &ref = y;
    ```

Comment

■One line comment:

// To increase the readability

■Multiple Line comment:

/*

Usually comment out some functions/procedures

*/

Functions in C++

■A function consists of
- A function name
- A list of arguments (input)
- A return type (output) or void
- The body

■Example

```
int Max (int a, int b)
{
        if (a>b) return a;
        else return b;
}
```

Parameter Passing in C++

■Call by **value**

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

■Value is copied into local storage
■Will not modify the original copies

Parameter Passing in C++

■Call by **pointer**

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

■Will modify the original objects

Quiz

■ What if we want to pass a large object but do not want it to be modified in the function?

Function Overloading in C++

■ In C++, we can have following functions:

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

■ It is impossible to defined two functions with the same name in C

Dynamic Memory Allocation in C++

- Dynamic Memory Allocation in C
  - malloc, delete, realloc, memset, memcopy
  - Memory leak and memory fragmentation problems

- New dynamic memory allocation mechanism
  - Using keywords "new" and "delete"
  - Make sure you use 'delete' for pointer generated by 'new'

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[]/

Exceptions Handle

- Handle runtime errors or special conditions
- Provide more clear programing logic

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

int main () {
  try {
    throw 20;
  }
  catch (int e) {
    cout << "An int-type exception
occurred.
    Exception Nr. " << e << endl;
  }
  return 0;
}
```

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

int main () {
  try {
    throw "error occurs";
  }
  catch (char* e) {
    cout << "An char-type exception
occurred.
    Exception Nr. " << e << endl;
  }
  return 0;
}
```

Exceptions Handle

- **try-catch** block
  - Each try block is followed by zero or more **catch** blocks.
  - Each **catch** block is visited sequentially until the matched block
  - Each **catch** block has a parameter whose type determine the type of exception that may 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 their type

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

Exceptions Handle – throw

```
int main(void)
{
    int a =0;

    func1();

    int b = a +2;
    int c  = 3;


    return 0;
}
```

```
void func1(){

  func2();

  int  a = 2;
  //....
  return ;
}
```

```
void func2(){

  throw "error!";
  std::cout<<"hi";

return ;
}
```

C++ Class

C++ Class

■Class can support data abstraction and encapsulation

```
// In the header file Rectangle.h
#ifndef RECTANGLE_H
#define RECTANGLE_H
class Rectangle {
public:    // the following members are public
           // the next four members are member functions
           Rectangle ( );        // constructor
           ~Rectangle( );        // destructor
           int GetHeight ( );    // return the height of the rectangle
           int GetWidth ( );     // return the width of the rectangle
private:   // the following members are private
           // the following members are data member
           int xLow, yLow, height, width;
           // (xLow, yLow) are the coordinates of the bottom left corner of rec.

};
#endif
```

Data Abstraction

■Specification is placed in header file (e.g., Rectangle.h)

■Implementation is placed in source file (e.g., Rectangle.cpp)

```
// In the source file Rectangle.cpp
#include "Rectangle.h"

/* The prefix "Rectangle::" identifies GetHeight() and GetWidth() are member function of
class Rectangle. It is required because the member functions are implemented outside
the class definition*/

int Rectangle::GetHeight() {return height;}
int Rectangle::GetWidth() {return width;}
```

Class Usage

```cpp
// In a source file main.cpp
#include <iostream>
#include "Rectangle.h"

main() {
        Rectangle r, s;        // r and s are objects of class "Rectangle"
        Rectangle *t = &s;     // t is a pointer to class object s
        .
        .
        // use "." operator to access members of class objects.
        // use "->" operator to access members of class objects through pointers.
        If ( r.GetHigh ( ) * r.GetWidth ( ) > t→GetHeight ( ) * t→GetWidth ( ) )
           cout << "r";
        else cout << "s";
        cout << "has the greater area " << endl;
}
```

Data Encapsulation

■C++

```cpp
class Foo{
 private:
         int  x;
 public:
         int  y;
};

int main(void){
  Foo  obj1 ;
  obj1.x = 11; // compile ERROR
  obj1.y = 22; // access y
}
```

■C

```c
struct Foo{

   int  x;
   int  y;

};

int main(void){
   struct Foo  obj1 ;
   obj1.x = 11; // access x
   obj1.y = 22; // access y
}
```

Constructors and Destructors

```cpp
// In the source file Rectangle.cpp
#include "Rectangle.h"

// constructor
Rectangle::Rectangle (void)
{
    xLow = 0; yLow = 0;
    height = 1; width = 1;
}

// destructor
Rectangle::~Rectangle (void)
{
    xLow = yLow = height = width = 0;
}

int Rectangle::GetHeight() {return height;}
int Rectangle::GetWidth() {return width;}
```

Constructors

- A member function to initialize the data members
- Constructor is invoked when an object is created
- Must has the same name as class
- No return type or return value
- A class can have more than one constructors

Type of Constructors

- Default constructor
  - A constructor with no arguments

    Rectangle **( );**   // default constructor

- Augmented constructor
  - A constructor with arguments

    Rectangle **(int, int, int, int);**   // augmented constructor

- Copy constructor
  - Must be specified if the STL containers are used to store your class object.

    Rectangle **(const *Rectangle*&);** // copy constructor

Destructor

- A member function to delete data members when the object disappears
- Destructor is **automatically** invoked when a class object is out of scope or is deleted
- Must has the same name as class with prefix "~".
- No return type or return value
- Take no arguments
- Only one destructor in a class

Default Methods

- The compiler will generate 4 default methods, if not specified
  - Default constructor

    Rectangle **( );**   // default constructor

  - Copy constructor

    Rectangle **(const *Rectangle*&);**   // copy constructor

  - Destructor

    ~Rectangle **();**   // destructor

  - Assignment operator

    Rectangle& operator=**(const *Rectangle*&);**   // operator "="

---

struct vs. class

- C

```
struct  MyData{

   int  id;

};


// instance struct

struct MyData   data1;
```

- C++

```
class MyData{

public:

   int id;

};


// instance object

MyData   data1;
```

"struct" in C++

■C++ - "struct"

```
struct Student{

  int   age;

public :

  int   id;

  char   name[100];

};
```

■C++ - "class"
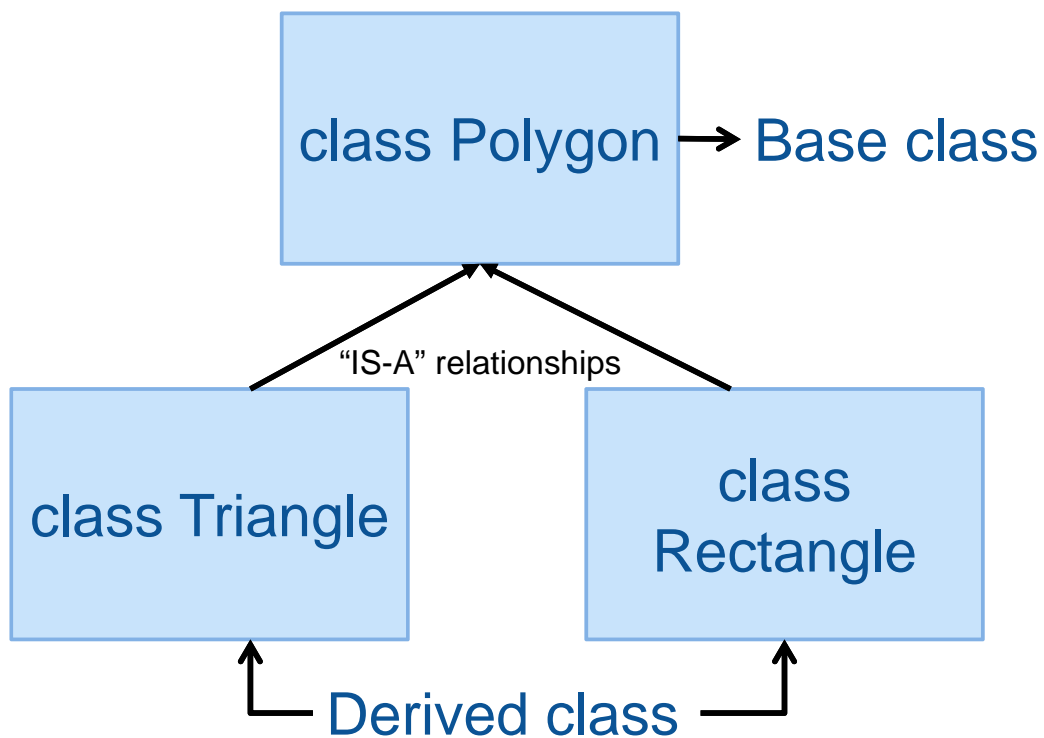
```
class Student{

  int   age;

public:

   int   id;

  char   name[100];

};
```

Inheritance

■Relate one class object to another

■Define a "IS-A" relationships between objects

- *Type B* **IS-A** data type of *Type A* if B is a **specialized** version of A and A is more **general** than B

■Members (data and functions) in Type A are implicitly copied to Type B.

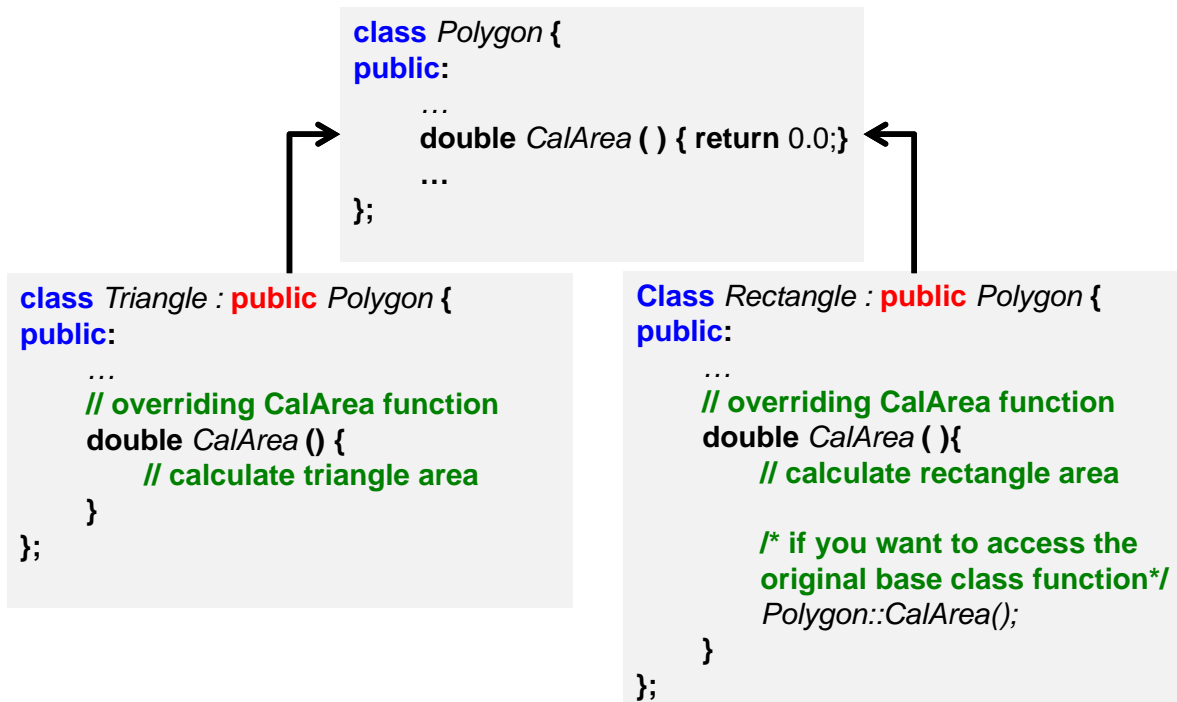■Reusability of codes

Class Diagram of inheritance

Specialization

- Put non-common members in private block of base class
  - Derived class can not access these members
- Re-declare the members (data and functions) in the derived class (overriding)

Overriding

```
class Polygon {
public:
    …
    double CalArea ( ) { return 0.0;}
    …
};
```

```
class Triangle : public Polygon {
public:
    …
    // overriding CalArea function
    double CalArea () {
        // calculate triangle area
    }
};
```

```
Class Rectangle : public Polygon {
public:
    …
    // overriding CalArea function
    double CalArea ( ){
        // calculate rectangle area

        /* if you want to access the
        original base class function*/
        Polygon::CalArea();
    }
};
```

Polymorphism

- Manipulate different objects through the common interface

```
class Foo
{
    public:
        virtual char* getName()
    { return "foo"; }
};
```

```
class Bar : public Foo
{
    public:
        virtual  char* getName()
    { return "Bar"; }
};
```

```
class Car : public Foo
{
    public:
        virtual char* getName()
    { return "Car"; }
};
```

```
int main(){
    Foo* myFoo = new Foo;
    Foo* myBar = new Bar;
    Foo* myCar = new Car;

                                    processObj(Foo* _obj)
                                    {… _obj->getName()…}
    processObj(myFoo);
    processObj(myBar);
    processObj(myCar);
}
```

Polymorphism

■ Function Overloading
- Data type is determined in compiler time

```
int main(){
      Foo myFoo;
      Bar myBar;
      Car myCar;

      processObj (myFoo);
      processObj (myBar);
      processObj (myCar);
}
```

■ Dynamic Binding
- Data type is determined in run time

```
int main(){
      Foo* myFoo = new Foo;
      Foo* myBar = new Bar;
      Foo* myCar = new Car;

      processObj(myFoo);
      processObj(myBar);
      processObj(myCar);
}
```
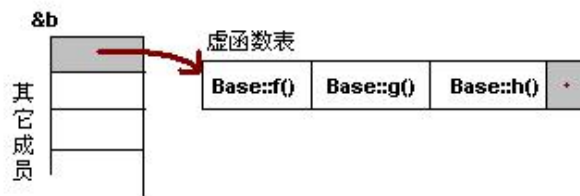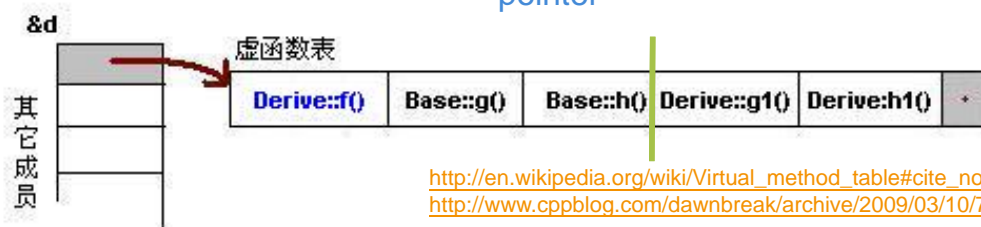
How dose Dynamic Binding Work ?

■ Virtual Method Table

Base * ptr = new Base;

Base * ptr = new Derived;

Virtual method table of Derived pointer will be "sliced" when it be assigned to a base pointer

http://en.wikipedia.org/wiki/Virtual_method_table#cite_note-4
http://www.cppblog.com/dawnbreak/archive/2009/03/10/76084.html

Dynamic Binding: Pros and Con

- Pros:
    - Ideal data abstraction.
    - Powerful mechanism of OOP (Design Pattern)
    - Widely used in large-scale software design
- Con:
    - Decreasing performance
        - Additional memory to store virtual function table
        - Additional runtime cost to access virtual function table

References

- *C++ Primer 5th*
    - http://books.google.com.tw/books?hl=zh-TW&id=J1HMLyxqJfgC&q=operator+overaling#v=onepage&q=chapter%2014&f=false
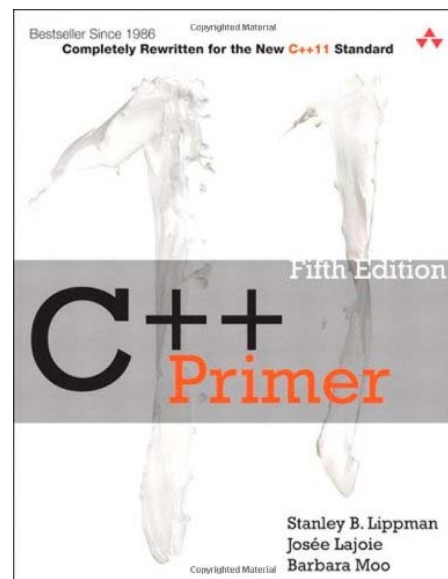- MIT's Introduction to C++
    - http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-096-introduction-to-c-january-iap-2011/lecture-notes/
- MSDN C++ Reference:
    - http://msdn.microsoft.com/en-us/library/3bstk3k5(v=vs.100).aspx
- NTU OCW:
    - http://ocw.aca.ntu.edu.tw/ntu-ocw/index.php/ocw/cou/101S112