**Front End Group**

**System Level Design Team**

**Design Methodology Committee**

# C/C++ Programming

Renesas Design Vietnam Co., Ltd.
Loi Huynh, Technology Development Dept.

May 17, 2010Date Rev. 0.00

Confidential

# Outline

- Summary
- Introduction to C++ language
- Introduction to Object-Oriented
- C/C++ programming
- Appendix

# Outline

- **Summary**
- Introduction to C++ language
- Introduction to Object-Oriented
- C++ programming
- Appendix

# Summary

- System Level Design languages expresses functional specification and design constraints
  - Algorithms, precisions, concurrency, clock, interruptions, etc.
  - Variety of abstraction levels.
- Major system-level design languages: ***SystemC*** (OSCI, IEEE std. 1666-2005), SpecC (Univ. of California, Irvine), BDL (NEC), Bach-C (Sharp), ***System Verilog*** (Accellera, IEEE std. 1800-2005), UML (OMG).
- In RVC, ***SystemC*** which is derived from ***C/C++*** programming language is used to develop timed model.
- This document will introduce about ***C/C++*** with illustration coding.

# Outline

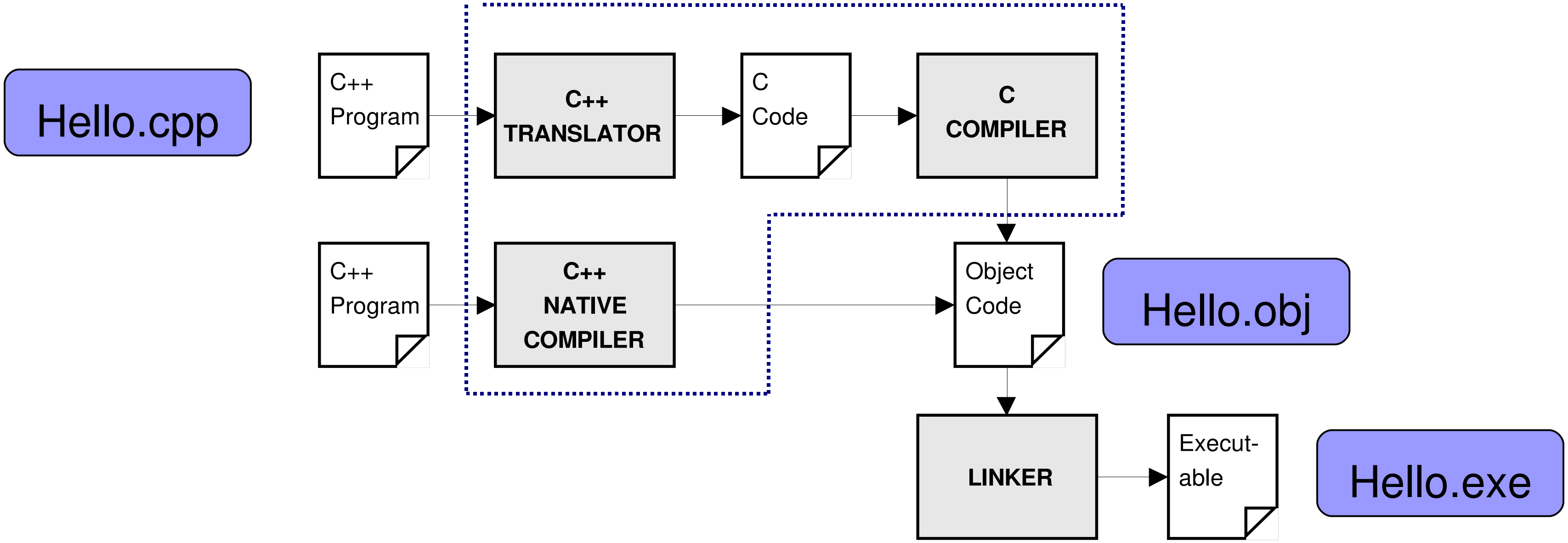
- Summary
- **Introduction to C++ language**
- Introduction to Object-Oriented programming
- C++ programming
- Appendix

# Introduction to C++ language (1/2)

- C++ was invented by Bjarne Stroustrup in 1979, at Bell Laboratories in Murray Hill, New Jersey.
- The language began as enhancements to C, first adding ***classes***, then ***virtual functions***, ***operator overloading***, ***multiple inheritance***, ***templates***, and ***exception handling*** among other features.
- All OOP language, including C++, have three traits in common: ***encapsulation***, ***polymorphism***, and ***inheritance***.
- GNU C++ compiler, Microsoft's Visual C++, and Borland's bcc are the major C++ compilers. Everyone can use them free of charge. SH Compiler also support C++. C++ compilers can compile C codes more strictly than C compilers.

# Introduction to C++ language (2/2)

- C++ compiler process
  - ***Source code*** is the text of a program that a user can be read.
  - ***Object code*** is translation of the source code of a program into machine code, which the computer can read and execute directly.
  - ***Executable program*** is output of the linker which is used to link separately compiled modules into one program.

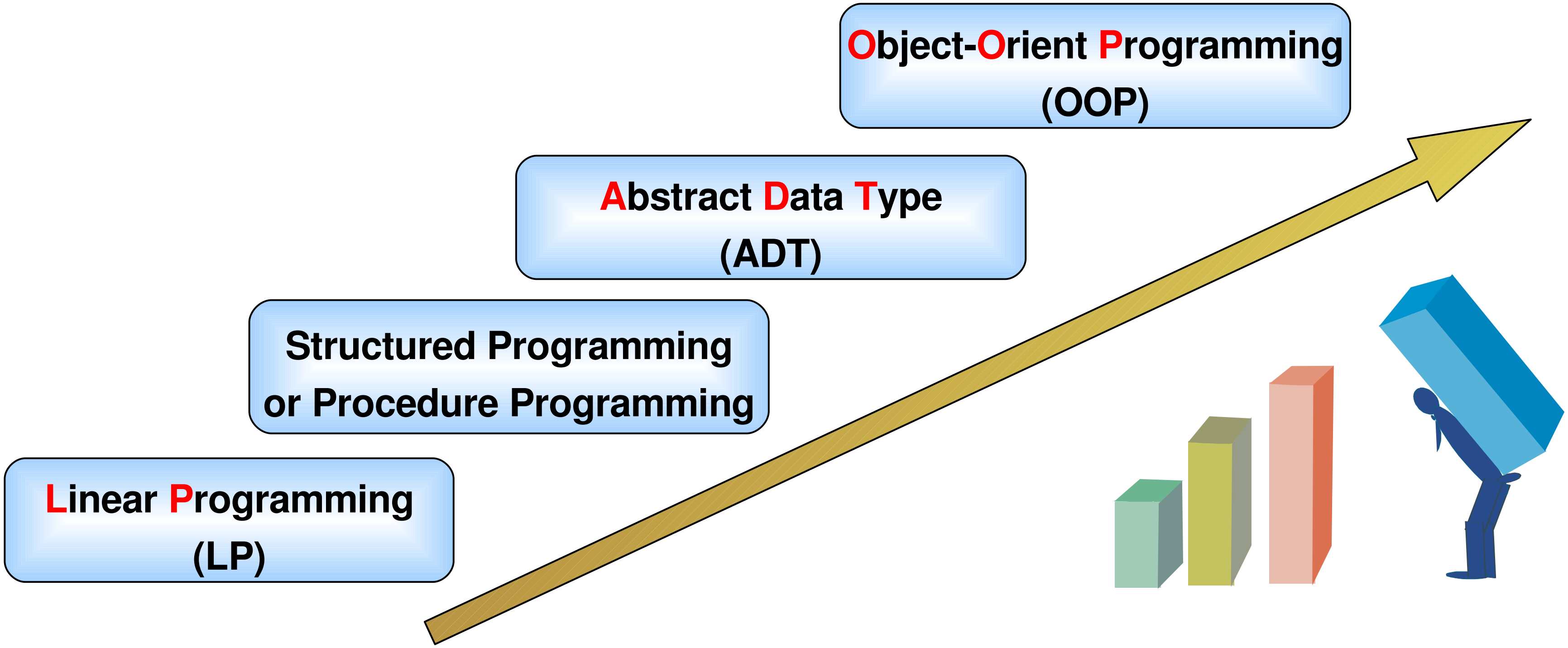


# Outline

- Summary
- Introduction to C++ programming
- Introduction to Object-Oriented programming
    - Object - Instance
    - Class
    - Attribute & Method
    - Message
    - Encapsulation
    - Inheritance
    - Polymorphism
- C++ programming
- Appendix

# What is Object-Oriented Programming

**Object-Oriented Programming (OOP) is a software design and development method basing on class and object architecture**



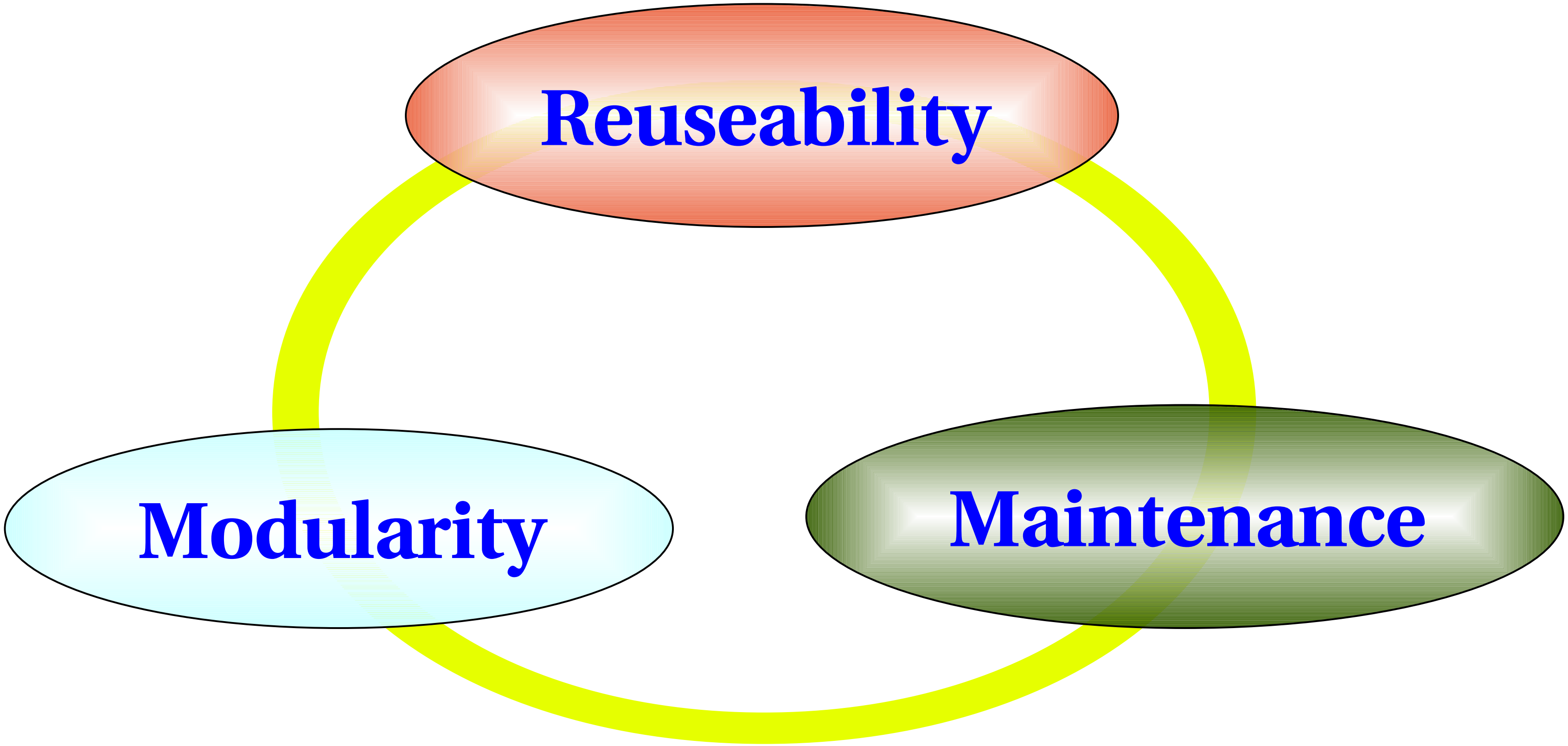
Object-Orient Programming (OOP)

Abstract Data Type (ADT)

Structured Programming or Procedure Programming

Linear Programming (LP)

# What is advantage of OOP?



Reuseability

Modularity

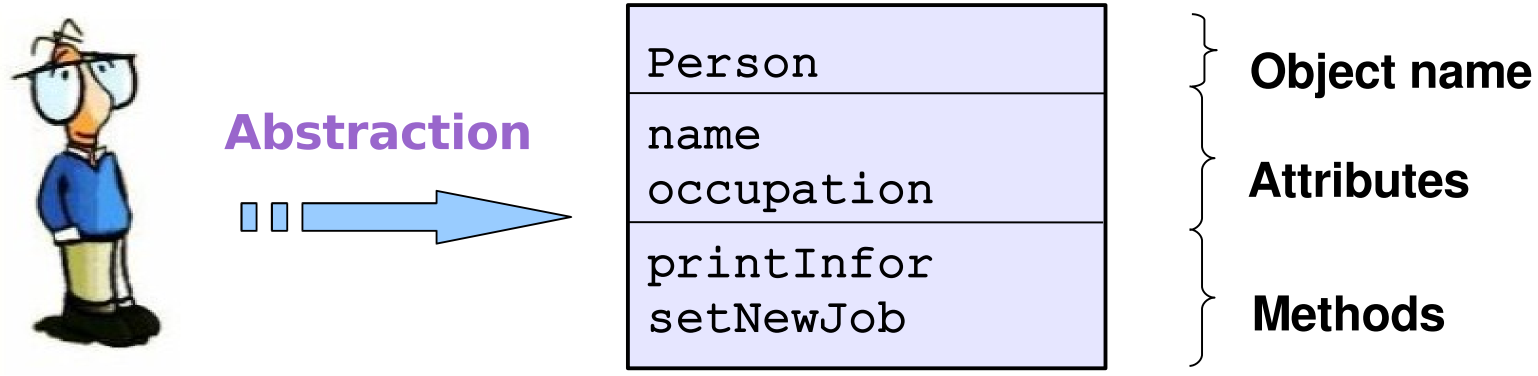
Maintenance

# How to study OOP?

- To use OOP efficiently, the Programmers need to understand OOP concepts.
- Some concepts will be introduced:
  - Object - Instance
  - Class
  - Method & Attribute
  - Message
  - Encapsulation
  - Inheritance
  - Polymorphism

# Object - Instance

**Object is an software entity which includes attributes and methods**



Abstraction

```
Person
name
occupation
printInfor
setNewJob
```

Object name

Attributes

Methods

Real Object

Software Object

**A particular object is called an instance.**

```
Person
name = Peter
occupation = Engineer
printInfor
setNewJob
```

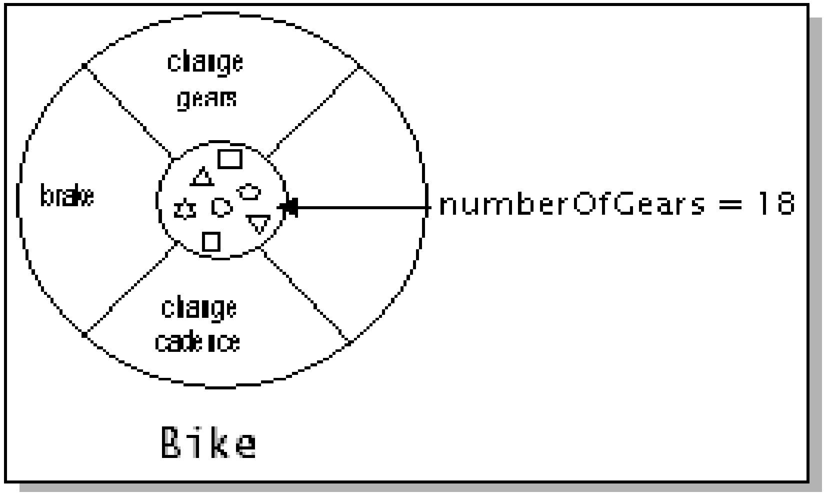
```
Person
name = Micheal
occupation = Singer
printInfor
setNewJob
```

# Class

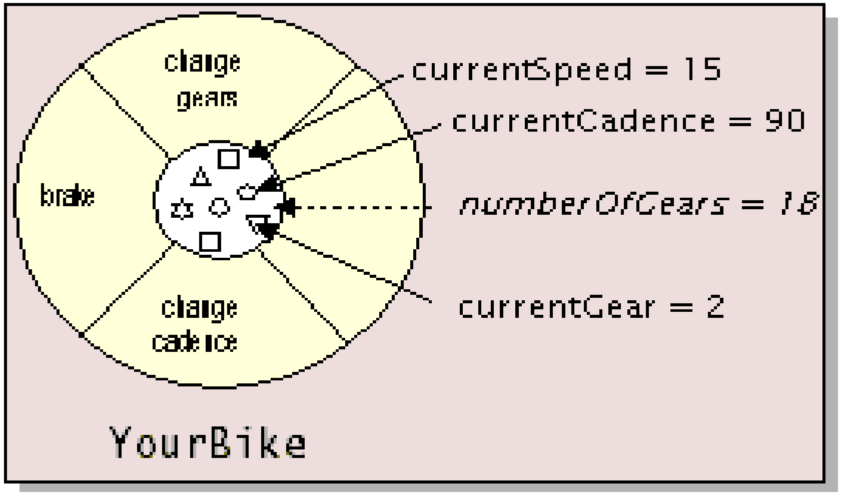
Class which is a blueprint or prototype defines all common attributes and methods for objects which are have the same type.

Class is essentially an Abstraction Data Type.

A object is a specific instance of a class.



Class

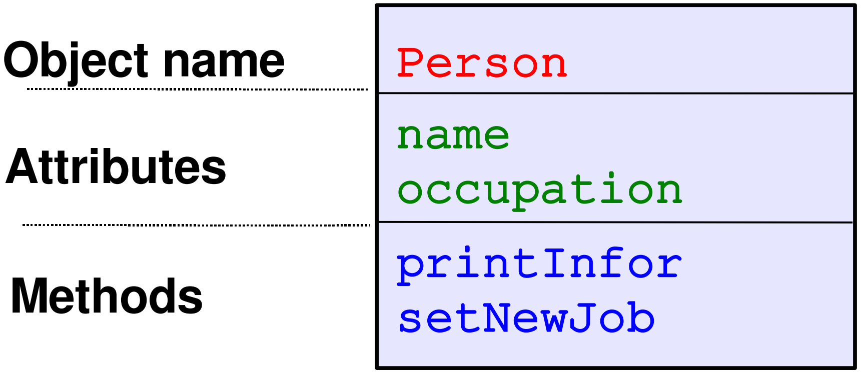


Instance of a Class

# Attribute – Method

**Attribute is data used to describe properties (variables) of a object**

**Method are behaviors which object can do. Each method have ability to effect attribute of object.**


Object name

Attributes

Methods

Person

name
occupation

printInfor
setNewJob

`printInfor` method print information by reading `name` attribute and `occupation` attribute.

`setNewJob` method is used to set value to `occupation` attribute

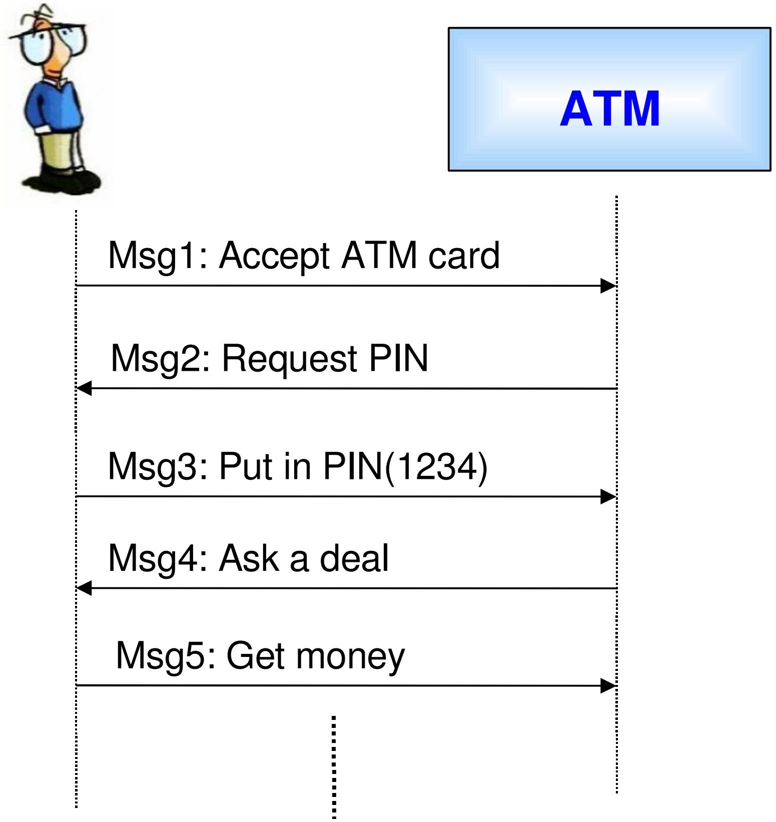
**In C++ programming, Attribute also is called Member data and Method is called Member function**

# Message

Message is a request an object to invoke a method.

A message consists of:

- Object receiving message.
- Method used to process message.
- Parameter contenting necessary information for method.

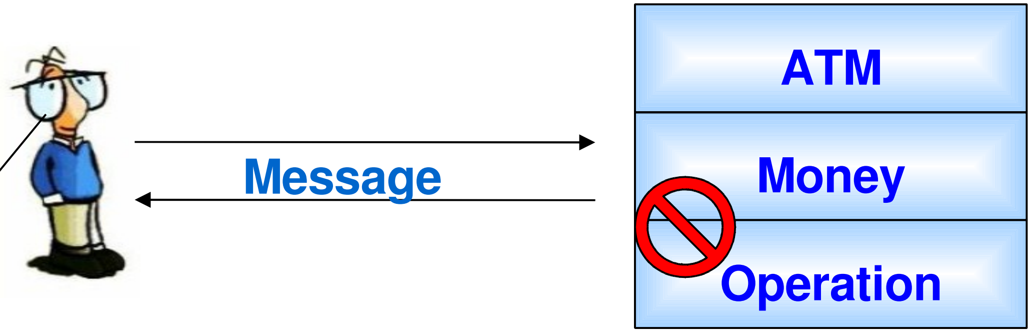


Msg stands for Message

ATM stands for Automated Teller Machine

# Encapsulation

**Encapsulation conceals the functional details of a class from objects that send messages to it.**



Message

ATM

Money

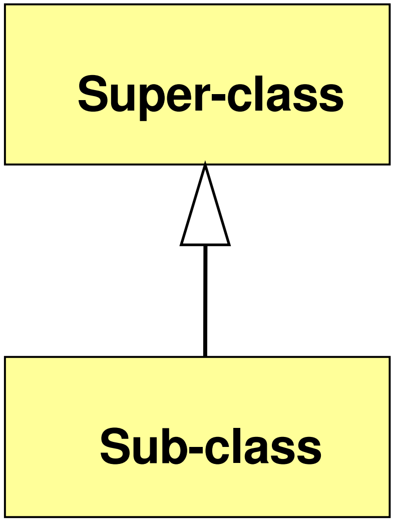
Operation

- How much money are there in this machine?
- How does the machine operate?
- Can I get money without PIN?

It is a top secret. So I can not reveal something more.

# Inheritance

Inheritance means that a class can inherit or re-use attributes and methods that are defined in other class.



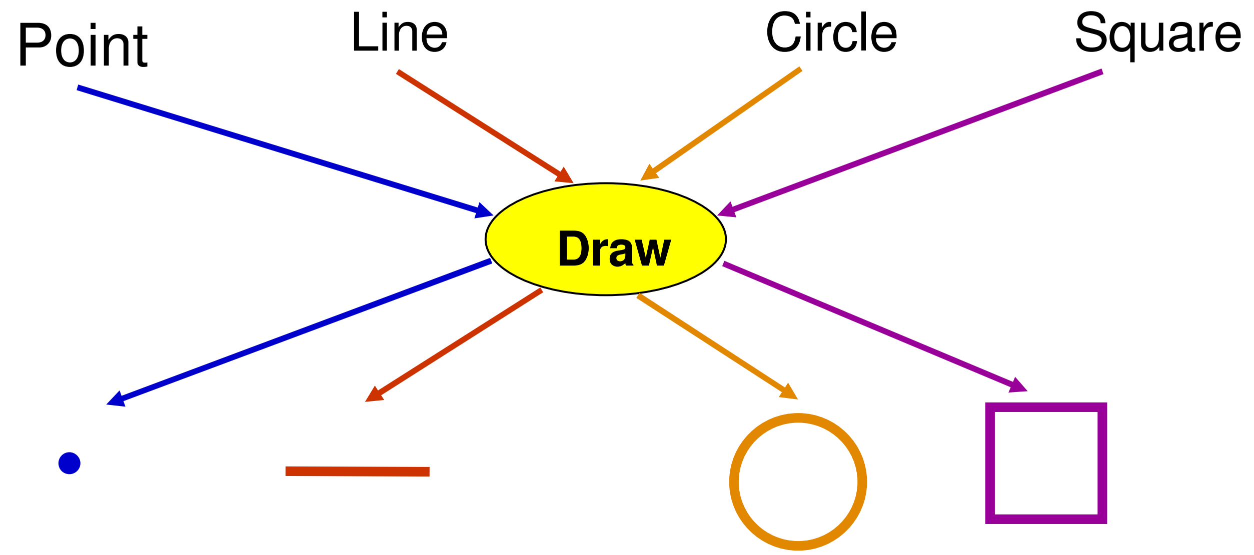
Super-class is a class having some attributes or methods that one or more classes inherit.

Sub-class is a class having some inherited attributes or methods from Super Class and adding its own attributes or methods.

✍ In C++ programming, Super-class is called Base Class and Sub-class is called Derived Class

# Polymorphism

**Polymorphism is the ability of objects belonging to different data types to respond to method calls of methods of the same name, each one according to an appropriate type-specific behavior.**



# Outline

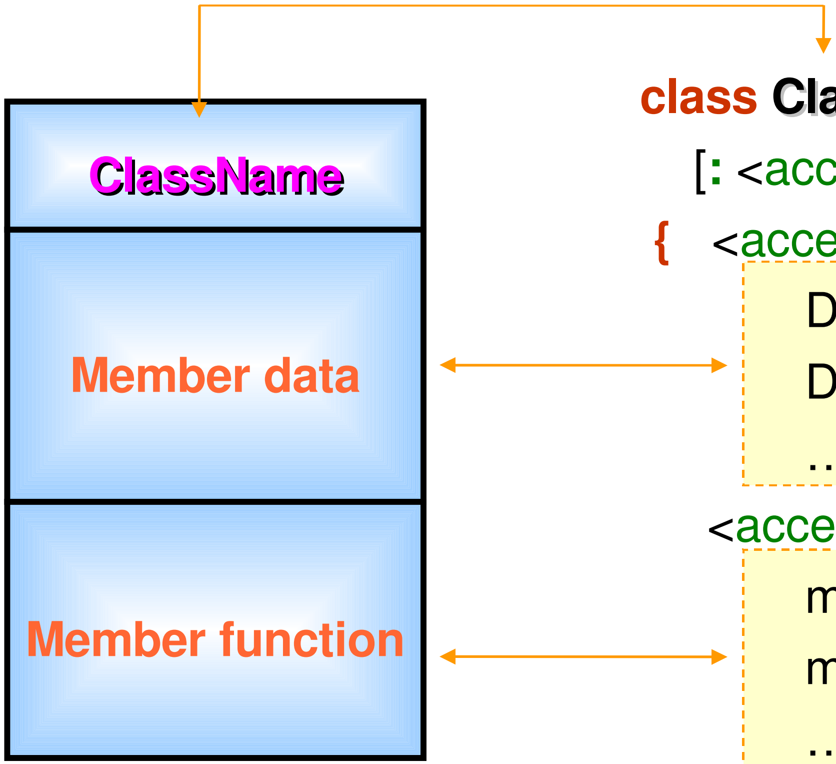
- Summary
- Introduction to C++ programming
- Introduction to Object-Oriented programming.
- C++ programming
  - Class and object
  - Overloading
  - Inheritance
  - Programming method
- Appendix

# Outline

- Summary
- Introduction to C++ programming
- Introduction to Object-Oriented programming.
- C++ programming
  - Class and object
    - Class
    - Inline member function
    - Constructor
    - Destructor
    - Friend
    - This pointer
    - Scope resolution operator ::
    - Member initialization list
    - Special member data
    - Nested class

# Class (1/4)

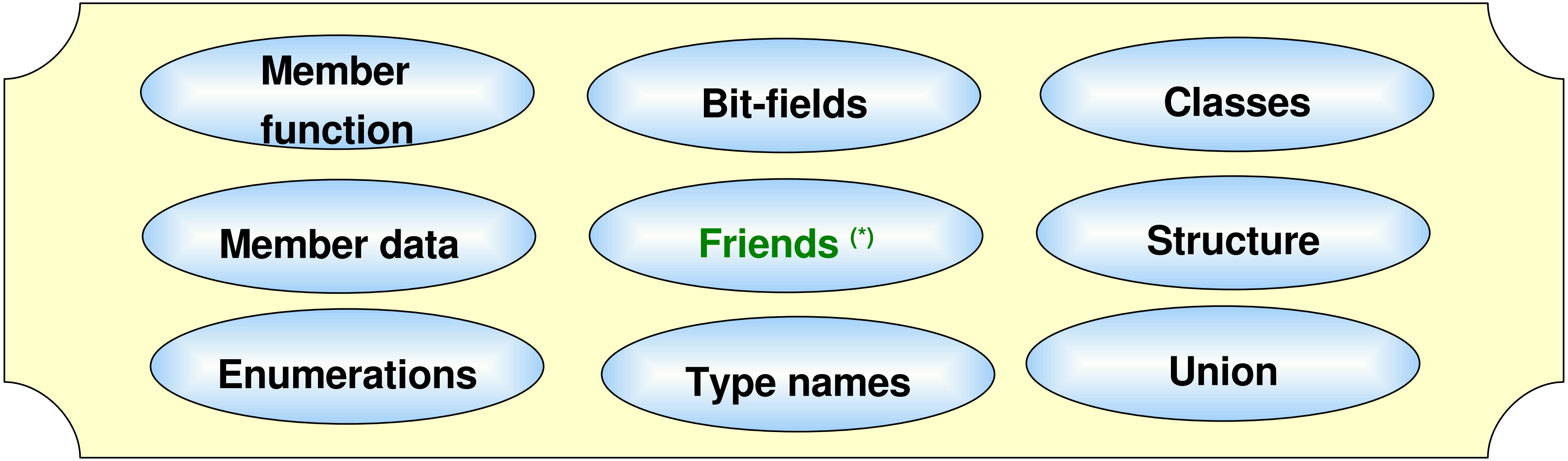
**Class** is an **Abstraction Data Type**.



```
class ClassName
   [: <access control> BaseClass ]
{   <access control> :
        DataType1  memberdata1;
        DataType2  memberdata2;
        .................
    <access control> :
        memberFunction1();
        memberFunction2();
        ..............
};
```

# Class (2/4)

**Class can have these kinds of members:**



Member function

Bit-fields

Classes

Member data

Friends [*]

Structure

Enumerations

Type names

Union

[*] **Friends are included in the preceding list because they are contained in the class declaration. However, they are not true class members, because they are not in the scope of the class.**

# Class (3/4)

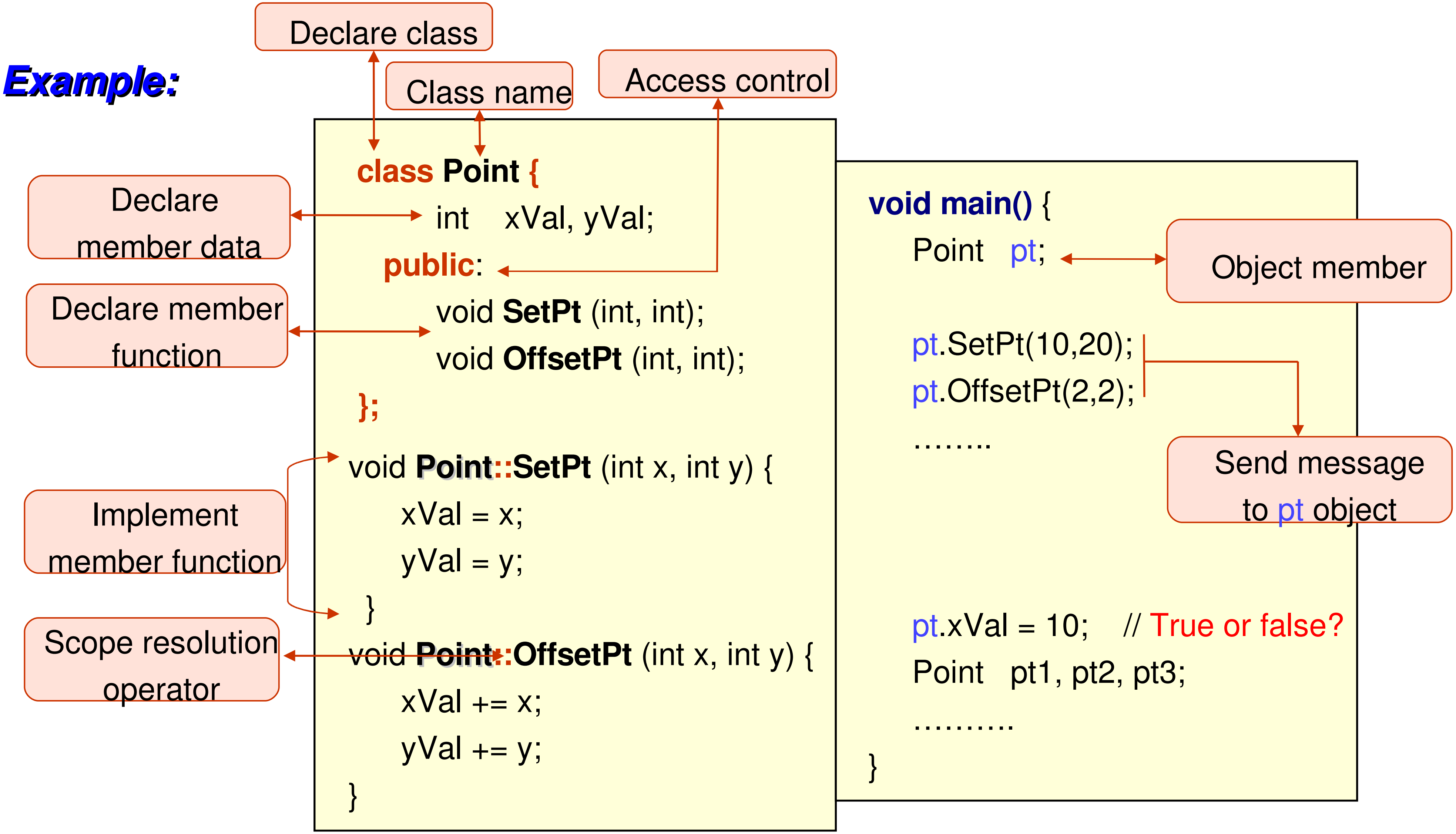
**Access control prevents you from using objects in ways they were not intended to be used.**

| Type of access | Meaning |
|---|---|
| **private** | Class members declared as **private** can be used only by member functions and friends (classes or functions) of the class. |
| **protected** | Class members declared as **protected** can be used by member functions and friends (classes or functions) of the class. Additionally, they can be used by classes derived from the class. |
| **public** | Class members declared as **public** can be used by any function. |

**Access control is equally applicable to all names: member functions, member data, nested classes, and enumerators.**

# Class (4/4)

***Example:***



```
class Point {
        int    xVal, yVal;
    public:
        void SetPt (int, int);
        void OffsetPt (int, int);
};

void Point::SetPt (int x, int y) {
        xVal = x;
        yVal = y;
  }
void Point::OffsetPt (int x, int y) {
        xVal += x;
        yVal += y;
}
```

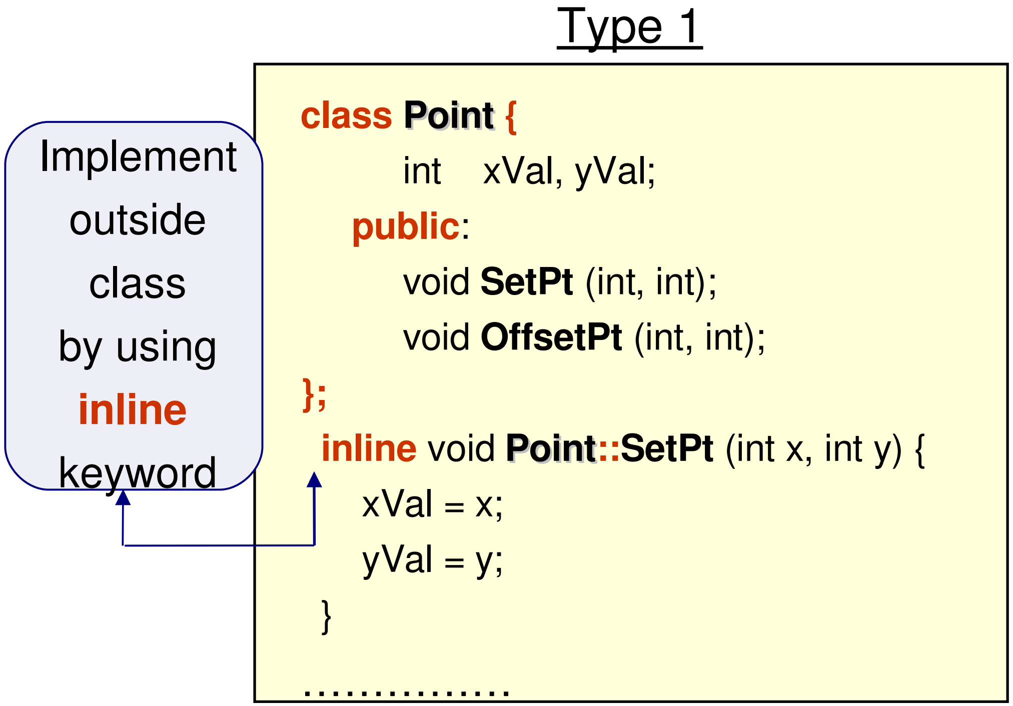
```
void main() {
    Point   pt;

    pt.SetPt(10,20);
    pt.OffsetPt(2,2);
    ……..

    pt.xVal = 10;    // True or false?
    Point   pt1, pt2, pt3;
    ……….
}
```

# Inline member function

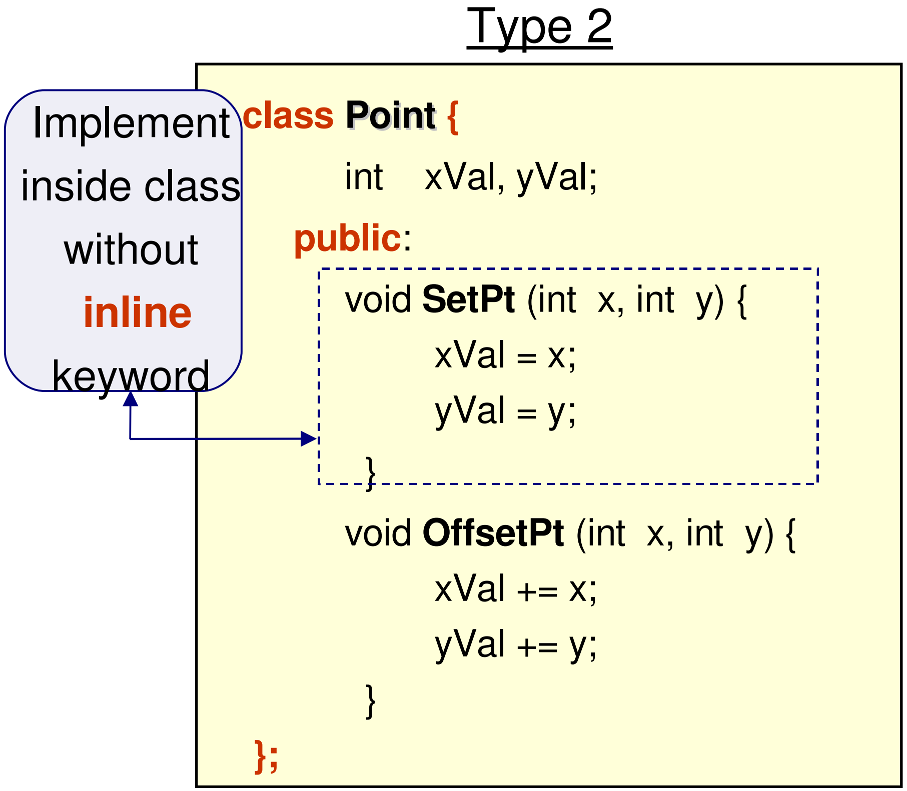
**The inline specifiers instruct the compiler to insert a copy of the function body into each place the function is called.**



Type 1

Implement outside class by using **inline** keyword

```
class Point {
        int   xVal, yVal;
    public:
        void SetPt (int, int);
        void OffsetPt (int, int);
};
 inline void Point::SetPt (int x, int y) {
      xVal = x;
      yVal = y;
 }
...............
```



Type 2

Implement inside class without **inline** keyword

```
class Point {
        int   xVal, yVal;
    public:
        void SetPt (int  x, int  y) {
              xVal = x;
              yVal = y;
         }
        void OffsetPt (int  x, int  y) {
              xVal += x;
              yVal += y;
         }
 };
```

**Type 1 have larger memory and fast execution.**

# Constructor (1/2)

**Constructor is a special function used to initialize member data. It is called automatically when object is created.**

**A member function with the same name as its class is a constructor function. Constructors cannot return values.**

```
class Point {
        int    xVal, yVal;
    public:
        Point (int  x, int  y) {
                xVal = x;     yVal = y;
        }
        void OffsetPt (int  x, int  y) {
                xVal += x;  yVal += y;
        }
 };
```

```
void main() {
    Point   pt1(10,20);
    pt1.OffsetPt(2,2);
    ........
  // What is wrong ?
    Point  pt2;
    Point  *pt3 = new Point();
    Point  pt4 = Point(5,5);
    Point  *pt5 = new Point(5,5);
    ..........
}
```

**Error**

We can classify constructor into 3 types:

- No constructor.
- Constructor without argument.
- Constructor with argument.

# Constructor (2/2)

**Constructors are called at the point an object is created. Objects are created as:**

- **Global** (file-scoped or externally linked) **objects**.
- **Local objects**, within a function or smaller enclosing block.
- **Dynamic objects**, using the new operator. The new operator allocates an object on the program heap or "free store."
- **Base class** sub-object of a class. Creating objects of derived class type causes the base class components to be created.

**If there is no constructor declared in class, compiler will create the default constructor without argument. An integer member data will be initialized 0 and a pointer member data  will be initialized as NULL pointer.**

# Destructor

"Destructor" functions are the inverse of constructor functions. It is called automatically when objects are destroyed (deallocated).

A class can define many constructor; however, a class has only one destructor.

```
class Set  {
   private:
        int    *elems;
        int    maxCard;
        int    card;
   public:
        Set(const int size) { …… }
      ~Set() {  delete[]   elems;  }
        ….
 };
```

What is difference between delete and delete[ ]?

In this case, using delete[] is illegal. It must be delete.

# Friend function (1/4)

```
class IntSet {
    public:
        //...
        void SetToReal(RealSet &set)
    private:
        int elems[maxCard];
        int card;
};
```

```
class RealSet {
    public:
        //...
    private:
        float elems[maxCard];
        int   card;
};
```

SetToReal function change
Real set to Integer set

```
void IntSet::SetToReal (RealSet &set) {
    card = set.card;
    for (register i = 0; i < card; ++i)
        set.elems[i] = (float) elems[i];
}
```

How to access
private member
of RealSet?

# Friend function (2/4)

A friend function is a function that is not a member of a class but has access to the class's private and protected members.

Friend functions are not considered class members.

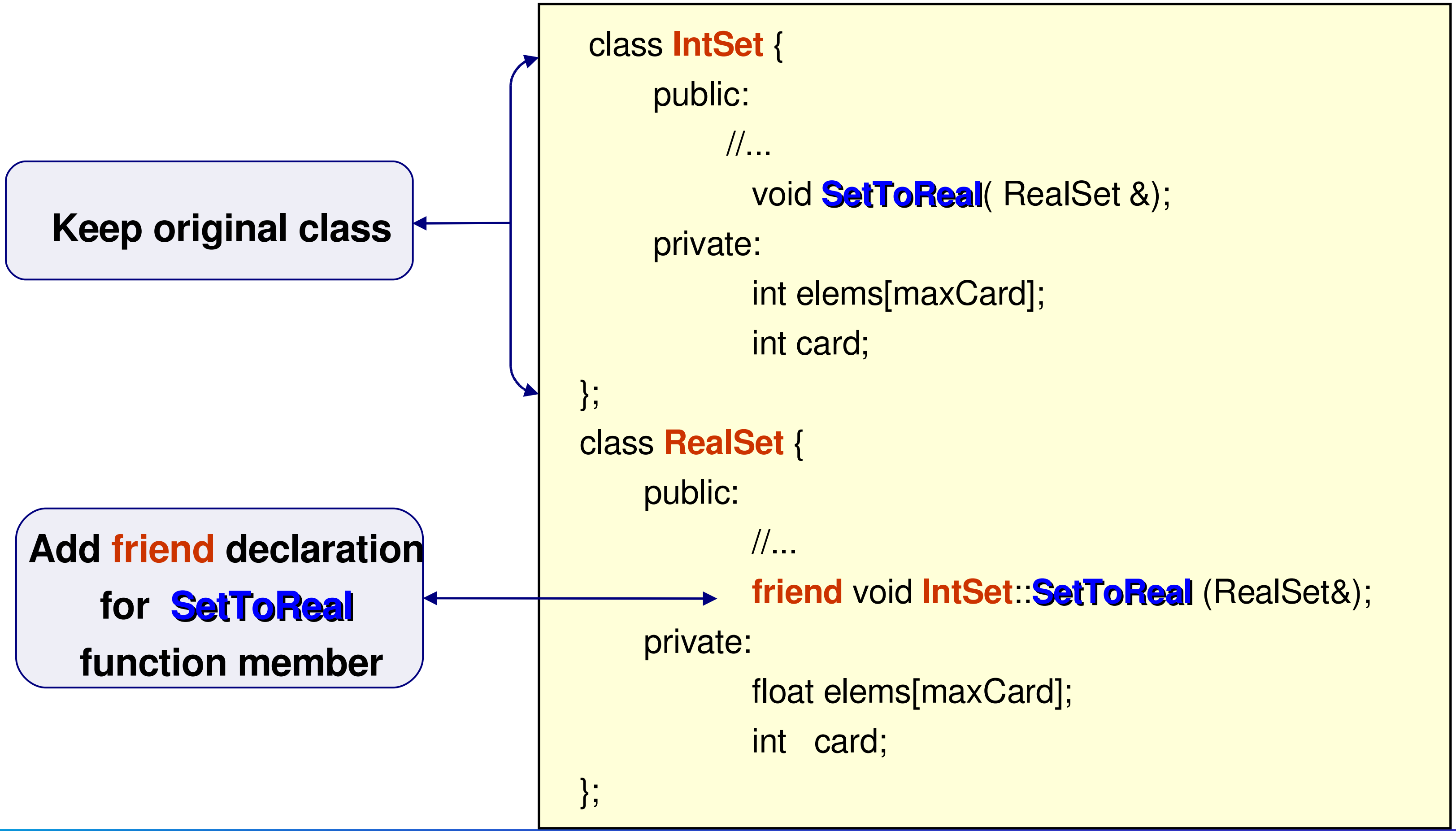
➔ Friends are not in the class's scope

➔ Friends are not called using the member-selection operators (. and –>)

The friend declaration can be placed anywhere in the class declaration. It is not affected by the access control keywords.

# Friend function (3/4)

**Type 1: Declare the SetToReal of IntSet class as a friend function of RealSet class**



Keep original class

Add friend declaration for SetToReal function member

```
 class IntSet {
        public:
                //...
                  void SetToReal( RealSet &);
        private:
                  int elems[maxCard];
                  int card;
};
class RealSet {
        public:
                  //...
                  friend void IntSet::SetToReal (RealSet&);
        private:
                  float elems[maxCard];
                  int   card;
};
```

# Friend function (4/4)

**Type 2:**

- **Make SetToReal as a global function.**
- **Declare SetToReal function as friend of IntSet and RealSet class.**

```
class IntSet {
       public:
               //...
               friend void SetToReal (IntSet &, RealSet&);
       private:
               int elems[maxCard];
               int card;
};
class RealSet {
      public:
               //...
               friend void SetToReal (IntSet &, RealSet&);
      private:
               float elems[maxCard];
               int   card;
};
```

The global function and the friend of both classes

```
void SetToReal (IntSet&    iSet,
                         RealSet& rSet )
{
   rSet.card = iSet.card;
   for (int i = 0; i < iSet.card; ++i)
     rSet.elems[i] =
            (float) iSet.elems[i];
}
```

# Friend class

**A friend class is a class all of whose member functions are friend functions of a class**

```
class A;
class B { // ……….
        friend class A;
};
```

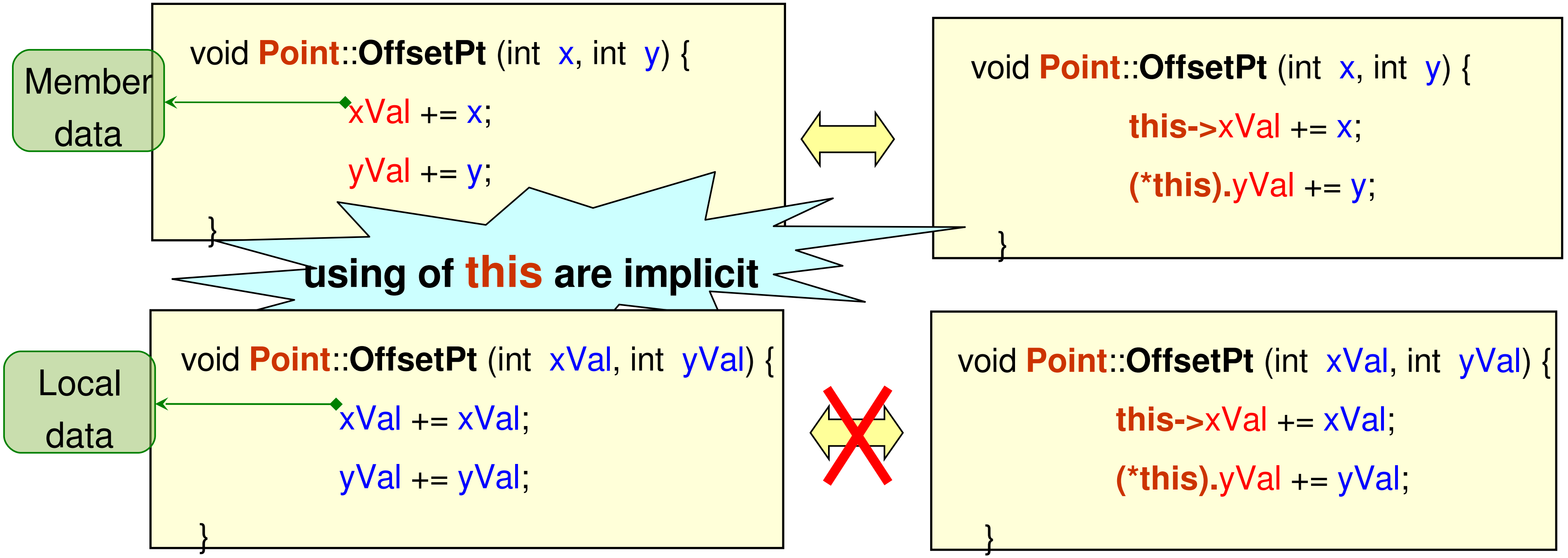
**Syntax**

```
class IntSet { ……….. }
class RealSet { // ……….
        friend class IntSet;
};
```

**Example**

# This pointer

**In C++, this (also called self or Me in other language) is a keyword that is used in instance methods to refer to the object on which they are working.**



Member data

```
void Point::OffsetPt (int  x, int  y) {
        xVal += x;
        yVal += y;
  }
```

```
void Point::OffsetPt (int  x, int  y) {
        this->xVal += x;
        (*this).yVal += y;
  }
```

**using of this are implicit**

Local data

```
void Point::OffsetPt (int  xVal, int  yVal) {
        xVal += xVal;
        yVal += yVal;
  }
```

```
void Point::OffsetPt (int  xVal, int  yVal) {
        this->xVal += xVal;
        (*this).yVal += yVal;
  }
```

**Static member functions do not have a this pointer.**

# Scope resolution operator ::

You can tell the compiler to use the global identifier rather than the local identifier by prefixing the identifier with ::, the scope resolution operator.

```
:: identifier

class-name :: identifier

namespace :: identifier
```

**Syntax**

*The identifier can be a variable or a function.*

It is necessary to use the :: in some cases:

- Call member function of base class.
- Access a identifier which is concealed by the local identifier.

# Member initialization list (1/2)

**The member initialization list is the preferred method to initialize the members of a class.**

```
class Point {
        int    xVal, yVal;
    public:
        Point (int  x, int  y) {
             xVal = x;
             yVal = y;
         }
        // ………………………
 };
```

```
Point::Point (int  x, int  y)

          : xVal(x), yVal(y)

   {  }
```

```
class Image {
        public:
                Image(const int w, const int h);
        private:
                int   width;
                int   height;
               //...
  };
Image::Image(const int w, const int h) {
                width = w;
                height = h;
                //....................
  }
```

```
Image::Image (const int w, const int h)
   : width(w), height(h)
  {   //..............   }
```

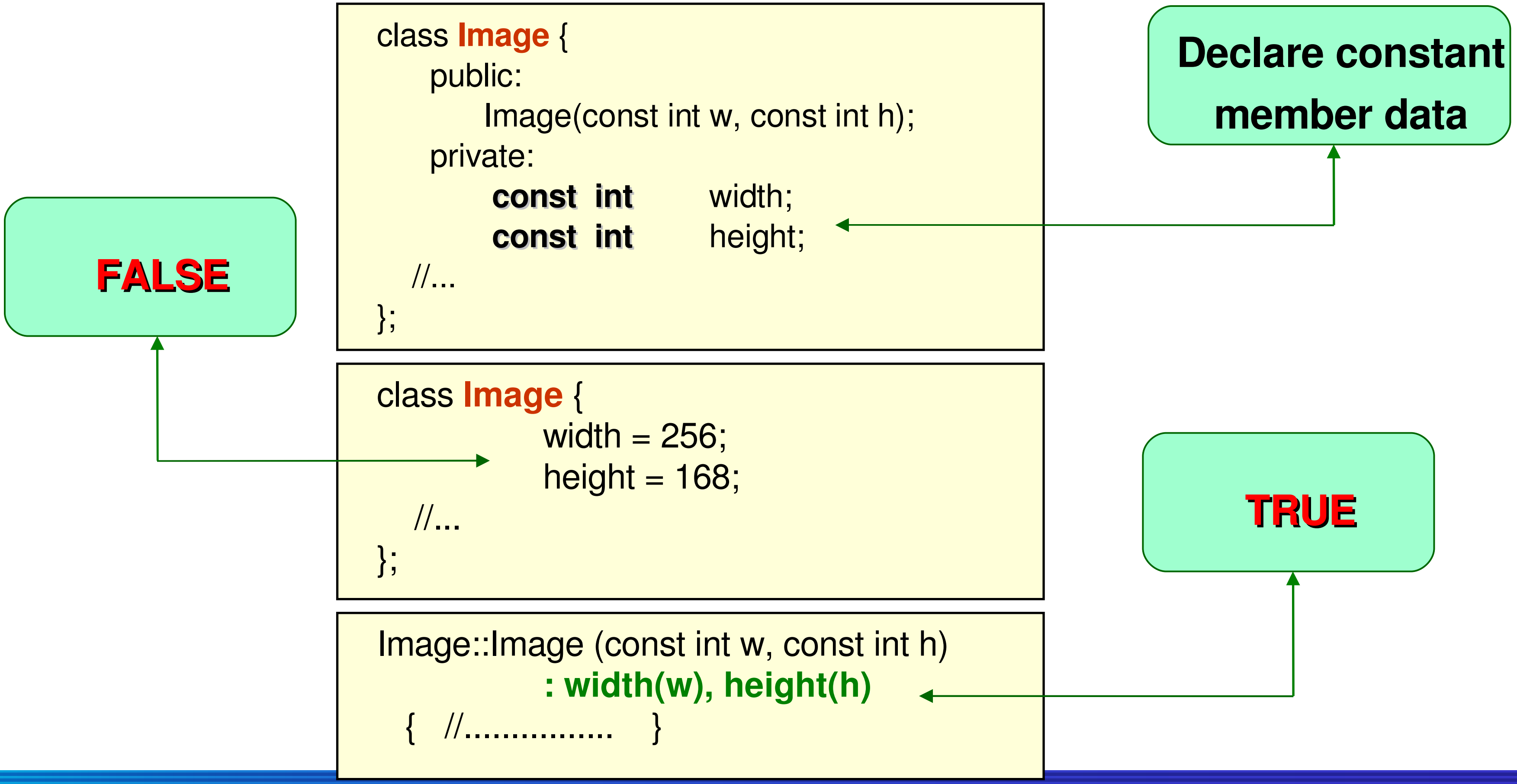
# Member initialization list (2/2)

**Member initialization list is used to initialize:**

- **Constant member data**
- **Reference member data**
- **Member object**
- **Base class**

# Constant member (1/2)

**The const keyword specifies that a member data's value is constant and tells the compiler to prevent the programmer from modifying it.**



```
class Image {
     public:
          Image(const int w, const int h);
     private:
          const  int       width;
          const  int       height;
   //...
};
```

Declare constant member data

```
class Image {
               width = 256;
               height = 168;
   //...
};
```

FALSE

```
Image::Image (const int w, const int h)
               : width(w), height(h)
   {   //...............    }
```

TRUE

# Constant member (2/2)

**Declaring a member function with the const keyword specifies that the function is a "read-only" function that does not modify the object for which it is called.**

**Declare constant member function**

```
class Set {
        public:
            Set(void){ card = 0; }
            Bool     Member(const int) const;
            void     AddElem(const int);
            //...
};
Bool Set::Member (const int elem) const
 {       //...
 }
```

**Implement constant member function**

**Declare constant object**

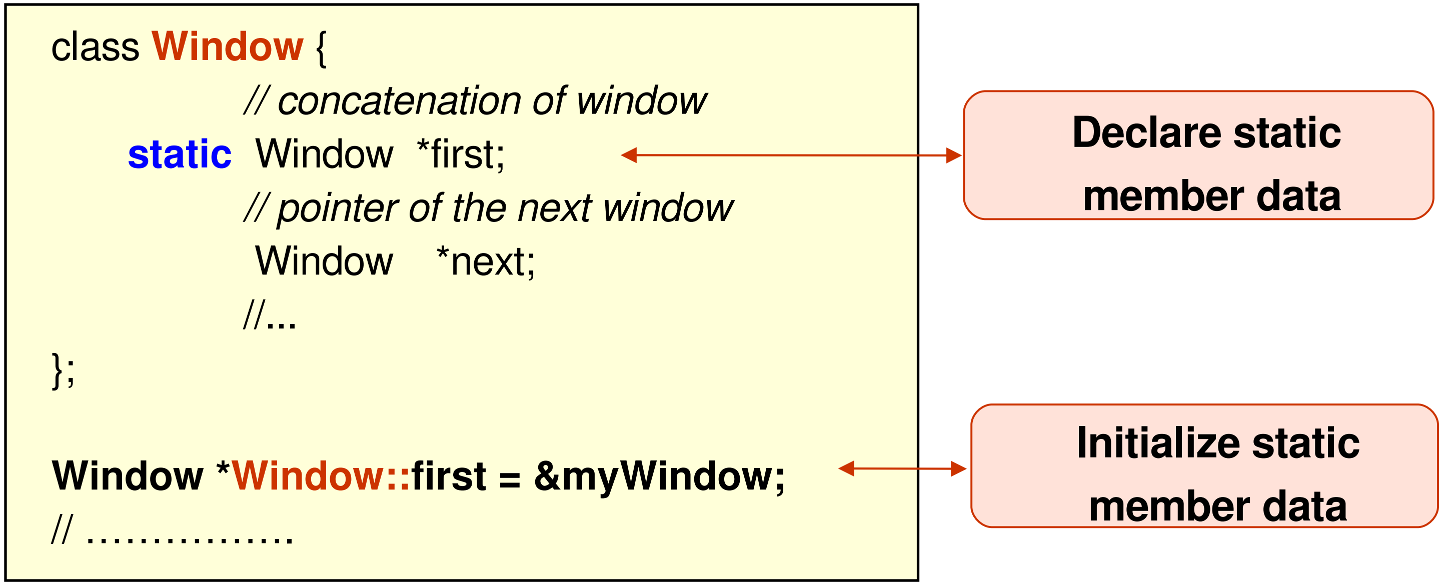
```
void main() {
    const Set s;
    s.AddElem(10);     // WRONG
    s.Member(10);      // OK

 }
```

# Static member (1/2)

**Static member data:**

- **Only one copy of the data is maintained for all objects of the class.**
- **Usage: <class_name>::<member_data_name>**
- **Often used as a count variable of object.**



```
class Window {
                // concatenation of window
      static  Window  *first;
                 Window    *next;
                // pointer of the next window
                //...
};

Window *Window::first = &myWindow;
// …………….
```

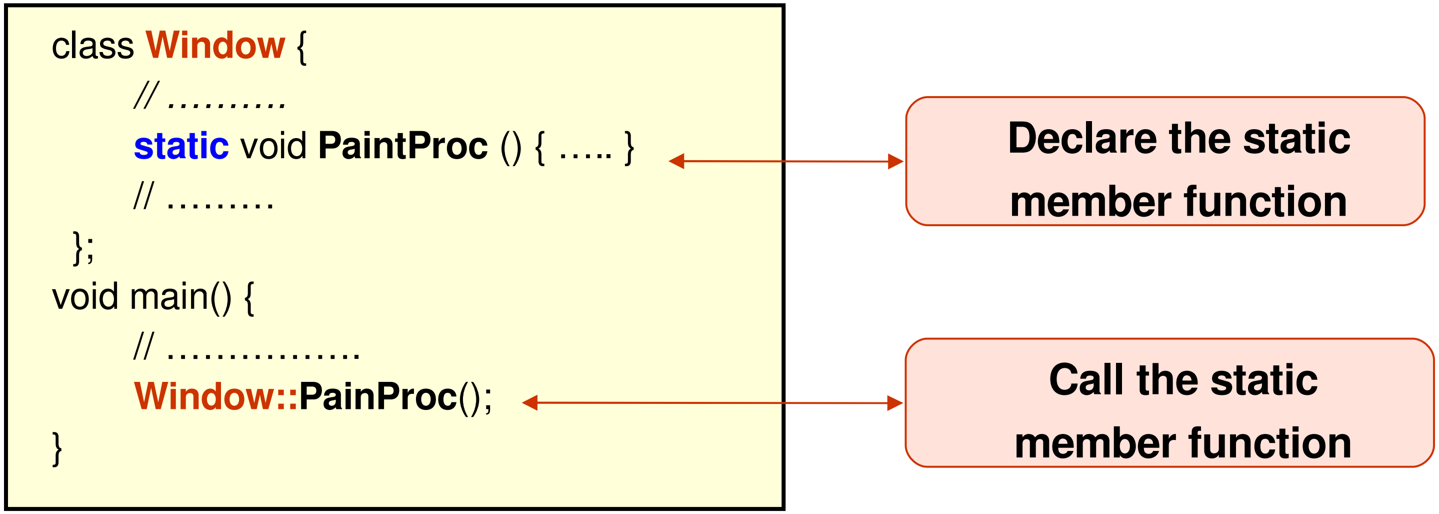
**Declare static member data**

**Initialize static member data**

# Static member (2/2)

**Static member function:**

- **Be considered to have class scope.**
- **Can use only static data members, enumerators, or nested types directly.**
- **Can be accessed without using an object of the corresponding class type.**



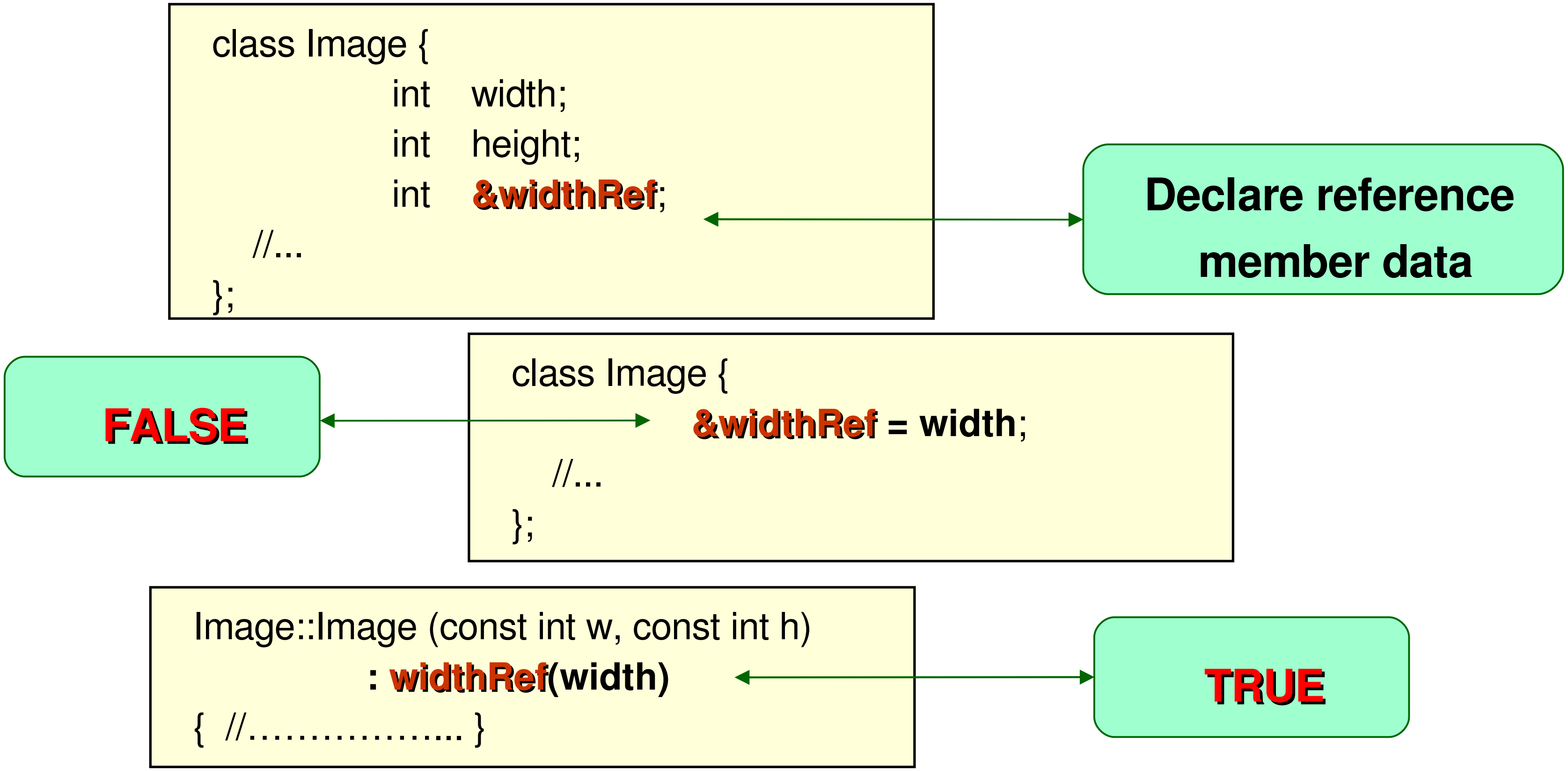
```
class Window {
        // ……….
        static void PaintProc () { ….. }
        // ………
  };
void main() {
        // …………….
        Window::PainProc();
}
```

Declare the static member function

Call the static member function

# Reference member

A **reference member data** holds the address of an object or a variable, but behaves syntactically like an object.



```
class Image {
                int    width;
                int    height;
                int    &widthRef;
    //...
};
```

Declare reference member data

```
class Image {
                &widthRef = width;
    //...
};
```

FALSE

```
Image::Image (const int w, const int h)
                : widthRef(width)
{  //……………….. }
```

TRUE

# Member object (1/2)

Classes can contain member objects of class type.

Ensure that initialization requirements for the member objects are met:

- The contained object's class requires no constructor.
- Or, The contained object's class has an accessible default constructor.
- Or, The containing class's constructors all explicitly initialize the contained object.

# Member object (2/2)

***Example:***

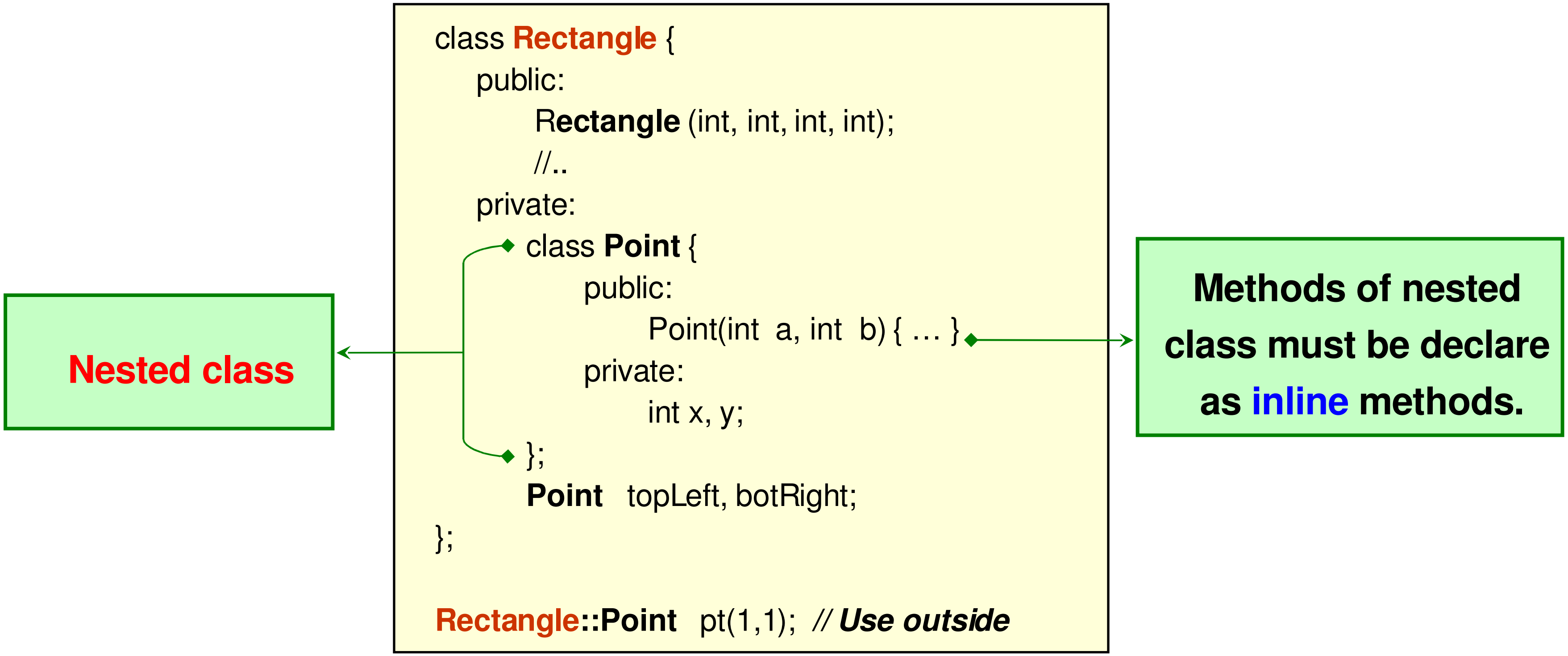
```
class  Point { ……. };
class  Rectangle {
        public:
                 Rectangle (int left, int top, int right, int bottom);
                 //...
         private:
                 Point    topLeft;
                 Point    botRight;
};

Rectangle::Rectangle (int left, int top, int right, int bottom)
                 : topLeft(left,top), botRight(right,bottom)
{ }
```

**Initialize member objects in member initialization list**

# Nested classes

**A class can be declared within the scope of another class. Such a class is called a ”nested class”. Nested classes are considered to be within the scope of the enclosing class and are available for use within that scope.**



```
class Rectangle {
    public:
          Rectangle (int, int, int, int);
          //..
    private:
         class Point {
               public:
                     Point(int  a, int  b) { … }
               private:
                     int x, y;
         };
         Point   topLeft, botRight;
};

Rectangle::Point   pt(1,1);  // Use outside
```

Nested class

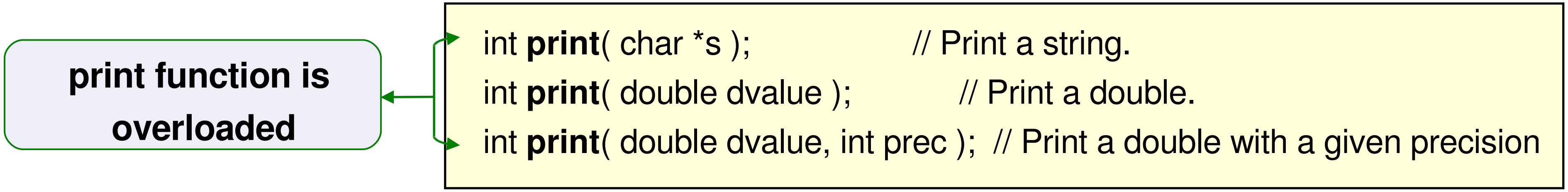
Methods of nested class must be declare as inline methods.

# Outline

- Summary
- Introduction to C++ programming
- Introduction to Object-Oriented programming.
- C++ programming
  - Overloading
    - Function overloading
    - Operator overloading

# Function Overloading

**C++ allows specification of more than one function of the same name in the same scope. This is called Function Overloading.**



**print function is overloaded**

```
int print( char *s );                    // Print a string.
int print( double dvalue );              // Print a double.
int print( double dvalue, int prec );  // Print a double with a given precision
```

**Overloaded functions must be different from input argument list.**

**Quantity** **Order** **Type**

```
int print( double dvalue );
int print( double dvalue, int prec = 3);
```

# Operator Overloading

The *operator-symbol* can be overloaded as a normal function.

**type operator operator-symbol ( parameter-list )**

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Unary** | + | - | * | ! | ~ | & | ++ | -- | () | -> | ->* |
| | new | delete | | | | | | | | | |
| **Binary** | + | - | * | / | % | & | \| | ^ | << | >> | |
| | = | += | -= | /= | %= | &= | \|= | ^= | <<= | >>= | |
| | == | != | < | > | <= | >= | && | \|\| | [] | () | , |

**Some operator-symbol can not overloaded:**

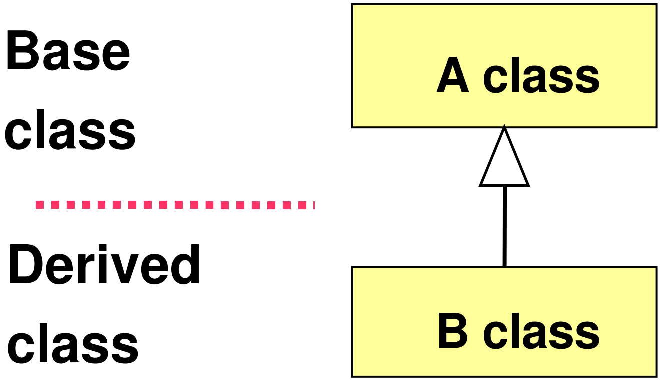
**. .* :: ?: sizeof**

# Outline

- Summary
- Introduction to C++ programming
- Introduction to Object-Oriented programming.
- C++ programming
    - Inheritance
        - Inheritance
        - Constructor & Destructor
        - Public/Protected/Private Base Class
        - Overriding
        - Virtual function

# Inheritance

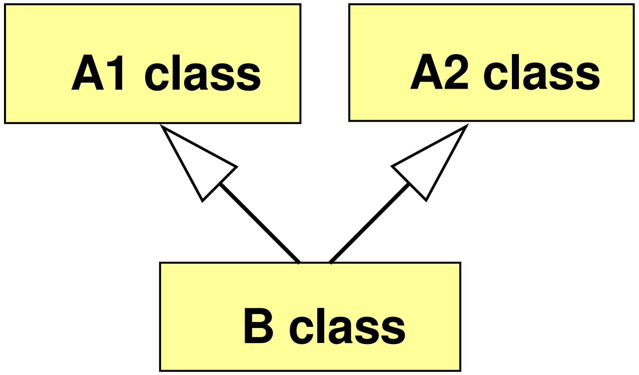
**New classes can be derived from existing classes using a mechanism called "inheritance"**



**Single inheritance**

```
#include “A.h”

class B: <access_control>  A
{
      //...
}
```



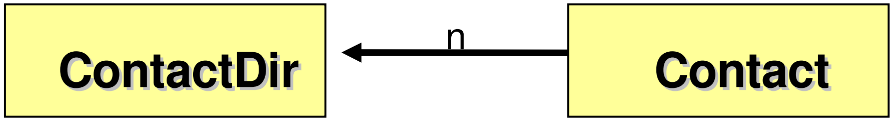
**Multiple inheritance**

```
#include “A1.h”
#include “A2.h”
class B: <access_control>  A1
            , <access_control>  A2
{
      //...
}
```

# Inheritance – Example (1/3)

**Example:**

composition


ContactDir

n

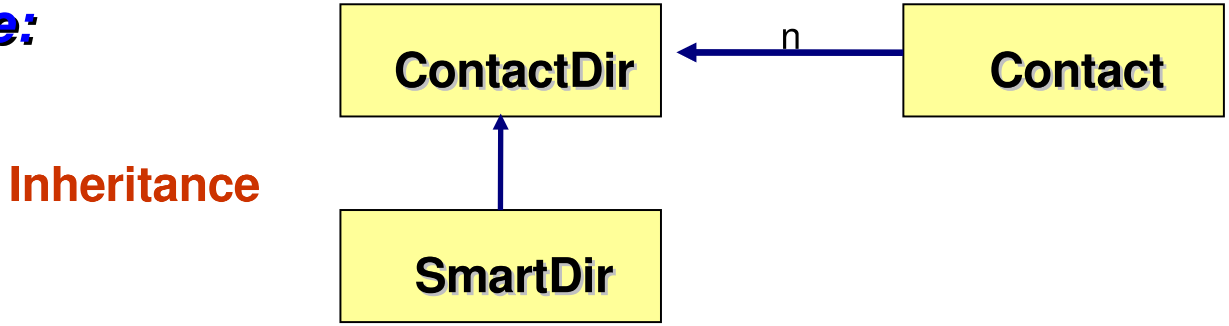
Contact

```
#include <iostream.h>
#include <string.h>
class Contact {
    private:
        char      *name;         // name of customer
        char      *address;   // address of customer
        char     *tel;                   // telephone number
    public:
        Contact (const char *name,
                      const char *address, const char *tel);
        ~Contact ();
        const char*  Name () const     { return name;}
        const char*  Address() const { return address;}
        const char*  Tel() const           { return tel;}
        friend ostream&  operator <<
                                ( ostream&,  Contact& );
};
```

```
class ContactDir {
    private:
        int      Lookup(const char *name);
        Contact    **contacts;     // list of contact
        int      dirSize;    // size of current contact directory
        int      maxSize;  // maximum size of contact directory
    public:
        ContactDir (const int maxSize);
        ~ContactDir();
        void    Insert(const Contact&);
        void    Delete(const char *name);
        Contact* Find(const char *name);
        friend  ostream&  operator <<
                                 (ostream&, ContactDir&);
        // …………
};
```

# Inheritance – Example (2/3)

***Example:***

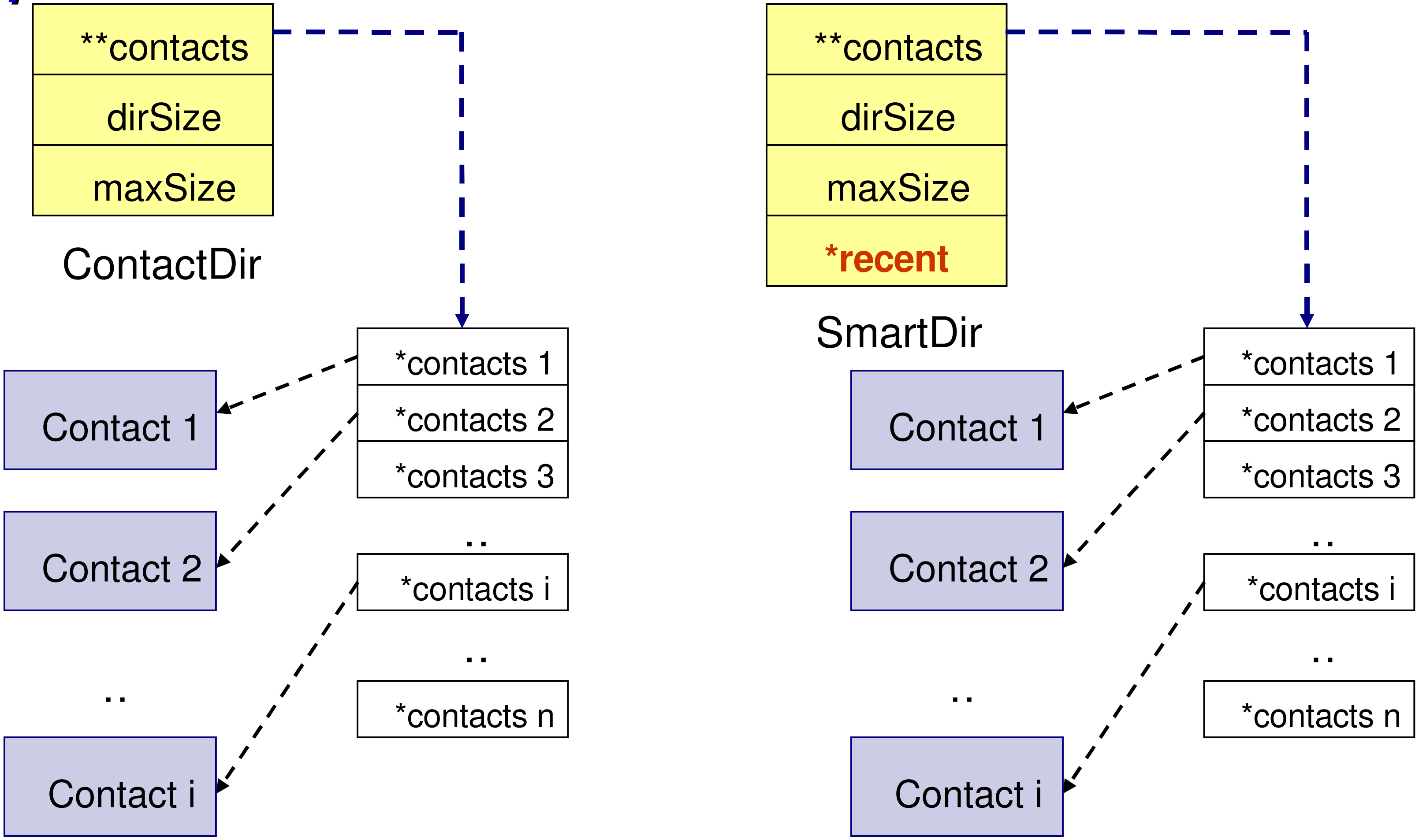


```
class SmartDir : public ContactDir {
    private:
        char  *recent;      //the recent found contact
    public:
        SmartDir(const int max) : ContactDir(max)
            { recent = 0; }
        Contact* Recent (void);
        Contact*  Find  (const char *name);
        // …………….
};
```

```
Contact* SmartDir::Recent (void) {
     return recent == 0 ? 0 :
                  ContactDir::Find(recent);
}
Contact* SmartDir::Find (const char *name) {
     Contact *c = ContactDir::Find(name);
     if (c != 0)
         recent = (char*) c->Name();
     return c;
}
```

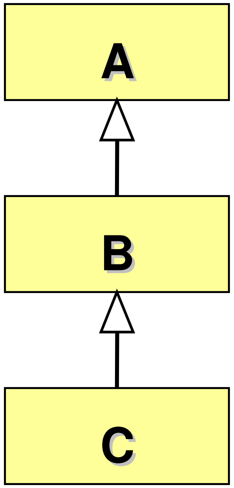
# Inheritance – Example (3/3)

**Example:**

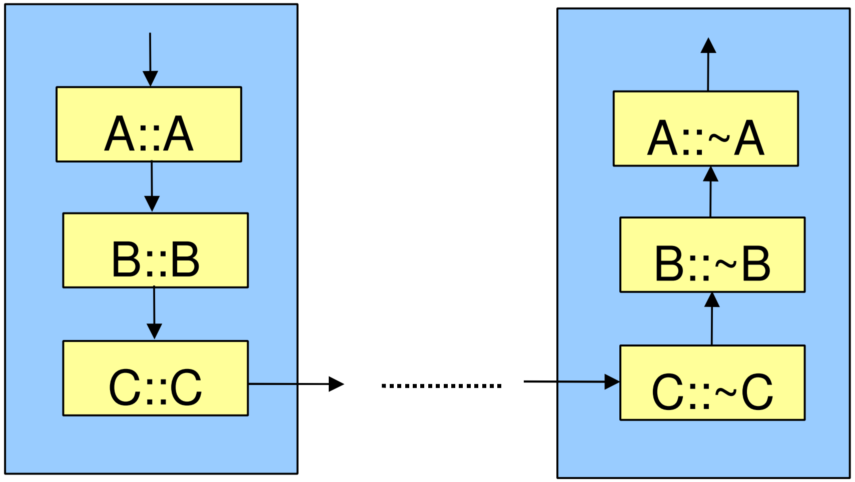


**Description in memory**

# Constructor – Destructor (1/4)



Single inheritance



Calling constructor order

Calling destructor order

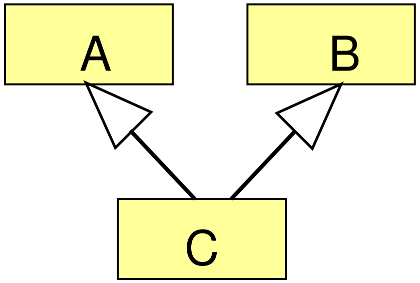
# Constructor – Destructor (2/4)

```
class SmartDir : public ContactDir {
     private:
          char                  *recent;    // /the recent found
contact
     public:
          SmartDir(const int max) : ContactDir(max)
               { recent = 0; }
          SmartDir(const SmartDir& sd): ContactDir(sd)
               { recent = 0; }
          ~SmartDir() {
                 delete recent;
             }
          // ................
};
```

Call constructor of Base Class

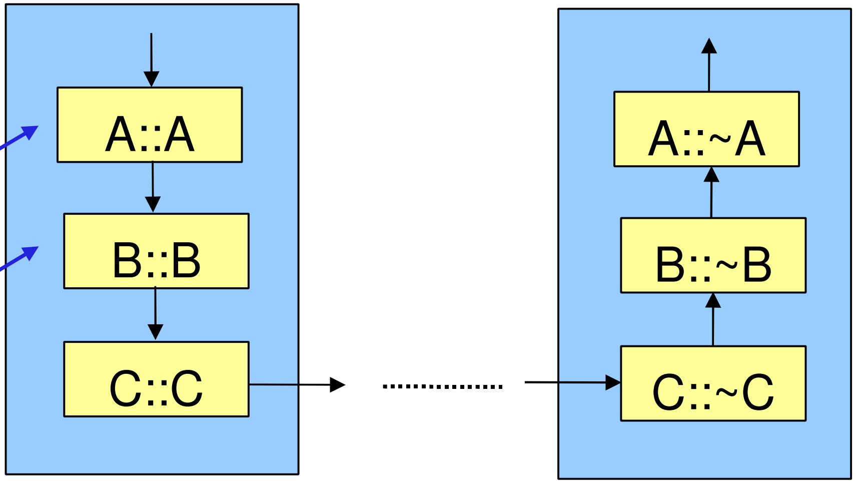
Retrieve memory area of **recent** pointer

# Constructor – Destructor (3/4)



**Multiple inheritance**

```
class A { … };
class B { … };
class C : public A, public B { … };
```



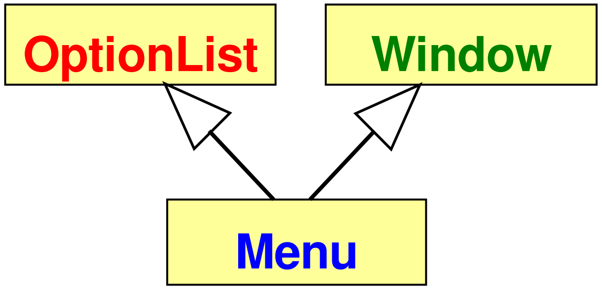
**Calling constructor order**

**Calling destructor order**

# Constructor – Destructor (4/4)

***Example:***

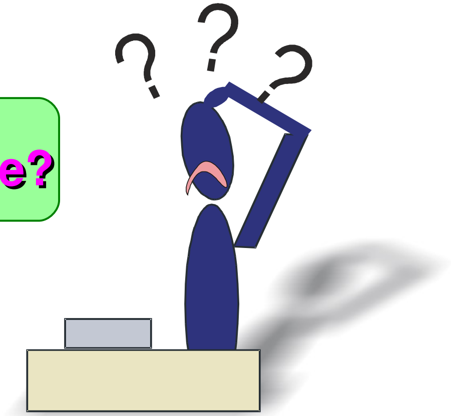
```
class  Menu : public OptionList, public Window {
  public:
      Menu (int n, Rect &bounds);
     ~Menu (void);
      //...
};
```



```
Menu::Menu (int n, Rect &bounds)
 : OptionList(n), Window(bounds){ ...}
```

```
Menu::Menu (int n, Rect &bounds)
 : Window(bounds), OptionList(n){ ...}
```

**What is true?**



# Protected member

**Issue:** Derived class inherits public and private members from Base class. However, derived class can not access private members.

How to access private member?

**Solution:** In Base Class, change private member to protected member.

```
class ContactDir {
                // …
  protected:
       int     Lookup(const char *name);
       Contact   **contacts;
       int     dirSize;
       int     maxSize;
};
```

# Public/Private/Protected Base class

```
class BaseClass {
            private:     int privateX;          void privateFx (void);
            public:      int publicY;           void publicFy (void);
            protected:   int protectedZ;        void protectedFz (void);
};
class DerivedClass1  : BaseClass {};     // default access control is private
class DerivedClass2 : private BaseClass {};
class DerivedClass3 : public BaseClass {};
class DerivedClass4 : protected BaseClass {};
```

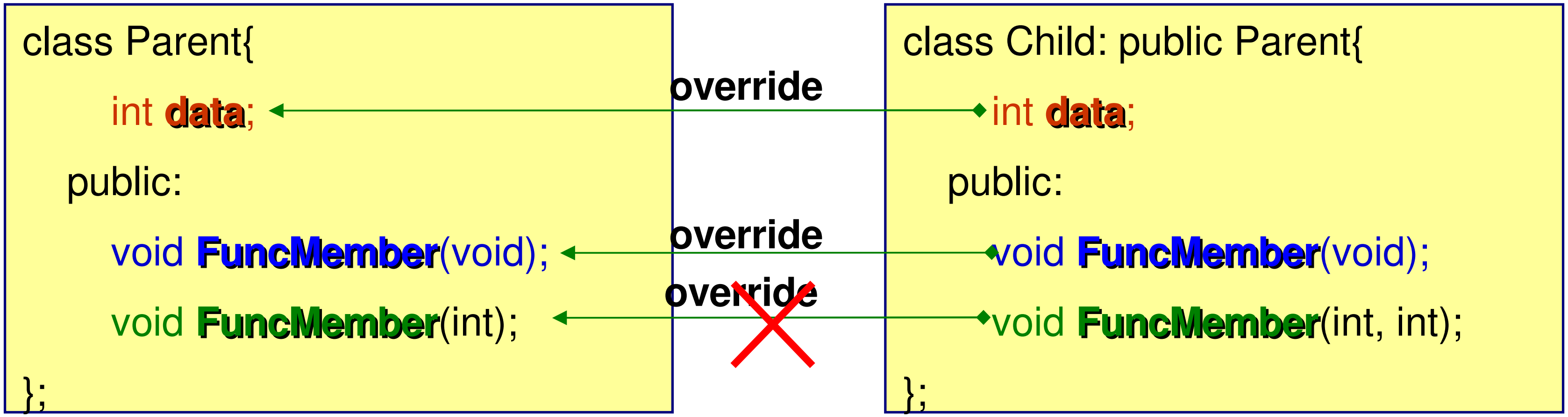
| Derived class<br>Base class | Derive private | Derive public | Derive protected |
|---|---|---|---|
| private member | inaccessible | inaccessible | inaccessible |
| public member | private | public | protected |
| protected member | private | protected | protected |

**Ex: In C class, the member function publicFy is a public member and protectedFz is a protected member because BaseClass is a public base class. privateFx is private to BaseClass, and it is inaccessible to any derived classes.**

# Overriding

Overriding allows a Derived Class to provide a specific implementation of a method that is already provided by Base Class.

The implementation in the Derived Class overrides (replaces) the implementation in the Base Class.



```
class Parent{
        int data;
    public:
        void FuncMember(void);
        void FuncMember(int);
};
```

```
class Child: public Parent{
        int data;
    public:
        void FuncMember(void);
        void FuncMember(int, int);
};
```

**Overloading >< Overriding**

Method overriding is an important feature that facilitates polymorphism in the design of object-oriented programs.

# Virtual Function

**A virtual function is a member function that you expect to be redefined in derived classes.**

- **Syntax: virtual type functionName(argument list)**
- **A virtual function is used in Dynamic Binding**

```
class Father { …
    public:
        virtual void VirtualFunc();
};
```

```
class Child:public Father { …
    public:
        void VirtualFunc();
};
```

**A pure virtual function is virtual function but it is not implemented in Base Class**

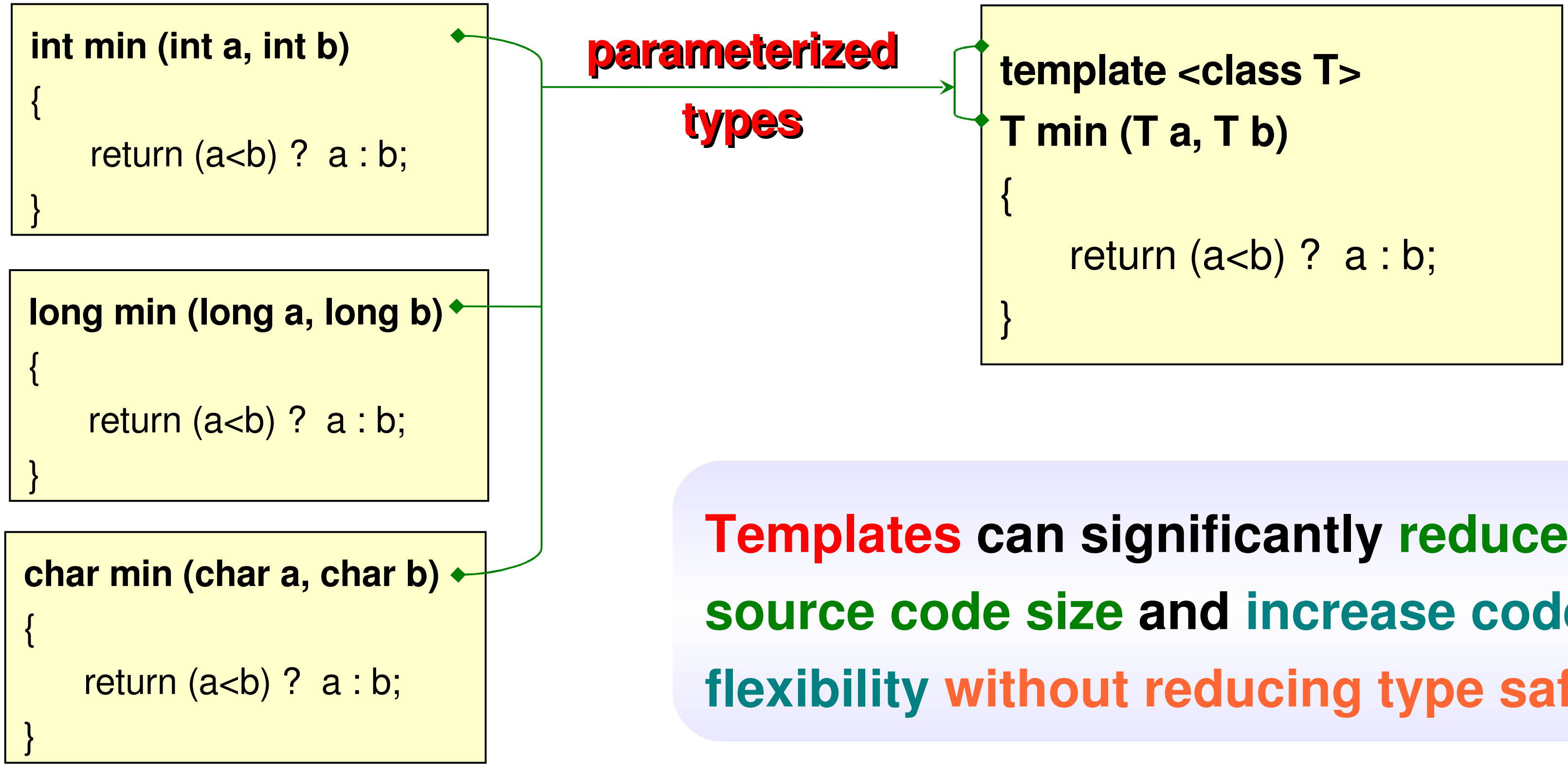
- **Syntax: virtual type functionName(argument list) = 0**

# Outline

- Summary
- Introduction to C++ programming
- Introduction to Object-Oriented programming.
- C++ programming
  - Programming method
    - Template
    - Function pointer

# Template

Templates are mechanisms for generating functions and classes based on type parameters.



```
int min (int a, int b)
{
      return (a<b) ?  a : b;
}
```

```
long min (long a, long b)
{
      return (a<b) ?  a : b;
}
```

```
char min (char a, char b)
{
      return (a<b) ?  a : b;
}
```

```
template <class T>
T min (T a, T b)
{
      return (a<b) ?  a : b;
}
```

Templates can significantly reduce source code size and increase code flexibility without reducing type safety.

# Template

**The template declaration specifies a set of parameterized classes or functions.**

- template-declaration :

  ```
  template < template-argument-list > declaration
  ```

- template-argument-list :

  ```
  template-argument, template-argument, ...
  ```

- template-argument :

  ```
  type-argument argument-declaration
  ```

- type-argument :

  ```
  class identifier
  typename identifier
  ```

```
template <class T, int i = 16>
class TestClass
{
    char buffer[i];
     T testFunc(T* p1 );
};
```

```
template <class T, int i>
T TestClass<T,i>::testFunc(T* p1)
{
    return *(p1++);
};
```

```
// To create an instance of TestClass
TestClass<char, 5> ClassInst;
```

**Reference: http://msdn.microsoft.com/en-us/library/x5w1yety(VS.71).aspx**

# Function pointer

**Function Pointer is a pointer which points to the address of a function.**

```
float Plus       (float a, float b) {return a+b; }
float Minus     (float a, float b) {return a+b; }
float Multiply (float a, float b) {return a+b; }
float Divide     (float a, float b) {return a+b; }
```

```
void Switch(float a, float b, char opCode)
{
    float result;
    switch (opCode) {
        case '+': result = Plus (a,b); break;
        case '-':  result = Minus(a,b); break;
        case '*':  result = Multiply(a, b); break;
        case '/':  result = Divide(a,b); break;
    }
    return result;
}
```

**Must have the same parameters and return-type**

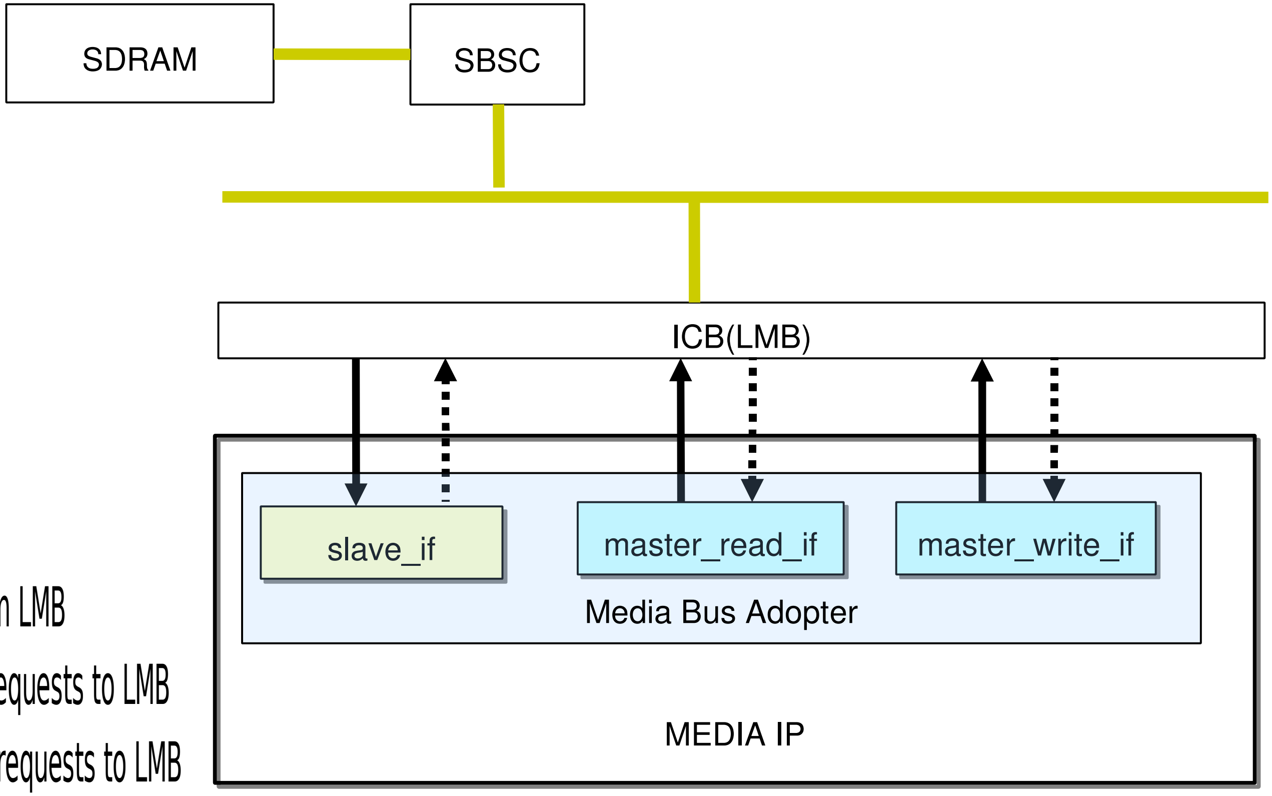
```
float Switch_With_Function_Pointer
    (float a, float b, float (*pt2Func) (float, float))
{
    float result = pt2Func(a,b);
    return result;
}
```

```
Void Replace_A_Switch()
{
    Switch(2,5, '+');
    Switch_With_Function_Pointer(2, 5, &Plus)
}
```

# Function pointer

*This is just an example*

## How to implement function pointer in C++ model



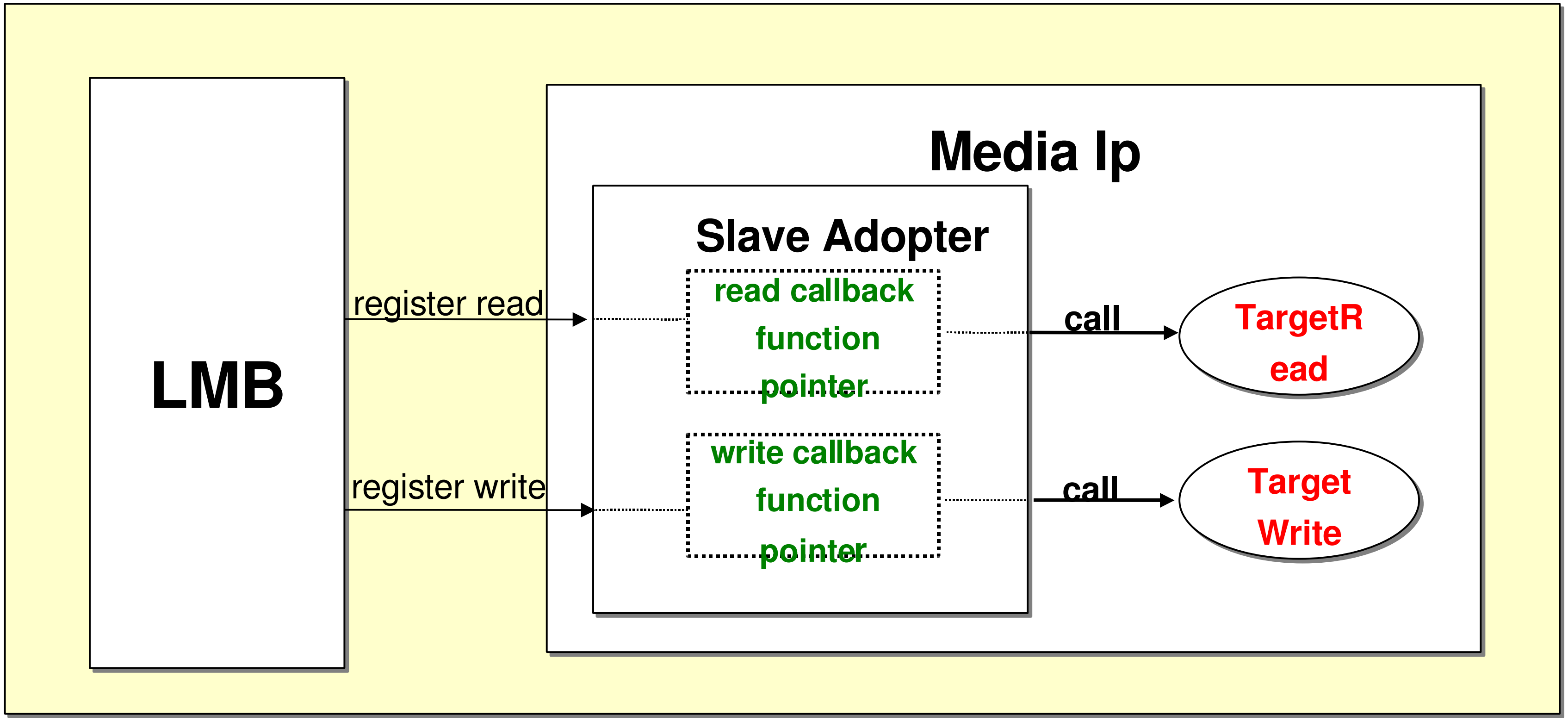
*This is just an example*

- **Legend**:
  - Request : ━━━━
  - Respond : ┆┆┆┆┆┆┆┆
- **Explanation**:
  - **slave_if** : Slave adopter handles requests from LMB
  - **master_read_if** : Master read adopter handles read requests to LMB
  - **master_write_if** : Master write adopter handles write requests to LMB

# Function pointer

*This is just an example*

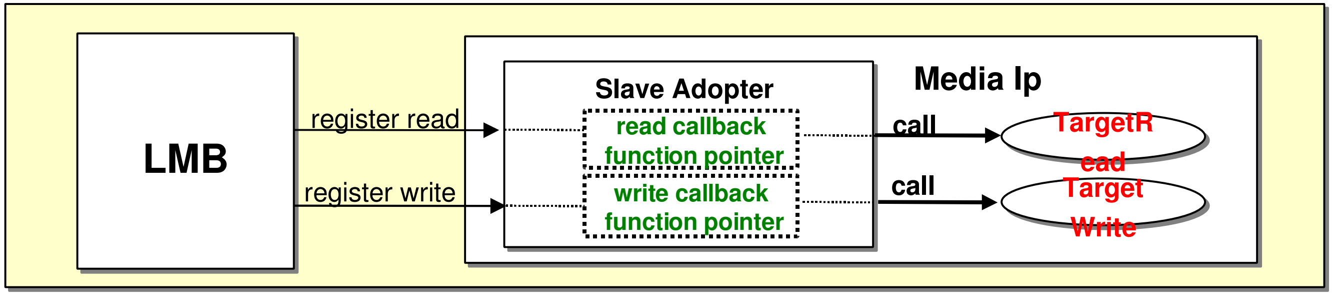
## How to implement function pointer in C++ model



# Function pointer

## How to implement function pointer in C++ model

*This is just an example*



```
#include “slave_adopter.h”

class media_ip
{
   slave_adopter *media_ip_tgt_if;
public:
   media_ip() {
       media_ip_tgt_if = new media_ip_tgt_if(“media_ip_tgt_if”);
       media_ip_tgt_if->SetReadCallBackFunction(this->TargetRead);
       media_ip_tgt_if->SetWriteCallBackFunction(this->TargetWrite);
   }
   static unsigned int TargetRead(unsigned int /*addr*/);
   static TargetWrite(unsinged int /*addr*/, unsigned int /*value*/);
}
```

```
class slave_adopter
{
public:
   unsigned int (*cbf_read) (unsigned int);
   void (*cbf_write)  (unsinged int, unsigned int);
   void SetReadCallBackFunction(unsigned int
                                            (*f_read) (unsigned int))
    {  this->cbf_read = f_read;  }
   void SetWriteCallBackFunction(void (*f_write)
                                         (unsigned int, unsigned int))
   {  this->cbf_write = f_write;  }
}
```

# Outline

- Summary
- Introduction to C++ programming
- Introduction to Object-Oriented programming.
- C++ programming
- Appendix
  - Appendix A: Operator Precedence and Associativity
  - Appendix B: Preprocessor Directives
  - Appendix C: C++ Keywords
  - Appendix D: volatile (C++)

# Outline

- Summary
- Introduction to C++ programming
- Introduction to Object-Oriented programming.
- C++ programming
- Appendix
  - Appendix A: Operator Precedence and Associativity
  - Appendix B: Preprocessor Directives
  - Appendix C: C++ Keywords
  - Appendix D: volatile (C++)

# Operator Precedence and Associativity

| Priority | Operator | | | | | | Type | Associativity |
|---|---|---|---|---|---|---|---|---|
| Highest | :: | | | | | | Unary | None |
| | () | [] | -> | . | | | Binary | Left to Right |
| | +<br>- | ++<br>-- | !<br>~ | *<br>& | new<br>delete | sizeof<br>() | Unary | Right to Left |
| | ->* | .* | | | | | Binary | Left to Right |
| | * | / | % | | | | Binary | Left to Right |
| | + | - | | | | | Binary | Left to Right |
| | << | >> | | | | | Binary | Left to Right |
| | < | <= | > | >= | | | Binary | Left to Right |
| | == | != | | | | | Binary | Left to Right |
| | & | | | | | | Binary | Left to Right |
| | ^ | | | | | | Binary | Left to Right |
| | \| | | | | | | Binary | Left to Right |
| | && | | | | | | Binary | Left to Right |
| | \|\| | | | | | | Binary | Left to Right |
| | ? : | | | | | | Ternary | Left to Right |
| | = | +=<br>- = | *=<br>/= | ^=<br>%= | &=<br>\|= | <<=<br>>>= | Binary | Right to Left |
| Lowest | , | | | | | | Binary | Left to Right |

**Reference: http://msdn.microsoft.com/en-us/library/126fe14k(VS.80).aspx**

# Outline

- Summary
- Introduction to C++ programming
- Introduction to Object-Oriented programming.
- C++ programming
- Appendix
    - Appendix A: Operator Precedence and Associativity
    - Appendix B: Preprocessor Directives
    - Appendix C: C++ Keywords
    - Appendix D: volatile (C++)

# Preprocessor Directives

**Preprocessor directives, such as #define and #ifdef, are typically used to make source programs easy to change and easy to compile in different execution environments.**

**The preprocessor recognizes the following directives:**

| | | | |
|---|---|---|---|
| #define | #error | #import | #undef |
| #elif | #if | #include | #using |
| #else | #ifdef | #line | |
| #endif | #ifndef | #pragma | |

**Directives in the source file tell the preprocessor to perform specific actions.**

**Reference: http://msdn.microsoft.com/en-us/library/3sxhs2ty(VS.80).aspx**

# Outline

- Summary
- Introduction to C++ programming
- Introduction to Object-Oriented programming.
- C++ programming
- Appendix
  - Appendix A: Operator Precedence and Associativity
  - Appendix B: Preprocessor Directives
  - **Appendix C: C++ Keywords**
  - Appendix D: volatile (C++)

# C++ Keywords

**Keywords are predefined reserved identifiers that have special meanings. They cannot be used as identifiers in your program.**

| | | | | | |
|---|---|---|---|---|---|
| asm | continue | float | new | signed | try |
| auto | default | for | operator | sizeof | typedef |
| break | delete | friend | private | static | union |
| case | do | goto | protected | struct | unsigned |
| catch | double | if | public | switch | virtual |
| char | else | inline | register | template | void |
| class | enum | int | return | this | volatile |
| const | extern | long | short | throw | while |

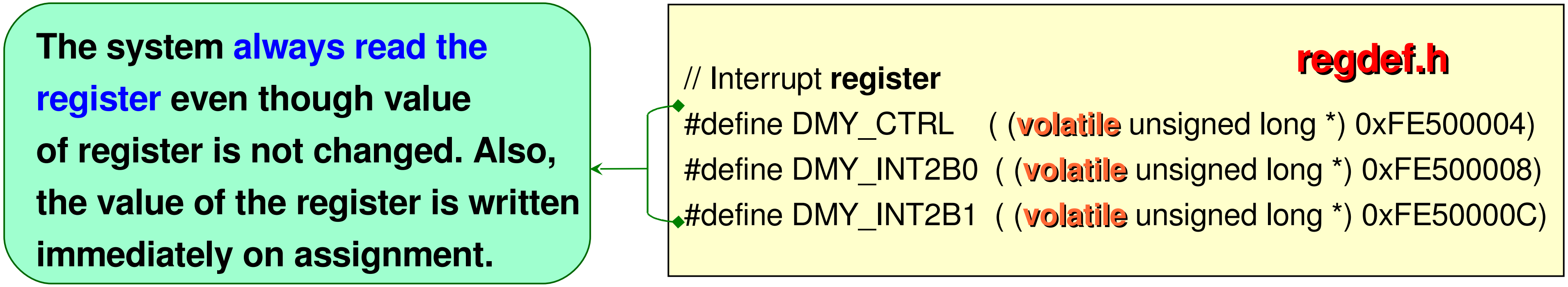
**Reference: http://msdn.microsoft.com/en-us/library/2e6a4at9(VS.80).aspx**

# Outline

- Summary
- Introduction to C++ programming
- Introduction to Object-Oriented programming.
- C++ programming
- **Appendix**
  - Appendix A: Operator Precedence and Associativity
  - Appendix B: Preprocessor Directives
  - Appendix C: C++ Keywords
  - **Appendix D: volatile (C++)**

# volatile (C++)

The **volatile** keyword is a type qualifier used to declare that an object **can be modified in the program** by something such as the operating system, the hardware, or a concurrently executing thread.



The system **always read the register** even though value of register is not changed. Also, the value of the register is written immediately on assignment.

**regdef.h**

```
// Interrupt register
#define DMY_CTRL    ( (volatile unsigned long *) 0xFE500004)
#define DMY_INT2B0  ( (volatile unsigned long *) 0xFE500008)
#define DMY_INT2B1  ( (volatile unsigned long *) 0xFE50000C)
```

Objects declared as **volatile** are **not used in certain optimizations** because their values can change at any time.

Reference: http://msdn.microsoft.com/en-us/library/12a04hfd(VS.80).aspx

# Q&A

# Thank you for your attention and discussion

# Reference

## DMS

**Documents/010_ENG/140_FrontEnd/Project/01_SLD/1_Common/Basic_References/06_Languages/02_C_C++**

## Book (available in Frontend group)

[1] Truong Van Chi Cong, *“Object Oriented Programming C++”*, Can Tho university, 2003 *(in Vietnamese)*

[2] Pham Van At, *“C++ and Object Oriented Programming”*, GTVT, 2009 *(in Vietnamese)*

## Web (available in RVC):

[1] *MSDN library,* http://msdn.microsoft.com/en-us/library/60k1461a.aspx

[2] *Teach Yourself C++ in 21 Days,* http://newdata.box.sk/bx/c/index.htm

[3] *C++ In Action,* http://www.relisoft.com/book/index.htm

[4] *Data Structures and Algorithms with Object-Oriented Design Patterns in C++,* http://www.brpreiss.com/books/opus4/index.html

[5] *Wikipedia,* http://en.wikipedia.org/wiki/Object-oriented_programming

# Revision History

| Revision | Contents | Approved | Checked | Created |
|---|---|---|---|---|
| Ver. 1.0 | • New creation | Vu Pham<br>09/09/2009 | Vu Pham<br>09/09/2009 | Loi Huynh<br>08/10/2009 |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

Renesas Design Vietnam Co., Ltd.