Programming Methodology

Spring 2009

C++ Basics

Topics

- From C to C++
- Basic features of C++
- Class
- Inheritance
- Multiple Inheritance
- Virtual function
- Operator overloading

C++

- An object-oriented descendant of C, which was developed from CPL (Combined Programming Language: a bulky language with high-level operations and bit operations useful for efficient system programming).
- a hybrid language with the duality
  - procedural/imperative programming paradigm
  - object-oriented programming paradigm
- features are not really new – mainly borrowed
  - most of syntax and semantics from C
  - data encapsulation was already well-known
  - notion of derived class and virtual functions from Simula67
  - operator overloadings and declaration within blocks from Algol68
- Why so popular today? … easier transition from C, and perfect C code reusability, one crucial feature of a language

History

- created at Bell laboratories in AT&T (1983)
  - At about 1980, it was developed with the name of C with Classes (AT&T) – Without operator overloading, reference type and virtual function
  - At 1983, “C++” was used.
  - At 1985, C++ Release 1.0

    → Providing operator overloading, reference type and virtual functions
- The name “C++” was made because the syntax of “C” was almost reused in this language and more improved than C, so it was added using incremental operator “++”
- Adjusted to not only “system programming” and “object oriented programming”

Differences of C++ from C

- class construct
  - the underpinning for object-oriented programming
    - information holding with private/shared
    - defines data objects and the associated operations (called *member functions* → e.g, **num_of_elements(), is_element()**)

– similar to the struct construct in C, but more general

```
struct Integer_Set {          //in C
   int* array_of_ints;
   int num_of_elements();     //error!
   …
}
… s.array_of_ints …           //OK!
```

C++ →

Class construct example

```
class Integer_Sets {
 private:
    int* array_of_ints;
 public:
    int num_of_elements();
    int is_element(int num);
…
}
Integer_Set s;
…
if (s.is_element(4)) …
 … s.array_of_ints   //error!
```

– defines data objects and the associated operations

Differences of C++ from C

- operator overloading for class objects

```
Integer_Set operator + (Integer_Set& s, Integer_Set& t) {
  …        //defined as union operators
}
Integer_Set operator + (Integer_Set& s, int t) {
  …        //defined as include operators
}
  eg.      operator+({1,2}+{2,4}) -> {1,2,4}
           operator+({1,2}+5) -> {1,2,5}

Integer_Set S,T;
int i, j, k;
  …
S = (S + i) + (T + (j + k));
```

- function: overloading and flexible number of parameters

```
int foo(char c);
int foo(float x, int j = 0);
…
a = foo(2.6, 7) + foo('p') + foo(3.3);
```

- inlining – useful only when carefully used

```
inline int dec(int & val) { … }
dec(j); -> j--;
cf:        inline int dec(int val) { … }
```

Differences of C++ from C

- call-by-reference

```
void inc1(int & val) { val += 1; }
void inc2(int * val) { *val += 2; }
  …
int j = 0;
int* jp = &j;
inc1(j);                    //call-by-reference
inc2(jp);                   //call-by-value
```

- explicit type conversion
- stream I/O – cin, cout

```
#include <stream.h>
char* c = "cis635";
cout << c << *c << (int) c << (int) *(c+1);
```

→ *output: cis635 c 134516088 105*

- enum for enumeration types

```
typedef enum { yellow, red, blue } Basic_Color;
Basic_Color color = red;
…
if (color == blue) …
```

Class

- With the concept of Class, programmer can create a new data type directly.
- You must do this to express new concept specifically, which can not be expressed with data types included in C++.
- If a new data type is defined well and closely to the concept, the program gets simple and plain and easy to understand.
- Fundamental idea of defining a new data type is concerned with dividing the names needed to use the object correctly and specifications accompanied implementing this object.

Member functions in C?

```
struct date { int month, day, year; };    // struct definition

date today;                              // struct variable declaration

void set_date(date*,  int, int, int);    // 3 functions processing date type var
void next_date(date*);
void print_date(date*);
```

Problems: There is no device connecting data type with function related to this definitively.

Member functions in the C++ `struct`

```
struct date {
   int month, day, year ;           // members

   void set(int, int, int, int);    // member function (or procedure, method)
   void get(int*, int*, int*);
   void print( );
   void next( );
};
```

Example: calling member function:

```
date today;          // definition of date type object  today (allocation of  memory storage)
….
today.set(9, 18, 1990);      // providing the same type of arguments, initialization
today.next( );
```

Member function

- It is possible that member functions with the same name are defined in different structures.
- So when you define a member function, you must appoint the name of the structure where this member function is included.

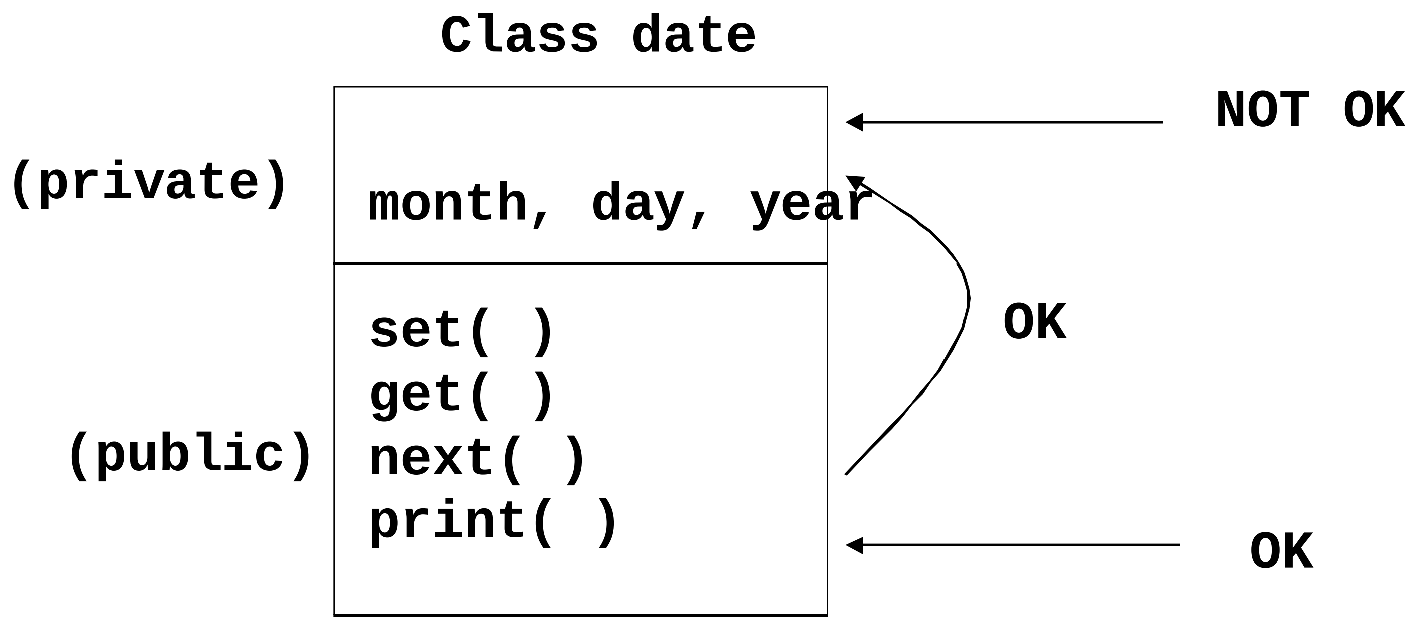
```
void date::next( )              // next() belongs to struct  date
{
      if (++day > 28) {         // the usage of day, member of  date struct
              …
      }
}
```

Information hiding

- It is not true that only member functions within date structure can access member of date structure.
- If you want to do this, you must use **`class`** instead of **`struct`**.

```
class date {
        int month, day, year ;                  // (private) member

 public:
        void set(int, int, int, int);           // (public) member function
        void get(int*, int*, int*);
        void print( );
        void next( );
};
```

Use of private and public members



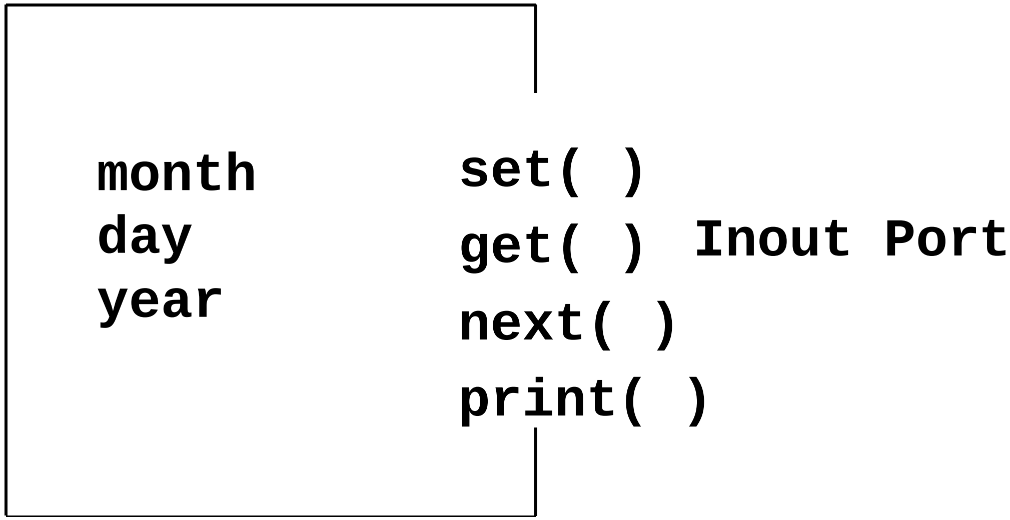
Information hiding within a class

- The function which is not a member function cannot use a private member of date class.

```
void backdate( )    // backdate( ) is not a member function of date
{
     today.day- -;   // error
}
```

- Advantages
  - Protection of interior data or decrease of possibility of error occurrence attributed to hiding.
  - You only have to understand user guider of member functions, which increases convenience because you do not need to know interior implementation/data structure.

Information hiding of Class



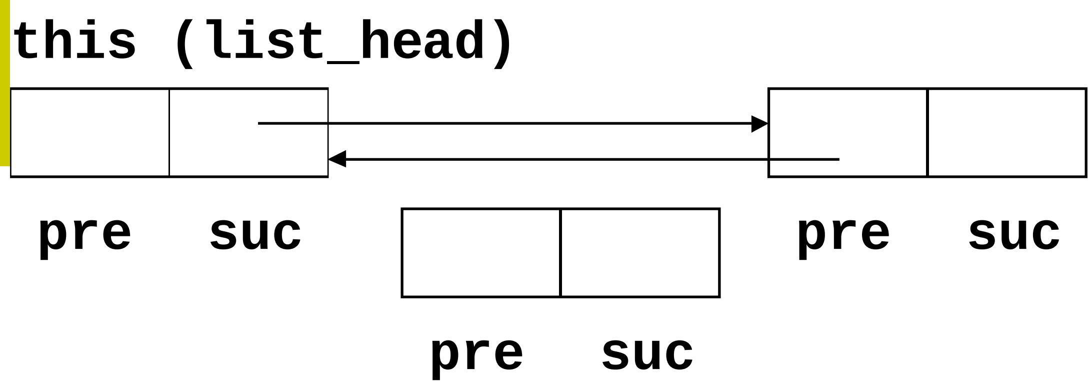
Self referencing pointer - `this`

```
class x {
    int m;
 public:
    int readm() { return m; }     // or return  this->m;
};                                // This represents the address of
                              // currently used object


x aa;
x bb;

void f()
{
     int a = aa.readm();          // substitution a for m in aa
     int b = bb.readm();          // substitution b for m in bb
     ….
}
```

List using “this”

```
class dlink {
     dlink* pre;
     dlink* suc;
 public:
    void append(dlink*);
};

void dlink::append(dlink* p)
{
      p->suc = suc;
      p->pre = this;
      if (suc)
        suc->pre = p;
      suc = p;
}
```

```
void f(dlink* a, dlink* b)
{
      ……
      list_head->append(a);
      list_head->append(b);
}
dlink* list_head;
……
f(……);
```



Instantiation of a class object

```
class date {
     int month, day, year;
public:
    void set_date(int m, int d, int y)
    {
          month = m;
          day = d;
          year = y;
     }
};
date lee;
lee.set_date(9, 6, 1957);
```

In C++, programmers can declare directly a member function (say, “constructor”) which is called automatically when the object is declared and instantiated.

↑ *Solution*

Problems: If you instantiate a class object using a member function, programmers get used to a mistake with missing an invocation of this function or invoking it multiple times.

Constructor

```
class date {
     int month, day, year;
  public:
    date(int m, int d, int y) {      // constructor(member function)
          month = m;
          day = d;
          year = y;
     }
};
```

- The *constructor* is a member function whose name is the same as the class.
- The method of calling a constructor

```
date today = date(23, 6, 1990);
date xmas(25, 12, 0);
date my_birthday;          // error, not assigned arguments to constructor
```

Multiple constructors

- It is possible to assign several constructors.

```
class date {
     int month, day, year;
   public:
     date(int, int, int);
     date(char*);
     date(int);
     date( );
}
```

- Instantiating an object by calling a proper constructor according to the data type and the number of arguments.

```
date today(4);            // date(int)
date july4("5 Nov");      // date(char*)
date now;                 // date( )
```

Member initialization

Generally, a constructor initializes the values of the member variables of the class object.

```
date::date(int m, int d, int y): month(m), day(d),
year(y)
{  }


date::date(int m, int d, int y) {
     month = m;
     day = d;
     year = y;
}
```

Example with a constructor

```
#include <iostream.h>
class x {
        int m;      // private member
  public
        x(int mm) {m = mm; }
        int readme( ) { return m; }
};

main () {
      x aa(3);    x bb(5);    x cc = aa;
      int a = aa.readme();
      int b = bb.readme();
      int c = cc.readme();
      cout <<  “a is  ” <<  a <<  “\n”;
      cout <<  “b is  ” <<  b <<  “\n”;
      cout <<  “c is  ” <<  c <<  “\n”;
 }
```

Another example with a constructor

```
#include <iostream.h>
class Date {
        int mo, da, yr;
  public
        Date() {
             cout << "\nDate constructor" ;
             mo = 0; da = 0; yr = 0;
        }
        Date(int m, int d, int y) { mo = m; da = d; yr = y; }
         // or Date(int m, int d, int y): mo(m), da(d), yr(y) {}
        void print() {
             cout << "\n" << mo << "/" << da << "/" << yr;
        }
};
main () {
     Date days[2];
     Date
temp(6,24,90);
     days[0] = temp;
     days[0].print();
     days[1].print();
}
```

Experiment result

```
Date constructor
Date constructor
6/24/90
0/0/0
```

Destructor

- In contrast to a constructor, a *destructor* eliminates an object that is no more needed by the program.
- The destructor of the class 'X' is expressed as ~X().
- While the constructor allocates a memory location from free space, the destructor deletes this memory allocation.
- It is automatically invoked when the program ends and returns the memory location it has used.

Example with a destructor

```
#include <iostream.h>
class Date {
        int mo, da, yr;
public
        Date( ) { mo = 0; da = 0; yr = 0; }
        Date(int m, int d, int y) {
             mo = m; da = d; yr = y;
         }
         ~Date( ) { cout << “\nDate destructor ” ; }
        void print( ) {
             cout << “\n” << mo << “/” << da << “/” << yr;
        }
};
main () {
     Date days[2];
     Date temp(6,24,90);

     days[0] = temp;
     days[0].print();
     days[1].print();
}
```

Experiment result

```
6/24/90
0/0/0
Date destructor
Date destructor
Date destructor
```

Friend

- The same member function often must be defined together in two or more classes.
- There is no need to define functions performing the same operations in each class.
- In this case, it is effective to make one function, called a *friend*, and use it together.
- It's the case where you must use "friend".
- The function declared by "friend" has the status as the same as one declared within the class.
- Namely, it can access private members of the class.

Examples with friend functions

```
class matrix;
class vector {
      float v[4];
       …
      friend vector multiply(matrix&, vector&);
};
class matrix {
      vector v[4];
       …
      friend vector multiply(matrix&, vector&);
};
vector multiply(matrix& m, vector& v)
{
       …
}
```

Outside the class

```
class x {
      …
      void f( );
};
class y {
      …
      friend void x::f( );
};
void x::f( ) {
       …
}
```

Inside the class

Another example

Using all member functions of one class as a friend of other classes

```
class x {
      …
      void f( );
      void g( );
      …
};

class y {
      …
      friend class x;
};
```

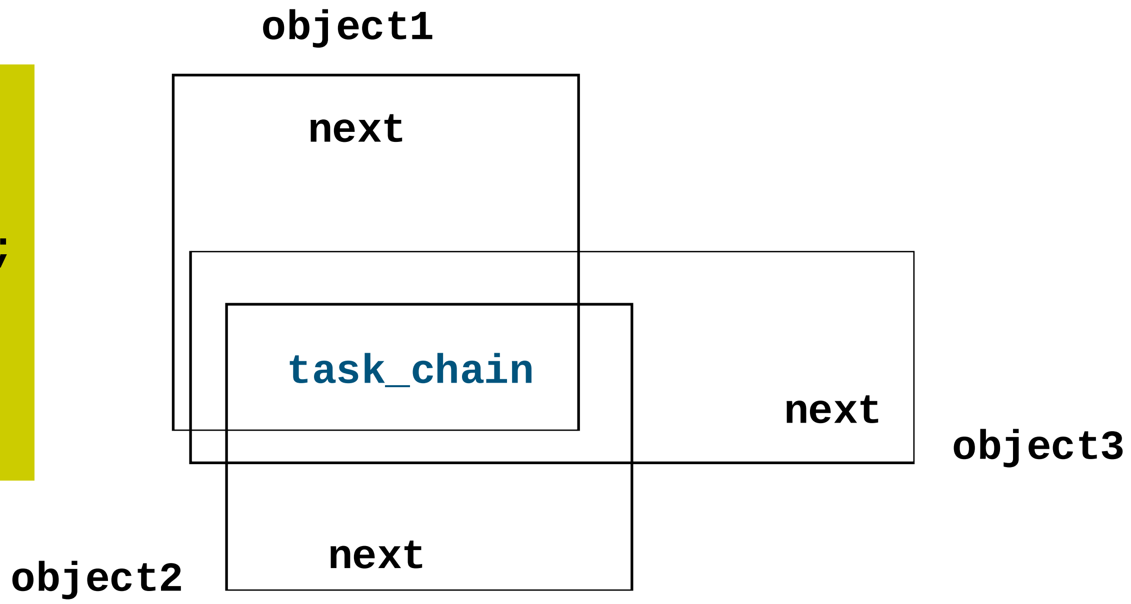
All member functions of class **x** *become a friend of* **y**, *but the member variables of* **x** *have nothing to do with* **y**.

Static member

- Recall that a class is only a type, not an object.

  → So several objects of the same class includes their own member.
- In some case, it is very comfortable for objects of the same type to hold one data in common.
- This shared data is declared by using "**static**".

```
class  task {
        ….
        task* next;
        static task* task_chain;
        void schedule(int);
        void wait(event);
        ….
};
```



Static member

- The scope of a static member is confined within the defined class.
- But if it is declared as a public member, it can be used in the outside of class by using the :: operator.

  → Example: `p = task::task_chain;`

- Static members are always instantiated to “0” by the compiler when the class is declared. But during program execution, it can be modified.

  → Example:

```
task job;
task *task::task_chain = &job;
```

Public/Private base class

Public base class

```
class manager : public employee {
                ...
};
```

```
class employee {
   ...
  public:
   char* name();
   ...
};
```

➔ meaning: *All <u>public</u> members in 'employee' become <u>public</u> members in class 'manager'.*

Ex:

```
void name_print(manager* p) {
    cout << p->name();  // using public member, 'next', in 'employee'
}
```

Public/Private base class

Private base class

```
class manager : private employee {
                    // or by default without private
     …
};
```

➔ meaning: All <u>public</u> members in ‘employee’ become <u>private</u> members in class ‘manager’.

Ex:

```
manager* man;
cout << man->name();
‘manager’
```

// error: **‘next’** is a private member in

Public/Private base class

Part public base class

```
class manager : private employee {
        …
    public:
        …
        employee::name();        // ‘next’ as a public member
};
```

Ex:

```
manager* man;
cout << man->name();  // no error
```

Protected members

While protected members is used like private members by users, those in a class derived from this class is used like public ones.
→ *merely those are impossible to access from the outside.*

```
class two {
   public:              // public
     char *name;
     void f2( );
   protected:           // protected
     float prot1, prot2;
   private:             // private
     float priv;
};
```

```
class three : public two
{
  public:
    void f3( );
  private:
    float z3;
};
```

```
void three::f3( ) {
  name = “korea”;         // public: ok
  prot1 = prot2 = 1;      // protected → public in derived class: ok
  priv = 5;               // private: not ok
}
main() {
  three sun;
  sun.name = “olympic”;              // public: ok
  sun.prot1 = sun.prot2 = 0;         // protected member is private for users: not ok
  su.priv = 5.0;                     // private: not ok
```

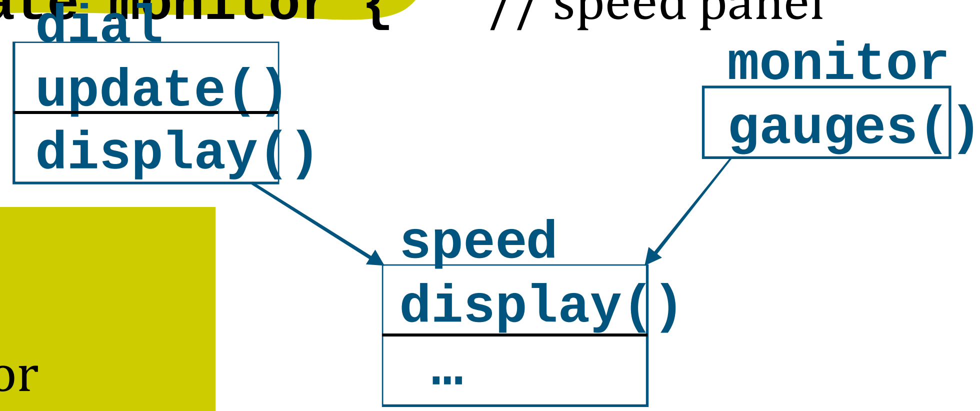
Derived class initialization

- If there is a constructor in base class, when defining a derived class, we must call it.

```
class base {
     ….
  public:
     base(int);    // base class constructor
     ~base( );
 };

class derived : public base {
    int m;
  public:
    derived(char *n) : base(10), m(20) { …. }
};
```

Multiple inheritance in C++

```
class dial {          // accumulated running distance mark of speed panel
  public:
   int update();
   virtual void display() {  …  }
};
class monitor {    // current speed mark
  public:
   int gauges();
};
class speed : public dial, private monitor {    // speed panel
   void display( ) {  …  }
        …
}
```




```
speed meter;
meter.update();         // from dial
meter.guages();         // from monitor
dial *dp = &meter;
monitor *mp = &meter;
dp->display();   // call speed::display()
mp->display();   // call speed::display()
```

Constructor for multiple inheritance

Classes ‘**b1**’ and ‘**b2**’ are made and each constructor is defined.
Class ‘**d**’ is derived from those two classes.

```
class d : public b1, public b2
{
        d(char*);
        int x;
};
```

Two possible initializations for **d**

```
d::d(char *s) : b1(strlen(s), atof(s)), b2(s[0])
{
        …
}
```

```
d::d(char *s) : b1(strlen(s), atof(s)), x(3), b2(s[0])
{
        …
}
```

Operator overloading

- The usage of pre-defined operator to the class by modifying the meaning intentionally.
- C++ can apply operators (**+**,**-**,*****,**/**,etc..) directly to the fundamental data type like "int", but it does not provide operators directly applicable to string, array, or user-defined types.
- But it provides the functionality to define the operator suitable for the class type.
- If you use a user-defined operator applicable to a class object, you can use class objects more conveniently and elegantly than simply using normal functions.

Example

```
class Date {
     int mo, da, yr;
public:
     Date( ) { }
     Date(int m, int, d, int y) { mo=m; da=d; yr=y; }
     void print( )  { cout << mo << “/” << da << “/” << yr; }
     Date operator+(int);     // operator overloading
};
static int dys[] = {31,28,31,30,31,30,31,31,30,31,30,31};

Date Date::operator+(int num)
{
      Date dt = *this;
      num += dt.da;
      ….
      return dt;
}
```

+ is applied to the **this** *object (first argument)*

```
main( )
{
     Date oldd(2,20,91);
     Date newd;
     newd = oldd + 12;
     newd.print( );
}
```

Operator overloading w/ 2 arguments

```
main( )
{    ….
     newd = 12 + odd;
 ?    ….
}
```

this → ?

```
class Date {
     int mo, da, yr;
public:
     Date( ) { }
     Date(int m, int, d, int y) { mo=m; da=d; yr=y; }
     void print( )  { cout << mo << "/" << da << "/" << yr; }
     …
     Date operator+(int n, Date& dt) { return dt+n; }  // Operator overlapping
};
```

- If it is defined not as a friend but as a member function, implicitly including the first argument, it has three arguments. → Then it becomes a ternary operation.
- In C++, only a unary operation and a binary operation are permitted.

Operator overloading with a friend

```
class complex {
      double re, im;
 public:
      complex(double r, double i) { re=r; im=i; }
      friend complex operator+(complex, complex);
};

complex operator+(complex c1, complex c2) {
       return complex(c1.re+c2.re, c1.im+c2.im);
}

void f( )
{
      complex a = complex(1, 3.1);
      complex b = complex(1.2, 2);
      complex c = b;
      a = b + c;     // a = (1.2+1.2, 2+2)
      b = b + c + a;  // observing general operation precedence
}
```

Overlapping operators

- For most embedded operators in C++, we can declare a function that defines the meaning of an operator, and overlap it.
  → Ex : +, -, *, /, &, <<, >>, &&, &=, *=, [ ]
- But we cannot change the priority or grammar of operator.
- Using an operator looks simpler than calling it as an ordinary function.

```
void f (complex a, complex b)
      complex c = a + b;           // Example that is reduced using operator
      complex d = operator+(a, b);    // Example that directly calls
                                      // operator function
 }
```

Binary/unary operators

- Binary operator is defined as (1) a member function with two arguments or (2) a friend function with two arguments.

```
int operator+(int);
friend int operator+(int, Date);
```

- The prefix of unary operator is defined as (1) member function with no argument or (2) a friend function whose first argument is itself.

```
&Date operator++( );    // 'this' is inserted as argument     friend
&Date operator++(&Date);
```

- Postfix of unary operator is defined as (1) member function with one 'int' argument or (2) a friend function whose first argument is itself and the second argument is 'int'.

```
&Date operator++(int);
friend &Date operator++(&Date, int);
```

Example

```
class X {
     friend X operator-(X);         // unary(-) operator
     friend X operator-(X,X);       // binary(-) operator
     friend X operator-( );         // error : no argument
     friend X operator-(X,X,X);     // error : ternary operator

     X* operator&( );       // unary operator : address calculation
     X operator&(X);       // binary operator: logical multiplication (AND)
     X operator&(X,X);      // error: ternary operator
};
```

```
class Date {
     Date( ) { }
     …
     // Date is the first argument.
     Date operator+(int);     // binary
     Date operator++( ) {
        *this = *this + 1; return *this; }    // Prefix
     Date operator++(int) {
        Date r = *this; *this = *this + 1; return r; } //
Postfix
     …
};
```

Overriding operator (=)

```
class string {
      char* p;
      int size;       // Array size is indicated by pointer p
      string(int sz) { p = new char[size = sz]; }
      ~string( ) { delete p; }
};


void f ( )
{
     string s1(10);
     string s2(20);
     s1 = s2;         // the loss of the pointer value of s1 which is assigned
}
```

→ Solution?

Overriding operator (=)

Solution: overriding operator

```
class string {
       char* p;
       int size;

       string(int sz) { p= new char[size = sz]; }
       ~string( ) { delete p; }
       void operator=(string&); // substitution operation overlapping
};
void string::operator=(string& a)
{
      if (this = = &a) return;   // s = s
      delete p;                                   // s1 disappearance
      p = new char[size = a.size];
      strcpy(p, a.p);
}
```

Subscript operators

Subscript operators, [] ,are binary operators.

If there is aa[bb], ‘aa’ is the first operator argument and ‘bb’ is the second operator argument and subscript.

```
class String {   char *s;
      String(char *p) {
             s = new char[strlen(s) + 1];
             strcpy(s, p);
      }
      char& operator[ ] (int n) {
            return *(s + n);
       }
      …..
 };
```

```
main( ) {
     String mstr(“The xxxx of April”);
     mstr[4] = ‘2’; mstr[5] = ‘5’;
     mstr[6] = ‘t’; mstr[7] = ‘h’;
}
```