C++ Classes: Advanced Concepts

Sebti Foufou

NYU Abu Dhabi

sfoufou@nyu.edu

More About Classes

1. The “this” pointer
2. Constant objects and constant functions
3. Friend classes and friend functions
4. Composition
5. Operator overloading
6. Overloading input/output operators
7. Classes and memory allocation

1. The “this” pointer

Every member function MF has a hidden parameter, called **`this`**, whose value is the address of the object that was used to call the function. This parameter allows the MF to access the data members DM of the object by which the function was called..

In a class Date the next two definitions of the MF setDay() are perfectly equivalent.

```
void Date::SetDay(int d){
   day = d ;

}
```

```
void Date::SetDay(int d) {
    this->day = d ;
    //(*this).day = d ;
}
```

`this` is a constant pointer whose value is the object itself. In the function `SetDay()` for example `this` is of type `Date const *`:

```
int main() {
  Date dt(12, 04, 1998) ;

  dt.SetDay(15); //in SetDay() “this” pointes to dt
  return 0 ;
}
```

2. Constant MF and constant objects

```
class Date {
   public :
     …

     // getters functions
     int GetDay()   const ;
     int GetMonth() const ;
     int GetYear()  const ;
     …
} ;
```

```
// get Day
int Date::GetDay() const {
    return(Day);
}
// get Month
int Date::GetMonth() const {
    return(Month) ;
}
// get Year
int Date::GetYear()  const {
    return(Year) ;
}
```

- Constant objet : add the keyword `const` in the object declaration.

```
const Date dEndWar(07,05,1945) ;
dEndWar.SetDay(08) ;          // Error
```

Error: a non-constant MF called by a constant object

```
int DFin = dEndWar.GetDay() ;  // Ok, no error
```

No error: `GetDay()` is a constant function called by a constant object.

inline Member Functions

- Definition: the compiler can, under certain conditions, replace each call to inline function with the source code included in the body of the function.
- Which will reduce the time needed to call this function during the program runtime.
- inline functions should be kept simple

- Any MF defined within its class is considered as inline function.

```
class Date {
    public :
       …
       inline int GetDay()   const { return Day ;}
       inline int GetMonth() const { return Month ;}
       inline int GetYear()  const { return Year ;}
       …
 } ;
```

`GetDay, GetMonth` et `GetYear` are `inline` MF.

It is preferable to add the keyword `inline` in front of these functions (see the code above).

inline Member Functions

- Any MF defined in the same file as its class with the keyword `inline` in front of its declaration is considered `inline`.

```
class Date
{
   public :
      …
       // get functions
      inline int GetDay()   const ;
      inline int GetMonth() const ;
      inline int GetYear()  const ;
      …
} ;

inline int Date::GetDay()   const { return Day ;}
inline int Date::GetMonth() const { return Month ;}
inline int Date::GetYear()  const { return Year ;}
```

3. Friend Functions and Friend classes

```
class A {
   public:
     A(int v) {
         value = v;
     }
     int getval() {
         return (value);
     }

   private:
     int value;
};
```

```
void decrement(A &a)
{
    a.value--;
}

class B
{
  public:
    void touch(A &a) {
       a.value++;
    }
};
```

- Compilation Error : Functions `decrement()` and `B::touch()` try to access private DM `value` of class `A`

Friend Functions and Friend classes

```
class A {
   friend void decrement(A& obj);
   friend void C::f() ;
   friend class B ;

   public:
     A(int v) {
         value = v;
     }
     int getval() {
         return (value);
     }
   private:
     int value;
};
```

```
void decrement(A &a) {
    a.value--;
}
class B {
  public:
     void touch(A &a) {
         a.value++;
     }
};
class C {
  public:
     ...
     void f() ;
};
```

- Remove the Error : To authorize `decrement()`, all the functions of class `B` and function `C::f()` to access `private` DM of class `A`, We declare them, in class `A`, as a friend using the keyword **`friend`**.

4. Composition

```
class Circle {
  private:
    double radius, Xcenter, Ycenter, Zcenter ;

  public:
    Circle(double r, double x, double y, double z) {
      radius = r ;
      Xcenter = x ; Ycenter = y ; Zcenter = z ;
    }
    …
};
```

Class Circle without composition

Composition (2)

```
class Point
{
  private: // private DM
    double X, Y, Z ;

  public:
    Point() {X=Y=Z=0 ;}  // default constr

    //
    Point(double xx, double yy, double zz)
    {
      X = xx; Y = yy; Z= zz ;
    }

    // Set function for the 3 DM
    void SetPoint(double xx, double yy, double zz)
    {
      X = xx ; Y = yy; Z= zz ;
    }
};
```

Composition (3)

```
class Circle {
  private:
    double radius ;
    Point  center ;
  public:
    // 1st constr.
    Circle(double r, double x, double y, double z) ;
    Circle(Point pt, double r) ; // 2nd constr.
    …
};
```

Class Circle with composition

```
// First constructor
Circle::Circle(double r, double x, double y, double z) {
   radius = r ;
   center.SetPoint(x, y, z) ;
}
Circle::Circle(Point pt, double r) { // 2nd constructor
   center = pt ;
   radius = r ;
}
```

Composition (4)

- 1st constructor of `Circle :` calls function `Point::SetPoint` to initialize the DM `center`.
- 2nd constructor: call the default assignment operator of class `Point` to initialize DM `center`.
- The drawback of these two constructors is that the initialization of `center` has been done twice in each constructor:
  - First, the initialization is done by the default constructor of class `Point` which is implicitly called by the constructor of the class `Circle`.
  - Second, the initialization is done again by the call to the function `SetPoint` or the assignment operator.
- To avoid such a double initialization, it is better to explicitly call the constructor of the class `Point` before getting into the constructor of the `Circle` as follows:

```
// 1sf constructor
Circle::Circle(double r, double x, double y, double z): center(x,y,z) {
  radius = r ;
}
// 2nd constructor
Circle::Circle(Point pt, double r) : center(pt) {
  radius = r ;
}
```

5. Operator Overloading

The next slides cover the following topics

- Mechanisms for operator overloading
- Overloading operators as members functions
- Overloading operators as general functions
- List of operators that one can overload.

Operator overloading mechanisms

To define an operator `op`, we need to define a function `operator op()` :

- Either as a general function (usually declared as friend), in this case writing `a op b` is equivalent to

```
operator op (a, b)
```

- Or, as a member function of the class, in this case writing: `a op b` is equivalent to :

```
a.operator op (b)
```

- Examples : in the class of complex numbers, we define/overload operators +, - and * as MF, then we define them as general, but friend, functions.

Overloading operators as member functions

```
class  complex {
   private:
      double r, i ;

   public:
      // constructor
      complex( double x=0 ; double y= 0) ;

      //Operator
      complex operator + (complex const &c) ;
      complex operator - (complex const &c) ;
      complex operator * (complex const &c) ;
      complex operator * (double const  d) ;

      // Display
      void display() ;
 } ;
```

Overloading operators as member functions

```
// constructor
complex::complex(double x, double y) {
   r = x ;     i = y ;
}

// addition operator.
// e.g. about how we can return a pointer
complex complex::operator + (complex const &c) {
   complex res ;
   res.r = r + c.r ;     res.i = i + c.i ;
   return(res) ; //Shorter: return complex(r+c.r, i+c.i)
}

// Operator -
complex  complex::operator - (complex const &c)      {
   complex res ;
   res.r = r - c.r ;     res.i = i - c.i ;
   return(res) ;
}
```

Overloading operators as member functions

```
// Operator *
complex  complex::operator * (complex const &c) {
   complex res ;
   res.r = r * c.r - i * c.i ;
   res.i = r*c.i + i*c.r ;
   return(res) ;
}

// Operator *
complex  complex::operator * (double const  d)   {
   complex res ;
   res.r = r *d ;
   res.i = i*d
   return(res) ;
}

// Display
void complex::display()      {
   cout << r<<"  + i( "<<i<<") \n" ;
}
```

Overloading operators as member functions

```
// driver function
int main( )
{
   double d = 15.6 ;
   complex a(1, 2) , b(3, 4), c1, c2, c3, c4 ;

   c1 = a+ b ;          //c1 = a.oprator + (b) ;
   c1.display() ;

   c2 = a - b ;         //c2 = a.oprator - (b) ;
   c2.display() ;

   c3 = a * b ;         //c3 = a.oprator * (b) ;
   c3.display() ;

   c4 = a * d ;         //c4 = a.oprator * (d) ;
   c4.display() ;
   return 0 ;
}
```

Overloading operators as general functions

```
class  complex {

  //operators defined as general, friend, functions
  friend complex operator +(complex const &c1,complex const &c2);
  friend complex operator -(complex const &c1,complex const &c2);
  friend complex operator *(complex const &c1,complex const &c2);
  friend complex operator *(complex const &c1,double const  d) ;

  private :
      double r, i ;

  public :
      // constructor
      complex( double x=0 ; double y= 0) ;

      // display
      void display() ;
} ;

// constructor
complex::complex(double x, double y)     {
   r = x ; i = y ;
}
```

Overloading operators as general functions

```
// addition operator
complex operator+ (complex const &c1,complex const &c2){
   complex res ;
   res.r = c1.r + c2.r ;
   res.i = c1.i + c2.i ;
   return(res) ;
}
// Operator -
complex operator- (complex const &c1,complex const &c2){
   complex res ;
   res.r = c1.r - c2.r ;
   res.i = c1.i - c2.i ;
   return(res) ;
}
// Operator *
complex  operator* (complex const &c1,complex const &c2){
   complex res ;
   res.r = c1.r * c2.r - c1.i * c2.i ;
   res.i = c1.r*c2.i + c1.i*c2.r ;
   return(res) ;
}
```

Overloading operators as general functions

```
// Operator *
complex operator* (complex const &c, double const d) {
   complex res ;
   res.r = c.r *d ;     res.i = c.i*d ;
   return(res) ;
}
// display
void complex::affiche() {cout << r<<"  + i( "<<i<<") \n" ;}

// test
int main( )  {
   double d = 15.6 ;
   complex a(1, 2) , b(3, 4), c1, c2, c3, c4 ;

   c1 = a+ b ;          //c1 = oprator + (a, b) ;
   c1.display() ;
   c2 = a - b ;         //c2 = oprator - (a, b) ;
   c2.displaye() ;
   c3 = a * b ;         //c3 = oprator * (a, b) ;
   c3.displaye() ;
   c4 = a * d ;         //c4 = oprator * (a, d) ;
   c4.display() ;
   return 0 ;
}
```

Operators that we can Overload

```
+       -       *       /       %       ^       &       |       ~
!       =       <       >       <=      >=      ++      --      <<
>>      ==      !=      &&      ||      +=      -=      *=      /=
%=      ^=      &=      |=      <<=     >>=     []      ()      ->
->*     new     delete
```

Some operators can only be defined as member functions :

```
=       []      ()      ->
```

6. Overloading input/output operators

6. Overloading input/output operators

- class ostream
- class istream
- Overloading operators << and >>
- Connecting an output stream to a file
- Connecting an input stream to a file
- File opening modes

The ostream class

A stream is a channel which:

- Receives the information. An output stream is an object of the class `ostream`
- Provides the information. An input stream is an object de la class `istream`

- Operator<< is already overloaded for all standard types
- `cout` is an output stream connected to the standard output (the screen)
- `cerr` is an output stream connected to the standard error output (the screen)
- Function `put()` gives the argument it receives to the steam :

```
char c1 = 'T';
cout.put(c1) ;                    // cout <<c1 ;
```

- Function `write()` gives the set of chars it receives to the steam :

```
char* t = "bonjour" ;
...
cout.write(t) ;              // display bonjour
cout.write(t,4) ;            // display bonj
```

The istream class

- Operator `>>` is overloaded for all basic types.
  White spaces serve as delimiters during inputs

- `cin` is an input stream connected to the standard input (keyboard)

- Function `get()` extract one char from an input stream and assign it to a variable:

```
istream& get (char &c) :

char c ;
cin.get(c) ;      // Reads one char from the standard input
                  // and stores it in variable c
```

- Function `getline()` facilitates the reading of strings:

```
istream& getline(char *ch, int size, char delim="\n");
```

The istream class

- Function `gcount()` gives the length of the string stored in the memory by `getline()`:

```
const int LG = 80 ;
char ch [LG] ;
int nb ;
...
while(cin.getline(ch, LG))
{
    nb = cin.gcount( ) ;
    // processing of a set of N chars
}
```

- Function `read()` reads from the input stream a set of chars of specified length.

```
char  t[10] ;
cin.read(t,5) ;  // read 5 chars from the standard input
                 // and store them in variable t.
```

Overloading of operators << and >>

- Overloading operators >> and << as friend functions.

```
ostream & operator << (ostream &s, const class &obj) {
    //
    s << obj.attribut1<< endl ;
    s << obj.attribut2<< endl ;
    …
    return(s) ;
}
```

```
istream & operator >> (istream &e, class& obj)        {
    //
    e >> obj.attribut1 ;
    e >> obj.attribut2 ;
    …
   return(e) ;
}
```

Overloading of operators << and >>

```
class  complex {
    friend ostream& operator<<(ostream &s, const complex &c);
    friend istream& operator>>(istream &e, complex &c) ;
    private :
       double r, i ;
    public :
       // constructor
       complex( double x=0 ; double y= 0) ;
       ...
} ;

// Operator << for the class complex
ostream& operator << (ostream &s, const complex &c) {
   s << c.r << "  + i( " << c.i << ") " << endl ;
   return(s) ;
}

// Oprator >> for the class complex
istream & operator >> (istream &e, complex &c) {
   cout << "     Enter the real part : " ; e >> c.r ;
   cout << "     Entre the imaginary part: " ; e >> c.i ;
   return(e) ;
}
```

Overloading of operators << and >>

```
int main( ) {
      double d = 15.6 ;
      complex a, b ;
      complex c1, c2, c3, c4 ;

      cout<< "Saisir a : " ;
      cin >> a ;                          // oprator >> (cin, a) ;
      cout<< "Saisir b : " ;
      cin >> b ;                          // oprator >> (cin, b) ;

      c1 = a + b ;                        // c1 = a.oprator + (b) ;
      cout<< "c1 = "<< c1<<endl ;
      c2 = a - b ;                        // c2 = a.oprator - (b) ;
      cout<< " c2 = " ;
      cout <<c2 ;                         // operator<<(cout, c2) ;
      c3 = a * b ;                        // c3 = a.oprator * (b) ;
      cout<< " c3 = " <<c3<<endl ;
      c4 = a * d ;                        // c4 = a.oprator * (d) ;
      cout<< " c4 = " <<c4<<endl ;
      return 0 ;
}
```

Connecting an output stream to a file

To connect an output stream to a file we need to:

- include the header file `fstream.h`
- Create an object of the class `ofstream`
- Example :

```
ofstream out("test.dat", ios::out) ;
```

Object `out` is connected to the file `test.dat` of the current working directory

Argument `ios::out` shows that the file is open in writing mode.

Connecting an output stream to a file

```
const  int LGM = 20 ;
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
int main( ) {
   char fileName [ LGM + 1] ;
   int n ;
   cout << "Name of the file you want to create: " ;

   cin >> setw(LGM) >> fileName ; //The string length is limited to LGM chars
   ofstream  outf(fileName, ios::out); // associate output stream to file fileName
   if(!outf) {
       cout <<"impossible to create the file \n" ;
       exit(1) ;
   }
   do {
      cout << "Enter one integer: " ;
      cin >> n ;
      if(n) outf.write( (char *) &n, sizeof(int) ) ;
   } while(n && outf) ;
   outf.close( ) ;
   return 0 ;
}
```

Connecting an input stream to a file

To connect one input stream to a file we need to:

- include the file `fstream.h`
- create an object of the class `ifstream` :
- example :

```
ifstream inf("toto.dat", ios::in) ;
```

Object `inf` is of type `ifstream`. It is associated to the file toto.dat located at the same directory. Argument `ios::in` indicates that the file is opened in reading mode.

Connecting an input stream to a file

```
const  int LGM = 20 ;
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
itn main()  {
     char fileName [LGM+1] ;
     int n ;
     cout << « Enter the file name: " ;
     cin >> setw(LGM) >>fileName ;

     ifstream  inf(fileName, ios::in) ;
     if(!inf) {
          cout <<« File opening impossible \n" ;
          exit(1) ;
     }
     while(inf.read( (char *) &n, sizeof(int) )  {
          cout << "n = " << n <<"\n" ;
     }
     inf.close() ;
     return 0 ;
}
```

File opening modes

Mode	Signification
`ios::in`	Open in reading mode
`ios::out`	Open in writing mode
`ios::app`	Open in append mode
`ios::ate`	Go to the end of file after opening
`ios::nocreate`	The file should exist, no create
`ios::noreplace`	The file should not exist, create (unless if ios: :ate or ios: :app is activated)

7. Classes and memory allocations

7. Classes and memory allocation

- Class with pointer data members
- Constructors and destructor
  - Initialization of pointer data members
  - Memory allocation for pointer data members
  - Releasing memory allocated to pointer data members
- Copy constructor
  - Role
  - Definition
- Assignment operator
- Array of objects
- Array of object pointers

Class with pointer data members

```
class CTeam{
    friend ostream& operator<<(ostream& os,const CTeam &eq);
    friend istream& operator>>(istream& is, CTeam &eq) ;
private:
       int*  m_Index ;      // Index as a pointer
       char* m_Name ;       // Team name
       int   m_NbElem ;     // Number of team members
       int*  m_Members ;    // Ids of team members
public:
       CTeam() ;       // Default constructor
       CTeam(int ind, char* nom) ; // 2 args constructor
       ~CTeam() ;    // Destructor
       void SetName(char* nom) ;  // update the name;
       void SetIndex(int n) ;     // update the index
} ;
```

constructors et destructor

```
// two args constructor
CTeam::CTeam(int ind, char* name) {
        // Initialize the index
        m_Index= new int(); //Shorter: m_Index= new int(ind);
        *m_Index= ind ;

        // Initialize the name
        m_Name = new char [strlen(name)+1] ;
        strcpy(m_Name, name) ;

        // Initialize the members
        m_NbElem = 0 ;
        m_Members= NULL ;
}
// Destructor
CTeam::~CTeam() {
        delete m_Index ;
        delete [] m_Name ;  // Release the memory of name
        // Release the list of team members
        if(m_Members!=NULL) delete [] m_Members ;
}
```

```
// Default constructor
CTeam::CTeam()  {
        m_Index = NULL ;
        m_Name = NULL ;
        m_NbElem = 0 ;
        m_Members = NULL ;
}
```

Input and output operators

```
// input operator
istream& operator>> (istream& is, CTeam &eq) {
  cout<<" Number of elements : " ;
  is>>eq.m_NbElem ;
  if(eq.m_Members) delete [] eq.m_Members ;
  eq.m_Membres = new int [eq.m_NbElem] ;
  for(int i=0; i<eq.m_NbElem; i++) {
     cout<<"Element "<<i<<" : " ;
     is>>eq.m_Members[i] ;
  }
  return is;
}
```

```
// Output operator
ostream& operator<<(ostream& os,const CTeam &eq)
{
   os<<eq.m_Name<<",  "<<*eq.m_Index ;
   if(eq.m_Members!=NULL) {
     os<<",  "<<eq.m_NbElem<<" (" ;
     for(int i=0; i<eq.m_NbElem; i++)
              os<<eq.m_Members[i]<<" ";
     os<<") " ;
   }
   return os ;
}
```

SetName and SetIndex

```
// Update the name
void CTeam::SetName(char* name)
{
    delete [] m_Name ;
    m_Name=new char[strlen(name)+1];
    strcpy(m_Name, name) ;
}
```

```
// update the Index
void CTeam::SetIndex(int ind)
{
     *m_Index = ind ;
}
```

In the above, why we didn't delete m_Index, reallocate new space, and assign ind, like what we have done for m_Name?

The test function

```
// The main
int main() {
  CTeam e1(125, "Dream Team") ; // Create one team
  // test the >> and << operators
  cout<<"Before input: "<<e1<<endl ;
  cin>>e1 ;
  cout<<"After the input: "<<e1<<endl ;
  // Change the name and the index
  e1.SetIndex(150) ;
  e1.SetName("Modified dream team") ;
  cout<<"After the update: "<<e1<<endl ;
  return 0 ;
}
```

Results :

```
Before input: Dream Team,  125
Number of elements : 3
Element 0 : 5
Element 1 : 2
Element 2 : 7
After the input: Dream Team,  125,  3 (5 2 7 )
After the update: Modified dream team, 150,  3 (5 2 7 )
```

Copy Constructor

- The copy constructor is called in each of the following three cases:
- First case :

  When the system creates an object from another (existing) object instance of the same class.

```
class Test
{
        ...
} ;
int main()
{
      Test a ;                    // default constructor
      Test b = a ;                // copy constructor
      Test c(a) ;                 // copy constructor
      Test d = Test(a) ;          // copy constructor
      return 0 ;
}
```

Copy Constructor

- Second case:

  When an object is passed by value as an argument to a function

```
class Test {
        ...
} ;
void f(Test ob) {
        ...
}
int main()      {
   Test a ;                    // default constructor

   // The parameter of function f() is initialized by the
   // data of object a, automatic call to the copy
   // constructor.

   f(a) ;
   return 0 ;
}
```

Copy Constructor

- Third case :

  When a function returns an object instance of a class.

```
class Test {
        ...
} ;
Test f() {
   Test ob ;
   ...
   return(ob) ;
}
int main()      {
   // Object a is initialized from the data of the object
   // returned by function f() by an automatic call to
   // the copy constructor.

   Test a = f() ;
   return 0 ;
}
```

Default copy constructor

```
class Test {
    int v1 ;
    int *v2 ;

public :
    Test(int i, int j) {
        v1= i ;
        v2 = new int(j) ;
    }
    ~Test() {
        delete v2 ;
    }
    int getv1() {
         return(v1) ;
    }
    int getv2() {
         return(*v2) ;
    }
} ;
```

```
void f(Test a) {
    // any instructions
}
```

```
int main ( )
{
    Test x(5,7) ;
    ...
    f(x) ;
    cout<<"v1= "<<x.getv1()<<endl;
    cout<<"v2= "<<x.getv2()<<endl;
    return 0 ;
}
```

The behavior of default copy constructor

- Running the sequences of this program shows the problem of memory management that may result when the default copy constructor is used to copy objects of a class having pointer data members.

- When the call `f(x)` is performed, the argument `a` of function `f()` will be constructed and its data members initialized automatically by copying the values of corresponding data members of objects `x`. So attributes `v2` of the two objects `x` and `a` will then be pointing to the same memory zone.

- A the end of function `f()` the destructor is called automatically to destroy object `a`. Hence, attribute `v2` which is common between `a` and `x` is deleted.
- The data member `v2` of object `x` pointe now to a memory zone that has been released.
- The statement `x.getv1()` returns the value of `v1`, while the statement `x.getv2()` generates an error because the content of `v2` is a non-valid address.

The behavior of default copy constructor

- With the call `f(x)` all the data members of object `x` are copied to initialize the data members of object `a`, The formal argument of function `f()`.
- These copies have been performed by a constructor defined by the system, called default copy constructor. For a class `Test` this constructor is written as:

```
Test::Test(const Test & obj) ;
```

- This constructor has a good behavior for classes with no pointer data members. However this default copy constructor is the source of memory management errors for all the classes with pointer data members. For these classes the overloading of this constructor is mandatory.

Example of copy constructor

```
class CTeam {
        …
    CTeam(const CTeam& eq) ; // copy constructor
```

```
// copy constructor
CTeam::CTeam(const CTeam& eq) {
   m_Index = new int (*eq.m_Index) ; // Copy the index
   m_Name = new char [strlen(eq.m_Name)+1] ; //Copy Name
   strcpy(m_Name, eq.m_Name) ;

   // Copy the number and list of members
   if(eq.m_Membres!=NULL) {
      m_NbElem = eq.m_NbElem ;
      m_Members = new int [m_NbElem] ;
      for(int i=0; i<m_NbElem; i++)
            m_Members[i]= eq.m_Memebrs[i] ;
   }
   else { m_NbElem = 0 ; m_Members = NULL ; }
}
```

The default assignment operator

```
class Test {
        ...
} ;
int main() {
    Test x ;
    Test y ;
    ...
    x = y ;
    return 0 ;
}
```

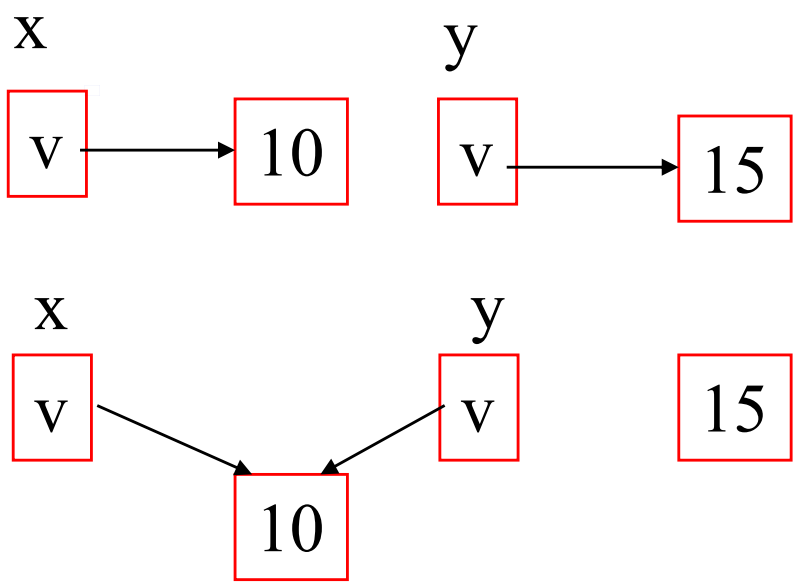
The statement `x = y` is interpreted as a call to a special function, called `operator=()`, as follows: `x.operator=(y) ;`

To redefine the assignment operator of any class `Test`,
we need to define a member function `operator=()` in the class :

```
Test& operator=(const Test& source) ;
```

Behavior of the default assignment operator

```
class Test
{
    int *v ;
public:
    Test(int x) {v= new int ; *v= x;}
    ~Test() {delete v ;}
    Test(const Test& t) {v= new int ; *v= *t.v;}
} ;
int main() {
    Test x(10) ;
    Test y(15) ;
    ...
    x = y ;
    ...
}
```



When x is destroyed the memory zone of `x.v` will be released. y becomes then invalid as its attribute `v` is pointing to a memory zone that was released. The system will generate an error when the destructor tries to destroy object y.

Example: defining assignment operator for CTeam class

```
// Add this declaration to the class CTeam
   CTeam& operator=(const CTeam& eq) ; // Assignment
```

```
CTeam& CTeam::operator= (const CTeam& eq) {
   if(this != &eq) {
     delete m_Index ;                  // Copy the index
     m_Index = new int (*eq.m_Index) ;
     delete [] m_Name ;                // Copy the name
     m_Name = new char [strlen(eq.m_Name)+1] ;
     strcpy(m_Name, eq.m_Name) ;

     // Copy the number and list of elements
     if(m_Members) delete [] m_Members ;
     if(eq.m_Members) {
        m_NbElem = eq.m_NbElem ;
        m_Members = new int [m_NbElem] ;
        for(int i=0;i<m_NbElem;i++) m_Members[i]=eq.m_Members[i];
     }
     else {m_Members= NULL ; m_NbElem = 0 ;}
   }
   return(*this) ;
}
```

Assignment Operator vs Copy Constructor

- We notice the following processes that are common between the copy constructor and the assignment operator in one hand, and between the assignment operator and the destructor in the other hand.

- The duplication of data members is done in:
  - The assignment operator
  - Copy constructor

- The release of the memory occupied by data members is done in:
  - The assignment operator
  - The destructor

End

Next Lecture: Classes and Inheritance (OOP).

Thank you