Virtual Functions

Virtual Function

- A virtual function is a member function which is declared within a base class and is re-defined(Overriden) by a derived class.
- It is declared using the virtual keyword.
- Virtual function is used to achieve runtime polymorphism.

Why virtual function?

- Base class pointer:
- Base class pointer can point to the object of any of its descendant class.
- But its converse is not true.

Problem without virtual keyword

```
class A
{
   public:
     void f1()  { }
};
class B:public A
{
   public:
     void f1()  { }
};
int main()
{
   A *ptr,o1;
   B o2;
   ptr=&o1;
   ptr->f1();       // A::f1()
   ptr=&o2;
   ptr->f1();      // A::f1()
}
```

Problem without virtual keyword

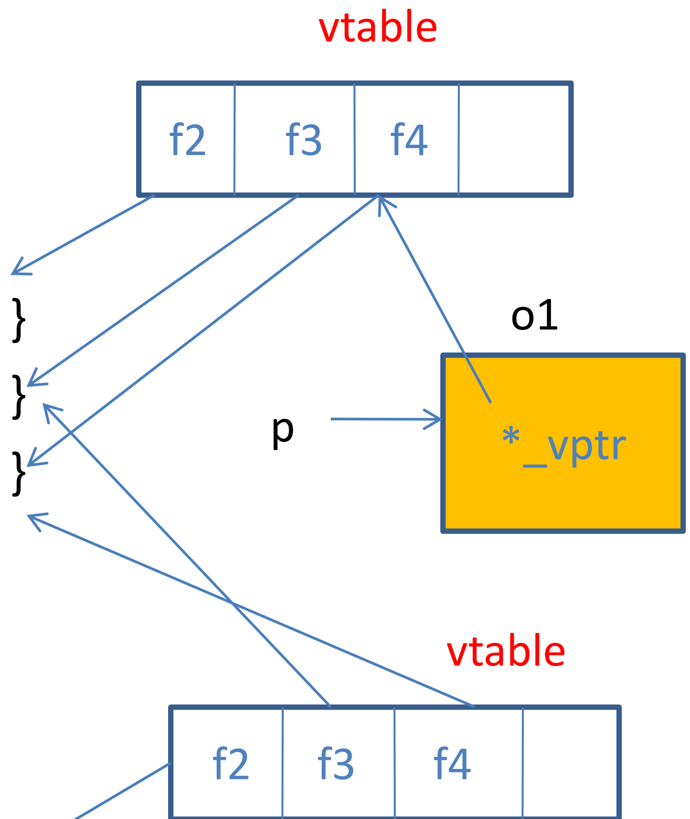
```
class A
{
   public:
     virtual void f1()  { }
};
class B:public A
{
   public:
     void f1()  { }
};
int main()
{
   A *ptr,o1;
   B o2;
   ptr=&o1;
   ptr->f1();       // A::f1()
   ptr=&o2;
   ptr->f1();       // B::f1()
}
```

Late Binding

- The compiler adds code that identifies the kind of object at runtime then matches the call with the right function definition.
- This can be achieved by declaring a virtual function.

Virtual Function working concept

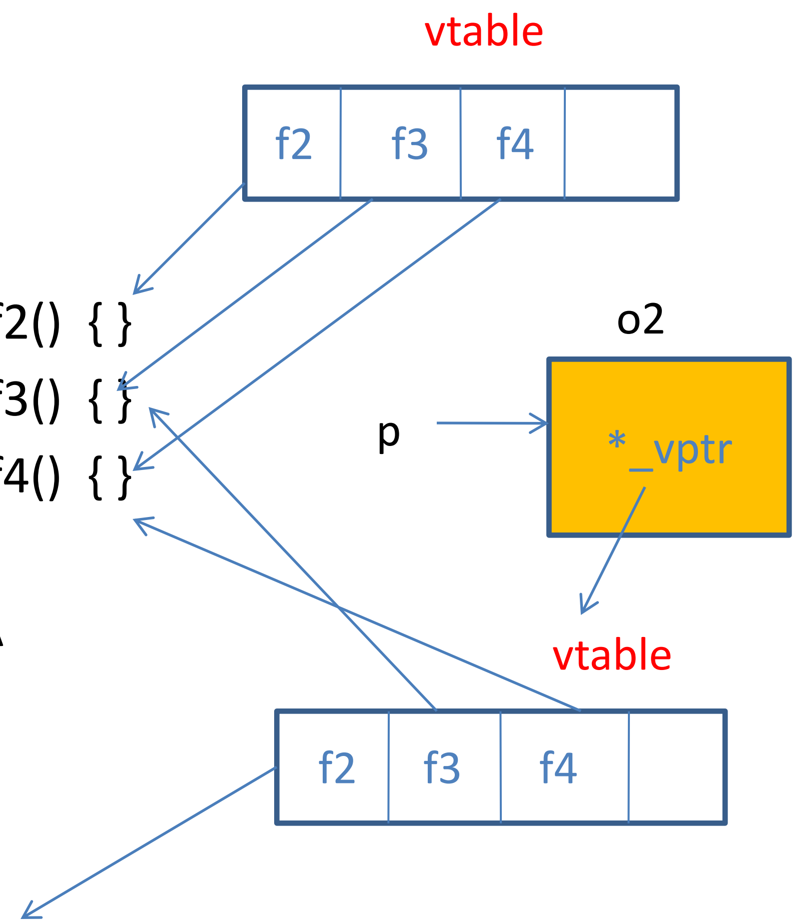
```
class A
{
  *_vptr;
  public:
   void f1()  { }
   virtual void f2()  { }
   virtual void f3()  { }
   virtual void f4()  { }
};
class B:public A
{
  public:
   void f1()  { }
   void f2()  { }
   void f4(int x)  { }
};
```




```
int main()
{
  A *p,o1;
  p=&o1;
  p->f1();     // EB
  p->f2();     // LB
  p->f3();     // LB
  p->f4();     // LB
  p->f4(5);  // EB - Error
}
```

Virtual Function working concept

```
class A
{
  *_vptr;
  public:
    void f1()  { }
    virtual void f2()  { }
    virtual void f3()  { }
    virtual void f4()  { }
};
class B:public A
{
  public:
    void f1()  { }
    void f2()  { }
    void f4(int x)  { }
};
```




```
int main()
{
   A *p,o1;
   B o2;
   p=&o2;
   p->f1();      // EB - A
   p->f2();      // LB - B
   p->f3();      // LB - A
   p->f4();      // LB - A
   p->f4(5); //EB - Errror
}
```

Pure virtual function

- A pure virtual function is a virtual function in C++ for which we need not to write any function definition and only we have to declare it.
- It is declared by assigning 0 in the declaration.
- Abstract class:
- An abstract class is a class in C++ which have at least one pure virtual function.
- We can’t create object of abstract class but pointers and references of Abstract class type can be created.
- If an Abstract Class has derived class, they must implement all pure virtual functions, or else they will become Abstract too.

Pure virtual function

```
class Base
{
public:
virtual void show() = 0;     //pure virtual function
};

class Derv1 : public Base
{
public:
void show()
{ cout << "Derv1\n"; }
};

class Derv2 : public Base
{
public:
void show()
{ cout << "Derv2\n"; }
};
```

```
int main()
{
// Base b1;          //can't make object
                      from abstract class
Base* arr[2];
Derv1 dv1;
Derv2 dv2;
arr[0] = &dv1;
arr[1] = &dv2;
arr[0]->show();
arr[1]->show();
return 0;
}
```

Example

```
class person
{
protected:
  char name[40];
public:
void getName()
{ cout << " Enter name: "; cin >> name; }
void putName()
{ cout << "Name is: " << name << endl; }
virtual void getData() = 0;
virtual bool isOutstanding() = 0;
};

class student : public person
{
private:
  float gpa;
public:
void getData()
{
person::getName();
cout << " Enter student's GPA: "; cin >> gpa;
}
bool isOutstanding()
{ return (gpa > 3.5) ? true : false; }
};
```

```
class professor : public person
{
private:
  int numPubs;
public:
void getData()
{
person::getName();
cout << " Enter number of professor's
publications: ";
cin >> numPubs;
}
bool isOutstanding()
{ return (numPubs > 100) ? true : false;
}
};
```

Contd..

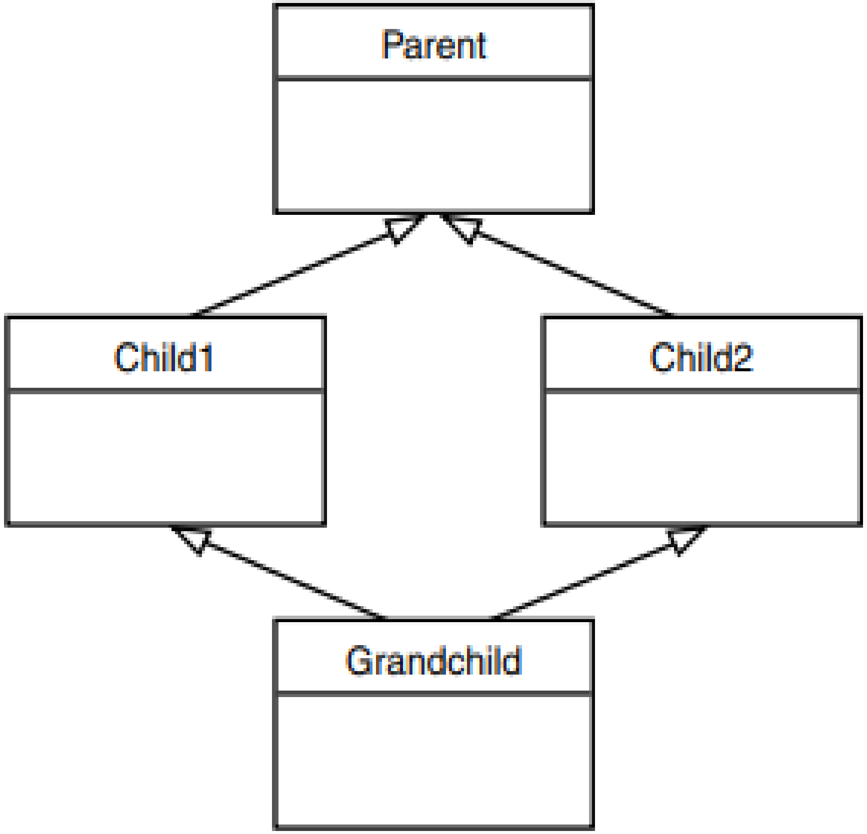
```
int main()
{
person* persPtr[100];
int n = 0;
char choice;
do {
  cout << “Enter student or professor (s/p): “;
  cin >> choice;
  if(choice==’s’)
    persPtr[n] = new student;
  else
    persPtr[n] = new professor;
  persPtr[n++]->getData();
  cout << “ Enter another (y/n)? “;
  cin >> choice;
} while( choice==’y’ );
for(int j=0; j<n; j++)
{
  persPtr[j]->putName();
  if( persPtr[j]->isOutstanding() )
    cout << “ This person is outstanding\n”;
}
return 0;
}
```

Virtual Destructors

```
class Base
{
public:
virtual ~Base()      //virtual destructor
{ cout << "Base destroyed\n"; }
};

class Derv : public Base
{
public:
~Derv()
{ cout << "Derv destroyed\n"; }
};

int main()
{
Base* pBase = new Derv;
delete pBase;
return 0;
}
```

Virtual Base Classes

- Virtual base classes are used in virtual inheritance in a way of preventing multiple “instances” of a given class appearing in an inheritance hierarchy when using multiple inheritances.



Example

```
class Parent
{
protected:
int basedata;
};
class Child1 : public Parent
{ };
class Child2 : public Parent
{ };
class Grandchild : public Child1, public Child2
{
public:
int getdata()
{ return basedata; }    // ERROR: ambiguous
};
```

Example

```
class Parent
{
protected:
int basedata;
};
class Child1 : virtual public Parent        // shares copy of parent
{ };
class Child2 : virtual public Parent        // shares copy of parent
{ };
class Grandchild : public Child1, public Child2
{
public:
int getdata()
{ return basedata; }
};
```

Friend Function

- Data hiding is a fundamental concept of object-oriented programming. It restricts the access of private members from outside of the class.
- There is a feature in C++ called **friend functions** that break this rule and allow us to access private member functions from outside the class.
- A friend function can access the private and protected data of a class.
- It is declared by using friend keyword inside the body of the class.
- If class A is a friend of B, then B doesn’t become a friend of A automatically.
- Friendship is not inherited

Example

```
class beta;
class alpha
{
private:
int data;
public:
alpha() : data(3) { }
friend int frifunc(alpha, beta);
};

class beta
{
private:
int data;
public:
beta() : data(7) { }
friend int frifunc(alpha, beta);
};
```

```
int frifunc(alpha a, beta b)
{
return( a.data + b.data );
}

int main()
{
alpha aa;
beta bb;
cout << frifunc(aa, bb) << endl;
return 0;
}
```

Friend in operator overloading

```
class Distance
{
private:
int feet;
float inches;
public:
Distance() : feet(0), inches(0.0)
{ }
Distance(float fltfeet)
{
feet = int(fltfeet);
inches = 12*(fltfeet-feet);
}
Distance(int ft, float in)
{ feet = ft; inches = in; }
void showdist()
{ cout << feet << "\'-" << inches << '\"'; }
Distance operator + (Distance);
};

Distance Distance::operator + (Distance d2)
{
int f = feet + d2.feet;
float i = inches + d2.inches;
if(i >= 12.0)
{ i -= 12.0; f++; }
return Distance(f,i);
}
```

```
int main()
{
Distance d1 = 2.5;
Distance d2 = 1.25;
Distance d3;
cout << "\nd1 = "; d1.showdist();
cout << "\nd2 = "; d2.showdist();
d3 = d1 + 10.0;   // OK
cout << "\nd3 = "; d3.showdist();
// d3 = 10.0 + d1;   //ERROR
// cout << "\nd3 = "; d3.showdist();
cout << endl;
return 0;
}
```

Friend in operator overloading

```
class Distance
{
private:
int feet;
float inches;
public:
Distance() : feet(0), inches(0.0)
{ }
Distance(float fltfeet)
{
feet = int(fltfeet);
inches = 12*(fltfeet-feet);
}
Distance(int ft, float in)
{ feet = ft; inches = in; }
void showdist()
{ cout << "\'-" << inches << '\"'; }
friend Distance operator + (Distance, Distance); //friend
};

Distance operator + (Distance d1, Distance d2){
int f = d1.feet + d2.feet;
float i = d1.inches + d2.inches;
if(i >= 12.0)
{ i -= 12.0; f++; }
return Distance(f,i);
}
```

```
int main()
{
Distance d1 = 2.5;
Distance d2 = 1.25;
Distance d3;
cout << "\nd1 = "; d1.showdist();
cout << "\nd2 = "; d2.showdist();
d3 = d1 + 10.0;   // OK
cout << "\nd3 = "; d3.showdist();
// d3 = 10.0 + d1;   //OK
 cout << "\nd3 = "; d3.showdist();
cout << endl;
return 0;
}
```

friends for Functional Notation

```
class Distance
{
private:
int feet;
float inches;
public:
Distance() : feet(0), inches(0.0)
{ }
Distance(int ft, float in) : feet(ft), inches(in)
{ }
void showdist()
{ cout << feet << "\'-" << inches << '\"'; }
float square();
};

float Distance::square()
{
float fltfeet = feet + inches/12;
float feetsqrd = fltfeet * fltfeet;
return feetsqrd;
}
```

```
int main()
{
Distance dist(3, 6.0);
float sqft;
sqft = dist.square();
cout << "\nDistance = ";
dist.showdist();
cout << "\nSquare = " << sqft ;
return 0;
}
```

friends for Functional Notation

```
class Distance
{
private:
int feet;
float inches;
public:
Distance() : feet(0), inches(0.0)
{ }
Distance(int ft, float in) : feet(ft), inches(in)
{ }
void showdist()
{ cout << "\'-" << inches << '\"'; }
friend float square(Distance);   //friend
};

float square(Distance d)
{
float fltfeet = d.feet + d.inches/12;
float feetsqrd = fltfeet * fltfeet;
return feetsqrd;
}
```

```
int main()
{
Distance dist(3, 6.0);
float sqft;
sqft = square(dist);
cout << "\nDistance = ";
dist.showdist();
cout << "\nSquare = " << sqft ;
return 0;
}
```

- The member functions of a class can all be made friends at the same time when you make the entire class a friend.

Example: Friend class

```
Class beta;
class alpha
{
private:
int data1;
public:
alpha() : data1(99) { }
friend class beta;
};
class beta
{
public:
void func1(alpha a) { cout << “\ndata1=” << a.data1; }
void func2(alpha a) { cout << “\ndata1=” << a.data1; }
};
int main()
{
alpha a;
beta b;
b.func1(a);
b.func2(a);
cout << endl;
return 0;
}
```

Question

```
class Base
{
public:
    virtual void show() { cout<<" In Base n"; }
};

class Derived: public Base
{
public:
    void show() { cout<<"In Derived n"; }
};

int main(void)
{
    Base *bp = new Derived;
    bp->show();
}
```

Question

```
class Base
{
public:
   virtual void show() { cout<<" In Base n"; }
};

class Derived: public Base
{
public:
   void show() { cout<<"In Derived n"; }
};

int main(void)
{
   Base *bp, b;
   Derived d;
   bp = &d;
   bp->show();
   bp = &b;
   bp->show();
   return 0;
}
```

Question

```
class Base
{
public:
    virtual void show() = 0;
};

int main(void)
{
    Base b;
    Base *bp;
    return 0;
}
```

Question

```
class Base
{
public:
   virtual void show() = 0;
};

class Derived : public Base { };

int main(void)
{
   Derived q;
   return 0;
}
```

Question

```
class Base  {
public:
   Base()    { cout<<"Constructor: Base"<<endl; }
   virtual ~Base()   { cout<<"Destructor : Base"<<endl; }
};
class Derived: public Base {
public:
   Derived()   { cout<<"Constructor: Derived"<<endl; }
   ~Derived()  { cout<<"Destructor : Derived"<<endl; }
};
int main()  {
   Base *Var = new Derived();
   delete Var;
   return 0;
}
```

Question

```
class A
{
public:
    virtual void fun() { cout << "A::fun() "; }
};

class B: public A
{
public:
   void fun() { cout << "B::fun() "; }
};

class C: public B
{
public:
   void fun() { cout << "C::fun() "; }
};

int main()
{
    B *bp = new C;
    bp->fun();
    return 0;
}
```

Question

```
class Base
{
public:
   virtual void show() { cout<<" In Base n"; }
};

class Derived: public Base
{
public:
   void show() { cout<<"In Derived n"; }
};

int main(void)
{
   Base *bp = new Derived;
   bp->Base::show();
   return 0;
}
```

this pointer

- Every object in C++ has access to its own address through an important pointer called **this** pointer.
- The **this** pointer is an implicit parameter to all member functions.
- Therefore, inside a member function, **this** may be used to refer to the invoking object.

Example

```
class where
{
private:
char charray[10];
public:
void reveal()
{ cout << “\nMy object’s address is “ << this; }
};

int main()
{
where w1, w2, w3;
w1.reveal();
w2.reveal();
w3.reveal();
cout << endl;
return 0;
}
```

Accessing member data with this

```
class what
{
private:
int alpha;
public:
void tester()
{
this->alpha = 11;
cout << this->alpha;
}
};

int main()
{
what w;
w.tester();
cout << endl;
return 0;
}
```

Use of this pointer

- When local variable's name is same as member's name

```
class Test
{
private:
  int x;
public:
  void setX (int x)
  {
     this->x = x;
  }
  void print() { cout << "x = " << x << endl; }
};

int main()
{
  Test obj;
  int x = 20;
  obj.setX(x);
  obj.print();
  return 0;
}
```

Use of this pointer

- To return reference to the calling object

```
class Test
{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  Test& setX(int a) { x = a; return *this; }
  Test& setY(int b) { y = b; return *this; }
  void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
  Test obj1(5, 5);
  obj1.setX(10).setY(20);
  obj1.print();
  return 0;
}
```

Overloading the Assignment Operator

```
class alpha
{
private:
int data;
public:
alpha()
{ }
alpha(int d)
{ data = d; }
void display()
{ cout << data; }
void operator = (const alpha& a)
{
    data = a.data;
    cout << “\nAssignment operator invoked”;
}
};
```

```
int main()
{
alpha a1(37);
alpha a2;
a2 = a1;        // invoke overloaded =
cout << “\na2=”;
a2.display();
alpha a3 = a2;  //does NOT invoke =
cout << “\na3=”;
a3.display();
}
```

Overloading the Assignment Operator

```
class alpha
{
private:
int data;
public:
alpha()
{ }
alpha(int d)
{ data = d; }
void display()
{ cout << data; }
alpha  operator = (const alpha& a)
{
    data = a.data;
    cout << "\nAssignment operator invoked";
    return alpha(data);
}
};
```

```
int main()
{
alpha a1(37);
alpha a2,a3;
a3 = a2=a1;       // invoke overloaded =
cout << "\na2=";
a2.display();
cout << "\na3=";
a3.display();
}
```

Overloading the Assignment Operator

```
class alpha
{
private:
int data;
public:
alpha()
{ }
alpha(int d)
{ data = d; }
void display()
{ cout << data; }
alpha&  operator = (const alpha& a)
{
    data = a.data;
    cout << “\nAssignment operator invoked”;
    return *this;
}
};
```

```
int main()
{
alpha a1(37);
alpha a2,a3;
(a3 = a2)=a1;       // invoke overloaded =
cout << “\na2=”;
a2.display();
cout << “\na3=”;
a3.display();
}
```

Overloaded Assignment Operator for different objects

```
class beta
{
int x;
public:
beta(int a=0):x(a){}
friend class alpha;
};
class alpha
{
int data;
public:
alpha()   { }
alpha(int d)
{ data = d; }
void display()
{ cout << data; }
alpha&  operator = (const beta& b)
{
       data = b.x;
       cout << “\nAssignment operator invoked”;
       return *this;
}
};
```

```
int main()
{
alpha a1(10);
beta b1(20);
a1=b1;
a1.display();
}
```

The Copy Constructor

- A copy constructor is a member function which initializes an object using another object of the same class.
- A copy constructor has the following general function prototype:

  ClassName (const ClassName &old_obj);

- Copy initialization:

  alpha a3(a2); // copy initialization

  alpha a3 = a2; // copy initialization, alternate syntax

Example: copy constructor

```
class alpha
{
private:
int data;
public:
alpha()
{ }
alpha(int d)
{ data = d; }
alpha(const alpha& a)   //copy constructor
{
data = a.data;
cout << “\nCopy constructor invoked”;
}
void display()
{ cout << data; }
void operator = (alpha& a)  //overloaded = operator
{
data = a.data;
cout << “\nAssignment operator invoked”;
}
};
```

```
int main()
{
alpha a1(37);
alpha a2;
a2 = a1; //invoke overloaded =
cout << “\na2=”; a2.display();
alpha a3(a1); //invoke copy constructor
// alpha a3 = a1; //equivalent definition of a3
cout << “\na3=”; a3.display();
}
```

Why to define our own assignment operator

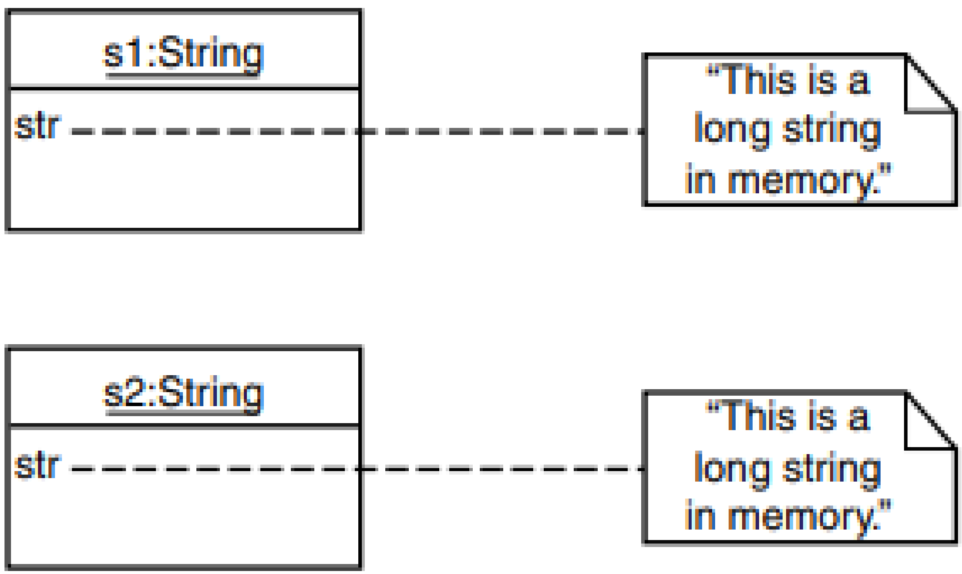
```
class Test
{
   int *ptr;
public:
   Test (int i = 0)      { ptr = new int(i); }
   void setValue (int i) { *ptr = i; }
   void print()          { cout << *ptr << endl; }
};

int main()
{
   Test t1(5);
   Test t2;
   t2 = t1;
   t1.setValue(10);
   t2.print();
   return 0;
}
```

Why to define our own assignment operator

```
class Test
{
	int *ptr;
public:
	Test (int i = 0)		{ ptr = new int(i); }
	void setValue (int i) { *ptr = i; }
	void print()				{ cout << *ptr << endl; }
	Test & operator = (const Test &t);
};
Test & Test::operator = (const Test &t)
{
	*ptr = *(t.ptr);
	return *this;
}
int main()
{
	Test t1(5);
	Test t2;
	t2 = t1;
	t1.setValue(10);
	t2.print();
	return 0;
}
```

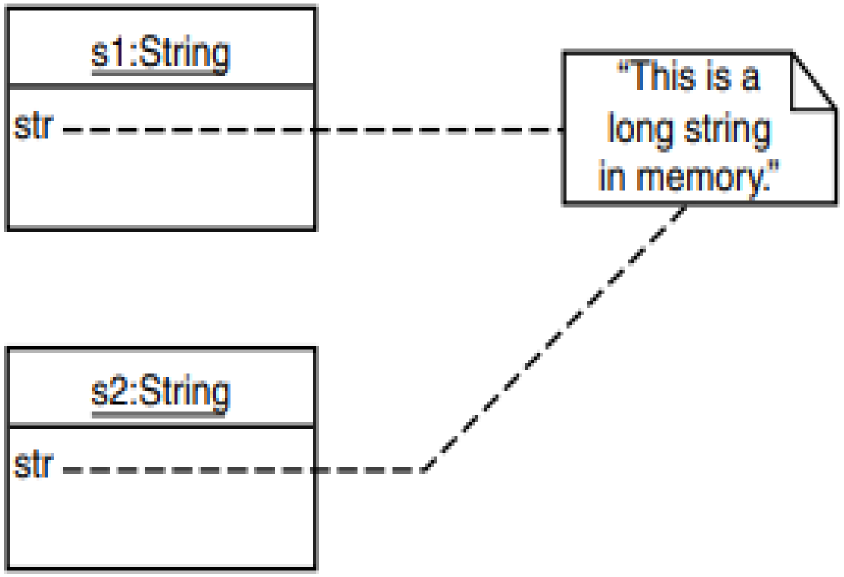
Use of overloaded assignment operator and copy constructor

- If we assign one String object to another (from s1 into s2 in the previous statement), we simply copy the string from the source into the destination object.
- This is not very efficient, especially if the strings are long.

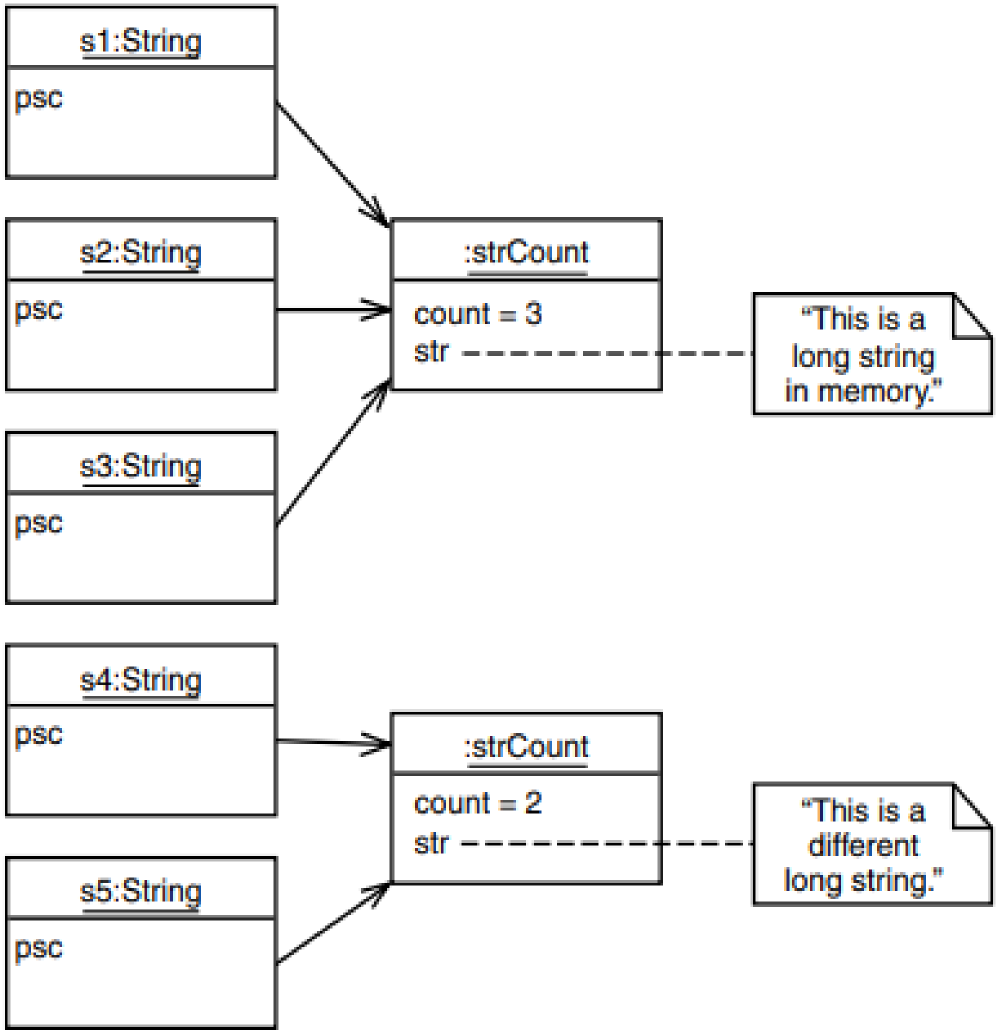


Use of overloaded assignment operator and copy constructor

- Instead of having each String object contain its own char* string, we could arrange for it to contain only a pointer to a string.



Use of overloaded assignment operator and copy constructor



Use of overloaded assignment operator and copy constructor

```
class strCount
}
int count;
char* str;
friend class String;
public:
strCount(char* s)
{
  int length = strlen(s);
  str = new char[length+1];
  strcpy(str, s);
  count=1;
}
~strCount()
{ delete[] str; }
};

class String
{
strCount* psc;
public:
String()
{ psc = new strCount(“NULL”);
 }
String(char* s)
{ psc = new strCount(s); }
```

```
String(String& S)
{
  psc = S.psc;
  (psc->count)++;
}
~String()
{
  if(psc->count==1)
    delete psc;
  else
    (psc->count)--;
}
void display()
{   cout << psc->str;    }
void operator = (String& S)
{
  if(psc->count==1)
    delete psc;
  else
    (psc->count)--;
  psc = S.psc;
  (psc->count)++;
}
};
```

```
int main()
{
String s3 = “When the fox
preaches, look to your geese.”;
cout << “\ns3=”; s3.display();
String s1;
s1 = s3;
cout << “\ns1=”; s1.display();
String s2(s3);
cout << “\ns2=”; s2.display();
}
```