Brown University ENGN2912B Fall 2017

Scientific Computing in C++

Lecture 8 | More about Classes

The copy constructor

Consider the following program fragment, that uses the class **`point`** from the last lecture.

```
point p1(1.0, 2.0);
void f(point p){…};
f(p1);//this is the copy constructor being called
```

It is very natural that a programmer will want to assign function arguments as **`point`** values, and pass the **`point`** variable by value. The question is what happens in this case? The compiler automatically creates a constructor for the class **`point`** called the *copy* constructor. The signature of the copy constructor is

```
point(const point& p);// or equivalently
point(point const& p);
```

When the compiler implements this constructor it does it in the crudest way possible. It just copies the memory layout of the class instance byte for byte. If the programmer doesn't want this exact copy to happen, then they must declare and implement the copy constructor in the class. What circumstances might arise where the programmer wouldn't want the default copy constructor called? Consider the following class specification.

```
class counter {
 public:
  counter();
  counter(counter const& c);
  void upcount();
  void downcount();
  void clear();
  unsigned count() const;
  void set_value(double value);
  double value() const;//how can this method be const?
 private:
  double value_;
  unsigned count_;
}
```

This class is intended to store a value and also maintain a count of the times that value is accessed. The implementation of the accessor to **`value_`** might be as follows.

```
double counter::value()const {
  counter* self = const_cast<counter*>(this*);
  self->count_++;
  return value_;
}
```

The programmer will want to copy **value_** but not **count_**, because that is particular to a given class instance. Also, the value of **count_**should be initialized to zero. The implementation of the two constructors is as follows.

```
counter::counter() {
  value_ = 0;
  count_ = 0;
}
//the copy constructor
counter::counter(counter const& c) {
  value_ = c.value();//value() must be const
  count_ = 0;
}
```

Note that is important that the method **`counter::value()`** be declared as **`const`** otherwise the copy constructor would have to cast away **`const`**.

The assignment operator =

A related method to the copy constructor is the assignment operator **=**. This operator by default, causes the memory of the class instances on each side of the **=** sign to be identical to the right side. As a simple example, consider the assignment, **`double x =  3.0;`** The compliler first allocates memory for **`x`** with an undefined value and then assigns the value to be `3.0`. Note that this behavior is not the same as the default copy constructor which creates an instance of the class. Instead, the assignment operator applies to two *existing* instances on the left and right side of **=**. The programmer may want to customize the **=** operator for similar reasons to that just described for the copy constructor.

Casting, subclasses, and virtual methods

Since a class is just a data type it is possible to cast classes to subclasses and vice versa. In order to really understand what is going on when classes are cast, it is necessary to understand how a class and its subclasses are laid out in memory.
Consider the **`curve`** example from the last lecture.

```
#ifndef curve_h_
#define curve_h_
#include "vector.h"
#include "point.h"
class curve {
public:
       curve(){};
       ~curve(){};
      curve(point const& p0, point const& p1);
      double length() const;
      virtual vector tangent(point const& p)const = 0;
       virtual char const* name(){return "curve";}
      point p0()const;
      point p1()const;
protected://described later
      point endpoint0_;
      point endpoint1_;
};
#endif  // curve_
```

Suppose the subclass **circle**class is defined as,

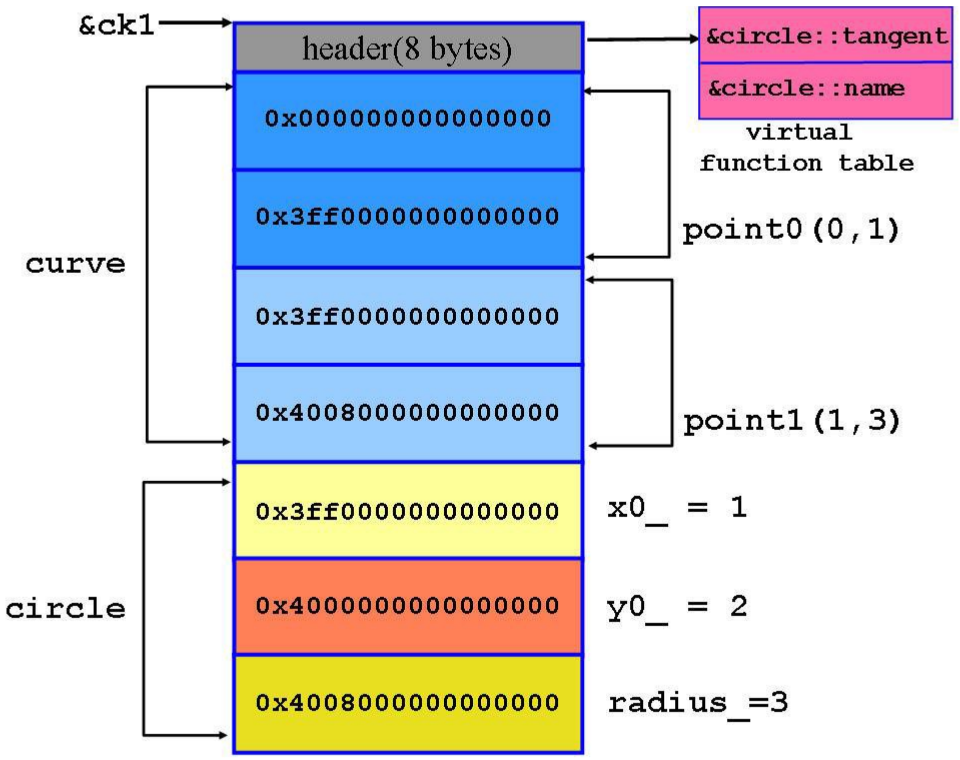
```
#ifndef circle_h_
#define circle_h_

#include "curve.h"
#include "point.h"
#include "vector.h"
class circle : public curve {
 public:
  circle();
  circle(point const& point0, point const& point1, double x0,
         double y0, double radius);
  ~circle();
  virtual vector tangent(point const& p)const;
  virtual char const* name(){return "circle";}
  vector tangent(double ang)const;
private:
  double x0_;
  double y0_;
  double radius_;
};
```

Consider the following construction of an instance of `circle`.

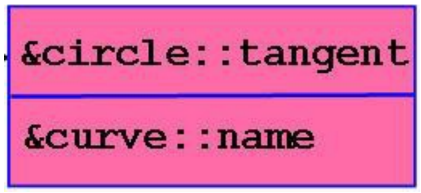
```
point p1(0.0, 1.0), p2(1.0, 3.0);
circle ck1(p1, p2, 1, 2, 3);
```

The layout of circle, `ck1`, in memory is shown below.



The header contains pointers to access the virtual functions. Then the data of class is stored. The endpoints of the curve are stored first, followed by the additional member values added by the **`circle`** class. Note that the member data values are stored in the order they are declared in the class specification.

The execution of virtual functions is handled by an indirect address table called the *virtual function table*, or *vtable*, shown in the upper right of the figure. The vtable contains addresses of the location of the virtual functions implemented for the class. In this case, **`circle`** implements both virtual functions, **`tangent`** and **`name`**, thus the table contains pointers to the implementations defined by **`circle`**. If **`circle`** did not implement the **`name`** method, then the vtable would look as below.



In this case, the value returned by calling **`name`** on **`circle`** would be **`"curve"`**.
The order of the virtual function addresses in the table are the same, regardless of which implementation is implemented.

Consider the following code fragment operating with the initial implementation of **`circle`**.

```
circle ck1(p1, p2, 1, 2, 3);
circle* ck1p = & ck1;
//note, could use dynamic_cast, which will return 0 if cast fails
//this topic is discussed below
curve* cvp = static_cast<curve*>(ck1p);
char const* n = cvp->name();//n == "circle"
```

The **`circle`** pointer is cast to a **`curve`** pointer. However the address is the same in both cases. That is,

```
ck1p == 0x00d9fd70
cvp  == 0x00d9fd70
```

The cast doesn't change the memory layout of the class, so the virtual function table pointer and data values in memory are unchanged. This arrangement explains how the **`tangent`** and **`name`** methods are resolved correctly in support of polymorphism. Even though **`cvp`** is a **`curve`** pointer, the **`name`** method is called on the **`circle`** class.

What about the **`tangent(double ang)`** method that is specific to **`circle`**?

The following statement will generate the indicated compiler error.

```
vector v = cvp->tangent(3.14159/4);
```

```
error C2664: 'curve::tangent' : cannot convert parameter 1 from
'double' to 'const point &'
```

On the other hand, there is no problem with the statement,

```
vector v = ck1p->tangent(3.14159/4);
```

The cast of the **`circle`** pointer to the **`curve`** pointer masks any functions or public member variables that exist solely on **`circle`**. The interface is only that declared for **`curve`**.

What if one tries to dereference the curve pointer? Consider the following code fragment.

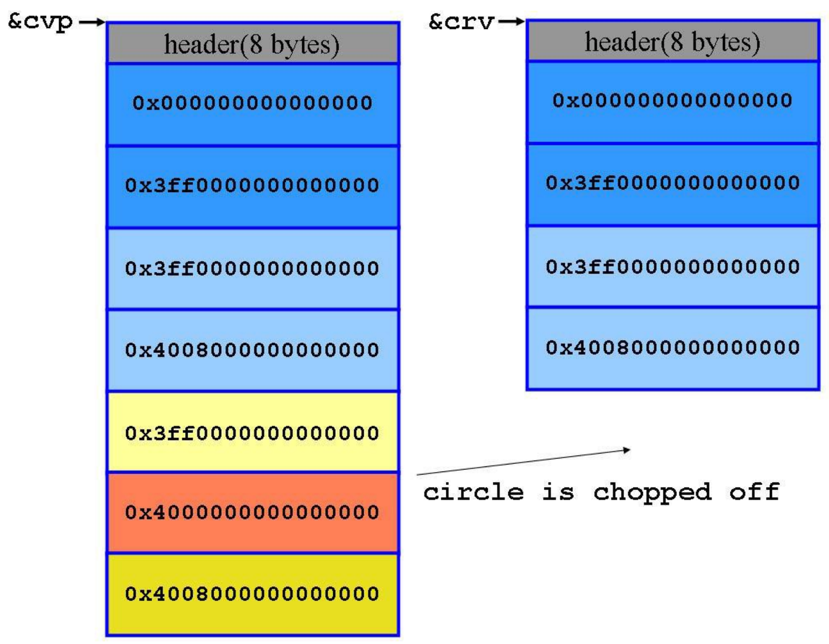
```
curve crv = *cvp;
```

A compiler error results,

```
Error 1     error C2259: 'curve' : cannot instantiate abstract
class
```

Since curve an abstract base class, it can never be instantiated as itself, only as a subclass.

Suppose for the moment that **`curve`** was not abstract. The cast above will "chop" off the **`circle`** part of memory layout and copy the top **`curve`** part into a new memory layout that has only the size of **`curve`**. The new virtual table will hold pointers to **`curve`** methods. This chopping action is shown in the figure below.



Thus, casting and dereferencing should be done with care, since subclass data will be unrecoverable after the cast. In spite of the dangers of casting, the benefits of polymorphism are well worth the risk. As an illustration of polymorphism consider the following code.

```
unsigned n ;
curve** curve_array = new curve*[n]

//create a heterogeneous array of curves
curve_array[0]= static_cast<curve*>( new circle(...) );
curve_array[1]= static_cast<curve*>( new line(...) );
curve_array[2]= static_cast<curve*>( new poly_line(...) );
...
//name is specialized to each actual type of curve
for(unsigned i=0; i<n; ++i)
  cout << (curve_array +i)->name() << endl;
```

The screen output of the program will be,

```
circle
line
poly_line
```
…

The **`name`** method is automatically specialized to each subclass instance of **`curve`** in the array.

Downcasting

So far, the main issues associated with casting a class "up" to the base class have been described. It is often necessary to cast a class back "down" to the subclass in order to access methods or data members specific to the subclass. Suppose there is a function that returns a pointer to curve,

```
curve* f();
```

The user of this function has no idea what subclass of **curve** might be returned by **f**. Without downcasting to the subclass only the methods of **curve** are accessible. The first approach might be to guess. The user might just assume all the curves are circles and apply the following cast.

```
fclass fc;
circle* c = static_cast<circle*>( fc.f() );
```

In this example **fclass** is the class that implements the method, **f()**. This cast is foolhardy since the **curve*** returned by **f** might be a pointer to a **line**. The cast will produce unpredictable and usually disastrous results by accessing irrelevant memory locations.

The safe approach is to use the virtual **name** method to find out which subclass the **curve** pointer is referring to. This test might look like the following.

```
curve* cvp = fc.f();
if( string_equal(cvp->name(), "circle") ){
circle* c = static_cast<circle*>(cvp);
// do something appropriate for circle
…
}
if( string_equal(cvp->name(), "line") ){
line* l = static_cast<line*>(cvp);
// do something appropriate for line
…
}
…
```

The function **string_equal** has been implemented somewhere else and is designed to test if two strings are equal. Since the virtual **name** method returns the string defined by the curve class or its subclasses, this approach will downcast the curve pointer appropriately.

The C++ language itself supports a mechanism for testing the type of a pointer, called runtime type identification (RTTI). The following program illustrates the use of RTTI, with the **dynamic_cast** function. In the example above, the test for name can be replaced by the cast as follows. Note that the header **typeinfo** must be included.

```
#include <typeinfo>
curve* cvp = fc.f();
if( circle* c = dynamic_cast<circle*>(cvp) ){
// do something appropriate for circle
…
}
if( line* l = dynamic_cast<line*>(cvp)){
// do something appropriate for line
…
}
...
```
The templated **`dynamic_cast`** function returns **`0`** if the cast is not successful and an appropriate subclass pointer if the cast is successful, i.e., the **`curve`** pointer is addressing the subclass pointer of the specified type. Note that this situation is an ideal application for declaring a local variable in the condition of the **`if`** statement. This approach avoids calling **`dynamic_cast`** again within the body of the **`if`**.

Note that **`dynamic_cast`** requires at least one function of the class to be declared **`virtual`**. If such polymorphism is not needed for a class hierarchy then the class destructor, **`~curve()`**, can be made **`virtual`** as a default to be able to use dynamic casting.
There are many difficult situations that dynamic casting has to cope with, such as templated classes, and it is only recently that compiler vendors have produced reliable and complete implementations of RTTI. C++ code written earlier, typically uses the **`name`** method approach for safe downcasting. Essentially this earlier technique is building run time type information into the class itself.

Static class methods and values

In some cases, it is useful to have methods or member values that apply to all the instances of a class. Such methods or member values are denoted by the **`static`** keyword. Note that this use is the third meaning of **`static`** that has been encountered so far in the C++ language.

The implementation of a static member and static accessor to the member is illustrated by the following class implementation of **`pointc`**. The contents of `pointc.h` is as follows. A good example of a static member is **`count_`**, the number of instances of the class that have been created during the life of the application. With this class specification, the value of **`count_`** will be the same for all class instances and so there is no need to create an instance of the class to dispatch the **`count()`** method.

```
#ifndef pointc_h_
#define pointc_h_

class pointc {
 public:
  pointc();
  pointc(double x, double y);
  ~pointc();
  double x() const;
  double y() const;
  static unsigned count();//static accessor of count_
 private:
  double x_;
  double y_;
  static unsigned count_;//static member
};
```

```
#endif  // pointc_h_
```

Static methods can be accessed through the class namespace operator **`pointc::`** . For example the following declaration accesses the **`count_`** value.

```
unsigned c = pointc::count();
```

No class instance is being constructed. A question arises that if there is no need to construct a class instance to access **`count_`** , how does its value ever get initialized in the first place? Keep in mind that the class specification in the .h file does not provide an initial value to **`count_`**.

The answer is that **`count_`** must be initialized outside of class scope since it doesn't depend on any particular instance of the class. The initialization must exist before any manipulation or access to **`count_`** .This initialization as well as the manipulation of **`count_`** in the constructors and destructor of **`pointc`** is shown below from the .cxx file.

```
unsigned pointc::count_ = 0;//initialize count_

pointc::pointc(){
  x_ = 0;
  y_ = 0;
  count_++;// increment count on construction
}

pointc::pointc(double x, double y) {
  x_ = x;
  y_ = y;
  count_++;
}
pointc::~pointc() {
  count_--; // decrement count on destruction
}
```

An example of using the static method **`pointc::count()`** in a program is shown below.

```
#include "pointc.h"
int main() {
  unsigned n = pointc::count();// n == 0
   {
    pointc p1(0,4), p2(1,1);
    n = pointc::count();// n == 2
  }
  n = pointc::count();// n == 0
}
```

In this example the initial value of **`count_`** is zero. Then two instances of **`pointc`** are constructed so **`count_ == 2`**. Then these instances leave the scope brackets and the **`count_`** returns to zero.

Some rules about static methods:

- A static method cannot access class members, including **`this`**. If such access were allowed the result would differ between class instances, a violation of the concept of static access.
- There is no need to implement a constructor or destructor for a class which has all static methods. There will never be a need to dispatch from a class instance. The default constructor should be designated **`private`** to prevent attempts to construct the class.

C++ does allow the definition of static integer constants in the class specification. For example,

```
class newton_solver
{
    static const unsigned max_iterations = 10000;
  …
}
```

This static variable is accessible through the static class namespace operator, as with other static variables, e.g.,

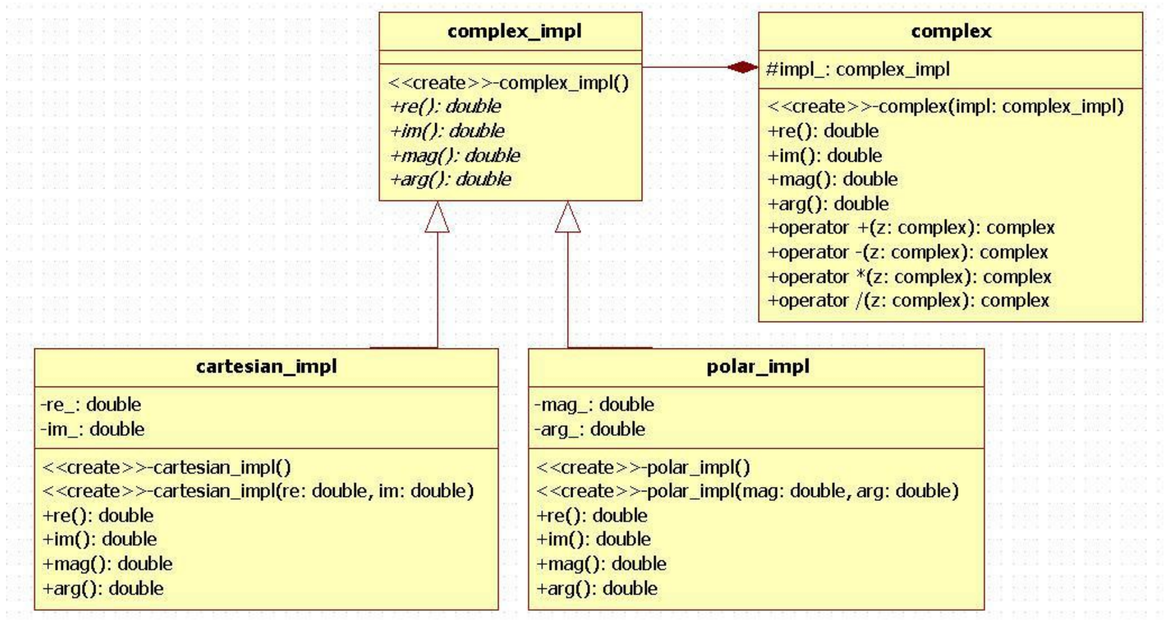
```
unsigned n = newton_solver::max_iterations;
```

Since only integer values are allowed, if a class defines many such constants it probably makes more sense to define them in an **`enum`** statement.

Class member reference data types and the "implementation" strategy

Consider the following example illustrated as a UML diagram. The class constructor is denoted by

```
 <<create>>-class_name(arguments).
```



This example illustrates how to define a complex number class that can hold data in two different formats.

The first format is the Cartesian form of a complex number $z = x + iy$, where x and y are real numbers. The second format is the polar form $z = \rho e^{i\alpha}$, where ρ>0 is the magnitude, and α is the argument.

From a mathematical point of view, these formats are different realizations of the complex number field, i.e., the realization provides a set of elements that satisfy all the axioms of complex numbers.

Suppose it is the case that complex numbers can be realized in either way, and the programmer wants to handle both variants with a single **`complex`** interface. The UML design shows that the complex class can have a member which is **`complex_impl`** (some implementation of a **`complex`** type). The realizations of two types of complex numbers are shown in the figure. They both are subclasses of **`complex_impl`** and thus can be represented in the **`complex`** class by a class member that is a pointer to the abstract base class of all complex realizations, i.e. **`complex_impl*.`** In the following section the **`complex_impl*`** member will be referred to as the *resource* of **`complex`** since it's the user's responsibility to manage it (e.g. allocate, delete and pass around).

The corresponding source code is as follows. First, the definitions of the classes.

```
class complex_impl
{
public:
  complex_impl(){}//constructor does nothing
  virtual double re() const = 0; //pure virtual methods
  virtual double im()const = 0;
  virtual double mag() const = 0;
  virtual double arg()const = 0;
  virtual complex_impl * clone() = 0;

};

class cartesian_impl : public complex_impl
{
public:
  cartesian_impl():re_(0), im_(0){}//empty body
  cartesian_impl(double re, double im): re_(re), im_(im){}
  virtual double re() const;
  virtual double im()const;
  virtual double mag() const;
  virtual double arg()const;
  virtual complex_impl * clone();
private:
  double re_;
  double im_;
};
```

```
class polar_impl : public complex_impl
{
public:
  polar_impl():mag_(0), arg_(0){}
  polar_impl(double mag, double arg):mag_(mag), arg_(arg){}
  virtual double re() const;
  virtual double im()const;
  virtual double mag() const;
  virtual double arg()const;
  virtual complex_impl * clone();
private:
  double mag_;
  double arg_;
};
```

The two subclasses implement the virtual methods of `complex_impl` in different ways but the virtual interface is identical and enforced by the pure virtual function definitions in `complex_impl`.

The `complex` class presents the same interface, regardless of the implementation. There could be other complex implementations added at a later date. For example, a complex number is also isomorphic to a 2-dimensional vector. With the design presented here the a new implementation can be added without the slightest effect on the implementation of the `complex` class or its use in other code.

Notice that the second and copy constructor of the `complex` class needs to instantiate the `impl_` resource somehow.
A statement like

```
this->impl_ = new complex_impl(…);
```

will fail to compile due to the fact that `complex_impl` is an abstract class which cannot be instantiated. To overcome this issue, one has to declare a `clone()` method on the base class `complex_impl` which will be overridden by the two subclasses `cartesian_impl` and `polar_impl.` The clone method is often referred to as a *virtual constructor*.

```
class complex {
  public:
      // default constructor needs some impl as a place holder
      complex():impl_(ci){}


 //constructor that takes a valid pointer to a complex_impl
      complex(complex_impl const * impl){
       //return a new instance of complex_impl by cloning impl
            this->impl_ = impl->clone();
      }
// copy constructor creates and takes the ownership of a new
instance of complex_impl  by cloning the resource of c.
      complex(complex const & c){
      // your code here
      }
      ~complex(){delete impl_;}
      double re()  const {return impl_->re();}
      double im()  const {return impl_->im();}
      double mag() const {return impl_->mag();}
      double arg() const {return impl_->arg();}
      complex operator + (complex const& c);
      complex operator - (complex const& c);
      complex operator * (complex const& c);
      complex operator \ (complex const& c);
      complex& operator = (complex const& c);
      complex_impl* get_impl()const {return impl_;}
   protected:
     complex_impl * impl_;
};
```

The following fragment of code shows how this design is used.

```
cartesian_impl ctimpl(-1.0, 0);
polar_impl plimpl(1.0, 3.1415926535897932);
complex cpx1(&ctimpl), cpx2(&plimpl);
double re1 = cpx1.re(), re2 = cpx2.re();
// re1 == -1.0,  re2 == -1.0
```

Note that instances of **`complex`** can be constructed from either implementation, which is automatically cast to the base reference type **`complex_impl*`**.

Inside the implementations, the approach to getting the required virtual methods can be quite different. For example, the **`cartesian_impl`** code for **`mag()`** is,

```
double cartesian_impl::mag() const
{return sqrt(re_*re_+im_*im_);}
```

while the implementation of **`re()`** in **`polar_impl`** is,

```
double polar_impl::re() const
{return mag_*cos(arg_);}
```

Note that the result of an operator on the **`complex`** class must result in an instance that has a valid pointer to a **`complex_impl`**, so that the operator output can be constructed properly. In this regard, it will be necessary to implement the = operator .

In designing this operator, one should keep in mind that both the right hand side (rhs) and the left hand side(lhs) need to have the **`complex_impl*`** resource at different memory addresses. Thus a copy of the resources from the rhs to the lhs need be performed.

Note that the = operator should return a pointer to **`this`** and the return type should be **`complex&`** and not **`complex`**. The latter will return a **value** which will force the calling of the copy constructor. This is unwanted since the copy constructor of the **`complex`** class should create a new instance on the heap of **`complex_impl`** which will remain as orphaned memory.

However, this implementation is slow since it forces a copy of what is on the rhs of the = operator. A faster alternative would be to just look at the resource on the rhs without explicitly copying it.

Doing so, no copy is performed, rendering the code faster. However, the resources of both the rhs and the lhs will have the same memory address which is unwanted more often than not. The only situation where this implementation would be appropriate is when the user doesn't care about what happens to the rhs. For example, consider the fragment:

```
cartesian_impl ctimpl(-1.0, 0);
polar_impl plimpl(1.0, 3.1415926535897932);
complex A(&ctimpl), B(&plimpl),C,D;

C = A + B;
D = C;
```

During runtime, the program will create an internal temporary object of type **`complex`**, say **`tmp_sum`** to store the value of **`A + B`**. This object will dissapear from memory right after the assignment **`C = tmp_sum`**. In this case the second implementation of the = operator is desireable since one doesn't care about the resource of **`tmp_sum`** and the object C can just take its ownership.

In the second assignment, the first implementation of the equal operator needs to be called since we want both **`D`** and **`C`** to have their **`complex_impl`** resource at different valid addresses.

To reconcile this issue, the 2011 revision of C++ , also known as C++11, introduced "*rvalue references*". Loosely speaking, rvalues are variables that can only exist on the right hand side of the equal operator while lvalues are variables that can exist on both left and right sides of the = operator. Specifically , A+B is a rvalue. A statement like

`A + B = C`; will result in a compiler error.

```
error C2106: '=' : left operand must be l-value
```

Finally, the **`complex`** class will have two implementations of the = operator. The first will take a const reference to the class **`complex`** while the second will take a reference to a rvalue, indicated to the compiler by the newly introduced syntax "&&".

```
complex& complex::operator = (complex const& c); //first
declaration taking a const reference
complex& complex::operator = (complex && c); //second declaration
taking a rvalue reference
```

The C++ compiler will automatically call the appropriate = operator based on whether `c` is a temporary rvalue or a persistent, modifiable lvalue. With these changes to the complex class