Writing C++ Classes

Format of a class declaration

class X : public class Y, class Z {
   // member declarations
}

All members must be declared in the class definition
Members may be defined elsewhere
Members can be defined in the class definition itself

All pointers and references have the same size but class objects have varying sizes
You cannot allocate space for an object until it has been defined (and its size known)
You can declare pointers to objects before its size is known
Special Members

Default constructor
  Used to initialize variables when no initializer is given
  Used to initialize array elements

Default destructor
  Does nothing before deallocating the object
  Generally all you need

Default copy constructor
  Called when a copy of an object is needed

Default assignment operator
  Used for assignment
  Will use copy constructor if not defined
Storage Allocation

C++ does not have a garbage collector
If you allocate storage you must release it or there will be a "space leak"

Kinds of storage at runtime
  program storage
  static or global storage
  local or stack storage
  heap or dynamic storage

You can also use malloc and free from the C library but this is not recommended
Program Storage

Actual compiled code

Library code

Sometimes literal strings go here

This memory is generally set "read only" by the operating system

You will get a fault if you try to modify it
Static or Global Storage

Used for
  global variables
  static variables

Space allocated when program execution starts
  Default constructor will be called for all uninitialized variables with a constructor

Released when program terminates
  Destructors will be called for all variables with a destructor
Local or Stack Storage

Used for
- Function or method parameters
- Local variables
- Temporary storage
- Objects created with constructor call (without "new")

Allocated when function or method or block is entered
- Default constructor will be called for all uninitialized variables with a constructor

Released when function or method or block is exited
- Destructors will be called for all variables with a destructor
Heap or Dynamic Storage

Used for all storage explicitly allocated by the program with "new constructor"

Memory must be explicitly deallocated with "delete"

Arrays

Arrays are allocated with "new Type[ length ]"

  The element Type must have a default constructor to initialize the elements of the array

  Arrays must be deallocated with "delete[] var"

The value of a new expression is a pointer to the newly allocated storage

You may call delete with a null pointer - this is not an error and has no effect
Pointer Variables

Pointer variables contain a pointer

```cpp
int *ii;
```

The type of a pointer is "pointer to type"

Pointer types are not classes

- They have no constructors
- They have no destructors
- They have no methods

You must dereference a pointer before accessing fields of a class

```cpp
Point *pp = new Point(3, 4);
cout << (*pp).getX();
```

The arrow operator "->" can be used to dereference and access a member at the same time

```cpp
Point *pp = new Point(3, 4);
cout << pp->getX();
```
Reference Variables

Reference variables are like pointer variables except they are automatically dereferenced when used.
It is impossible to access the "hidden" location of the reference itself.

```c++
Point *pp = new Point(3, 4);
Point &qq = *pp;
cout << qq.getX();
```

You can think about reference variables as a synonym of another variable.
This other variable may not have a name.

```c++
int a[] = {1, 2, 3};
int &a2 = a[2];
a2 = 4;
```
Polymorphism

In C++, polymorphism only works for pointer or reference variables

And then only for functions declared virtual

But pointer variables are tricky to use

Reference variables cannot be assigned to

Assignment to a reference variable assigns to what the pointer variable refers to or is a synonym for
Common Storage Problems - Containers

Containers of pointers

{
    vector< Card * > hand;
    hand.push_back( new Card( "Ace" ) );
    hand.push_back( new Card( "Deuce" ) );
    // more stuff

}  

Should vector's destructor delete the card objects?

    STL containers do not delete objects referred to by pointers
Common Storage Problems - Returning pointers to computed objects

A complex number class

class Complex {

Complex *operator+( const Complex &z ) {
    return new Complex( this->re + z.re,
                        this->im + z.im );
}

} // Complex

Now try

Complex x( 3, 4);
Complex y( 5, 6);
Complex z = *( x + y );

Who deletes the object returned by operator+ ?
Dangling pointers

What does this do?

```c
int * f() {
    int i = 3;
    return &i;
}

int *jj = f();

cout << *jj;
```

The memory for i is deallocated when f returns but we still have a pointer to this memory that may contain anything.

Dangling pointers can also result if you delete something that you still have a pointer to somewhere else.

```c
int *ii = new int(3);
delete ii;

cout << *ii;
```
Example - class Complex

Class Complex will have "value semantics"
This means that objects of type Complex cannot change their value once they are created
This is different from objects that have changable state
We would like all of the normal properties of numbers for complex numbers

There is a class complex defined in the standard template library that is similar to this example
class Complex

class complex {
    complex(const int& re = 0, const int& im = 0);
    complex(const complex&);
    int real() const;
    int imag() const;
    complex& operator=(const int&);
    complex& operator+=(const int&);
    complex& operator-=(const int&);
    complex& operator*=(const int&);
    complex& operator/=(const int&);
    complex& operator=(const complex&);
};

complex operator+ (const complex&, const complex&);
complex operator+ (const complex&, const int&);
complex operator+ (const int&, const complex&);
complex operator- (const complex&, const complex&);
complex operator- (const complex&, const int&);
complex operator- (const int&, const complex&);
complex operator* (const complex&, const complex&);
complex operator* (const complex&, const int&);
complex operator* (const int&, const complex&);
complex operator/ (const complex&, const complex&);
complex operator/ (const complex&, const int&);
complex operator/ (const int&, const complex&);
complex operator+ (const complex&);
complex operator- (const complex&);
bool operator== (const complex&, const complex&);
bool operator== (const complex&, const int&);
bool operator== (const int&, const complex&);
bool operator!= (const complex&, const complex&);
bool operator!= (const complex&, const int&);
bool operator!= (const int&, const complex&);
template<class int, class charint, class traits> basic_istream<charint, traits>& operator>>(
    (basic_istream<charint, traits>&, complex&);
template<class int, class charint, class traits>
basic_ostream<charint, traits>& operator<<(  
    (basic_ostream<charint, traits>&, const complex&);
standard template library complex

class complex {
    public: typedef T value_type;
    complex(const T& re = T(), const T& im = T());
    complex(const complex&);
    template<class X> complex(const complex<X>&);
    T real() const;
    T imag() const;
    complex<T>& operator= (const T&);
    complex<T>& operator+=(const T&);
    complex<T>& operator-=(const T&);
    complex<T>& operator*=(const T&);
    complex<T>& operator/=(const T&);
    complex& operator=(const complex&);
    template<class X>
    complex<T>& operator= (const complex<X>&);
    template<class X>
    complex<T>& operator+= (const complex<X>&);
    template<class X>
    complex<T>& operator-=(const complex<X>&);
    template<class X>
    complex<T>& operator*=(const complex<X>&);
    template<class X>
    complex<T>& operator/=(const complex<X>&);
};

template<class T>
complex<T> operator+ (const complex<T>&,
    const complex<T>&);

template<class T>
complex<T> operator+ (const complex<T>&,
    const T&);

template<class T>
complex<T> operator+ (const T&,
    const complex<T>&);
...