Usage of Classes

Stacks

A stack is a data structure that supports the following operation:

- Initialization (as empty stack).
- Pushing an object on the stack.
- Viewing the element on top.
- Removing an element from the stack.
- Asking for length of the stack.
- Restoring the stack to a certain length.
- Clean up of non-empty stack.
- Making a copy of a stack.
- Printing a stack.

Stacks, Not Object-Oriented

```
struct stack
{
   std::string val;
   stack* previous;
};
   // Stack is a linked list. Top element comes first.
...
{
   stack* s1 = 0;
      // Creates stack, initializes it empty.
```

```
void push( stack& *s, const std::string& val )
{
   stack* s1 = new stack;
   s1 -> val = val;
   s1 -> previous = s;

   s = s1;
}
```

```
void cleanup( stack& *s )
{
   while(s)
   {
      stack* s1 = s;
      free s;
      s = s1;
   }
}
```

Stacks, Not Object-Oriented

User of a stack would like to write

```
{
   stack s;

   push( s, "today's a fine day for science" );
   push( s, "what a great day for science" );

   stack s1 = s;

   // Push some things more on s.

   s = s1;
      // Inefficient, but it should be possible.
};
```

Stacks, Not Object-Oriented

Stack is an incomplete type:

We have to remember all the time, that a stack has to be represented by a pointer.

A stack cannot be initialized with =, nor assigned.

Passing a stack as parameter to a function is problematic. (Because copying is problematic.)

```
{
   stack* s1;

   ... push a few things on s1.

   stack* s2 = s1;
      // user probably thinks that s1, s2 are
      // different stacks but they are connected.
```

```
    push( s2, "fine" );
    pop( s2 );
    pop( s2 ); // Now s1 is corrupted.
}
```

User has to remember not to use =, to be careful with passing a stack as a parameter.

Stacks, Not Object-Oriented

```
do_something( stack* s )
{
   ... push a few things.
   ... pop a few things.

}

main(   .... )
{
   stack* s1;
   do_something(s1);

      // Should not change s1, but it probably does.
}
```

Data Structures, Not Object-Oriented

Same problem happens with all objects of unbounded size: Stacks, Vectors, Lists, Search Trees, Infinite Precision Integers.

Classes

Make it possible to hide implementation.

Make it possible to hide memory management, so that they can be copied and assigned with same easyness as primitive types.

Being able to invent a good class design is key to writing dependable, extendable, readable programs.

General Form of a Class Definition

```
// class and struct mean almost the same in C++:

struct myclass
{
   int x;
   double y;
   std::string s;   // Members.

   myclass( );      // Default constructor.
   myclass( const myclass& m ); //  Copy constructor.
   void operator = ( const myclass& m );
                                  // Copying assignment.
   ~myclass( );   // Destructor.
```

General Form of a Class Definition (2)

```
    int method( int x );           // Method.
    double method( int x ) const; // Const method.

};

std::ostream& operator << ( std::ostream& stream,
                            myclass cl );
std::ostream& operator << ( std::ostream& stream,
                            const myclass& cl );
    // Printing operators.
```

General Form of a Class Definition (3)

As said earlier, the class definition mixes implementation (the member fields), with the interface (the declarations of the other methods.)

I view this as a design failure of C^{++}, but one has to get used to it.

Essential and Important Member Functions

Whenever you define a class, you should define it in such a way that it can be later used in a convenenient way.

A class has object-semantics if it can be declared, copied and assigned in the same way as `int` or `std::string`.

Whenever possible, make sure that your classes have object semantics.

In order to obtain object-semantics, you should give attention to the following methods:

1. The essential methods: Constructors, Copy Constructor, Copying Assignment, Destructor. Note that C^{++} provides defaults. You should write your own methods only when the defaults are not right.
2. Important Member functions: Printing, Other Operators.

The Essential Member Functions (Default Constructor)

Default constructor. The default constructor is used when you declare a variable without providing an initializer.

```
date d;
complex c;
```

If a constructor for some class does not initialize one of the members of the class, the compiler will insert the default constructor.

Essential Member Functions (Default Constructor)

A default constructor is present for a class

1. when the user defines one.
2. when the compiler defines one. The compiler defines a default constructor only for classes that have no user defined constructors.

In all other cases, there is no default constructor. This may sometimes lead to unexpected error messages.

If the class has no reasonable default value (like **date**), you should not define one out of lazyness.

It is perfectly possible to define classes without default constructor.

Essential Member Functions (Copy Constructor)

The copy constructor constructs a new object from a reference (either **const** or **non-const**) to another object of the same type. It is used when passing a parameter to a method, when a vector is resized, and possibly when a method returns a value.

If you don't provide a copy constructor, the compiler will create one by calling the copy constructors of the class members.

Essential Member Functions (Copying Assignment)

Copying assignment. Assignment copies one object into another object. The difference with the copy constructor is that in case of assignment, an existing object needs to be overwritten.

If you don't provide a copying assignment, then the compiler will create one by calling the copy copying assignments of the class members.

Essential Member Functions (The Destructor)

The destructor is called when the object goes out of scope.

In case the object uses space on the heap, the destructor has to clean up the heap space. Otherwise, there is a memory leak.

I showed last weak how to detect them, at least for simple leaks. Most leaks are simple.

If you don't provide a destructor, the compiler will create one by calling the destructors of the class members.

Usage of the Essential Members

The copy constructor, copying assignment and destructor are essential when the class has memory on the heap. (like **list** or **vector**)

In many other cases, (like **dates** or **complex numbers**), they are not important.

Don't define your own methods, when the default methods do their work.

Usage of the Essential Members

Some uses, different from memory management:

File Handle Constructors open the file. Usage of copying assignment is blocked (by declaring one, but not providing one), the destructor closes the file.

Backtracking Point for Stack Constructor remembers state of the stack. Usage of copying assignment is blocked. Destructor restores the stack.

Other important member functions

A print operator `<<` .

There is no default print operator. Attempts to print an object, that has no print operator, results in long, incomprehensible error messages.

Print Operator

Unfortunately, the print operator is not a member function. One solution is as follows:

```
std::ostream& operator << ( std::ostream& stream,
                            const X& x )
{
   x. print( stream );
   return stream;
}

// In class X:
void print( std::ostream& stream ) const
{
   // Printing.
}
```

Examples of Member Functions

I will give examples of member functions, and of their use, on a simple **date** class.

```
class date
{
   unsigned int year;
   unsigned int month;
   unsigned int day;
      // One possible implementation.
      // Another way would be to count days
      // since 01.01.2000.
```

Examples of Constructor

```
date( unsigned int year,
      unsigned int month,
      unsigned int day ) :
   year( year ),      // Member initializers.
   month( month ),    // ...
   day( day )         // ...
{
   // Body of constructor. Often does nothing.
}
```

The class members are initialized by constructors of the proper types, when they exist.

Usage of Constructor In variable initializers:

```
{
   date d = date::date( 2011, 3, 29 );
   date d1( 2011, 3, 29 );
```

In field initializers:

```
struct event
{
   date d;
   std::string description;

   event( const date& d )
      : d(d)
   { }   // description is initialized by def. constr.
```

Be careful

Be careful with code of the following form:

```
{
   date d;
      // Calls default constructor. Will not
      // compile when there is no default constructor.

   d = date::date( 2011, 3, 29 );
      // Copying assignment.
```

Be Careful when Using Initializers

```
event( const date& d )
{
   (this -> d) = d;
}
// First, d is initialized with the default
// constructor. After that, d is overwritten.
// Will not compile when there is no default
// constructor.
```

Usage of Copy Constructor

```
date d1(2004,12,2);
   // Calls a constructor.

date d2 = d1;
   // Calls copy constructor.

date d3 = date::date(d2);
   // Also calls copy constructor.
```

In addition, the copy constructor is used for parameter passing to functions.

Copying Assignment

```
void operator = ( const date& d )
{
   year = d. year;
   month = d. month;
   day = d. day;
}

   // Note that this is a completely useless
   // assignment operator.
```

Copying Assignment

```
void operator = ( date d )
{
   year = d. year;
   month. = d. month;
   day = d. day;
}
   // Same as previous, but there is another
   // call of copy constructor.
   // Note that this operator is also useless.
```

Destructor

```
~ date( )
{
   // Body of destructor.
}
```

In the case of date, nothing needs to be done. Therefore, the whole desctructor can be omitted.

More Uses of Essential Member Functions

```
{
   date *p = new date( );
      // Default constructor.
   date *q = new date::date(2000, 1,1):
      // Constructor.

   *p = *q;
      // Copying assignment.

   delete p;
      // Destructor.
   delete q;
      // Destructor.
}
```

More Uses of Essential Member Functions (2)

```
{
   unsigned int k = 40;

   date *p = new date [k];
      // fourty times the default constructor.

   delete p;
      // fourty times the destructor.
}
```

Also, in all of the standard containers, like
`std::list`, `std::vector`, `std::map`, etc.

More Uses of Essential Member Functions (3)

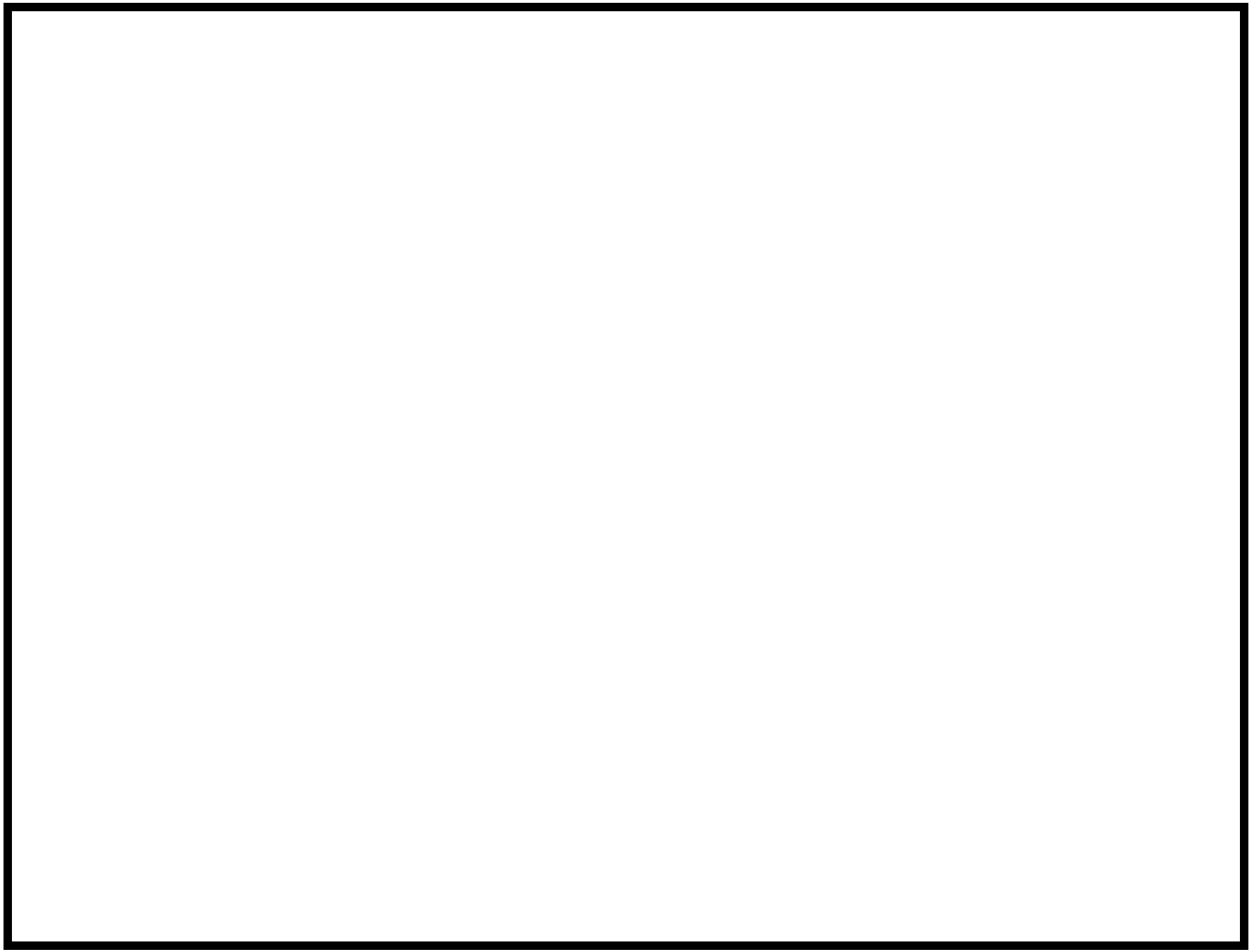
```
{
   date *p = new date( );
      // Default constructor.
   date *q = *p;
      // Copy constructor.


   p = q;
      // At this point, *p is simply lost. No special
      // function is called.
}
   // The pointers p and q go out of scope, but *q
   // remains in memory. No special function is
   // called.
```

Other Member Functions

A member function is either **const** or non-**const**. Default should be **const**. Make it non-**const** only when the member function can change the object.

```
unsigned int days_since( const data& d ) const
{
   // compute number of days between d and *this.
}
void setdate( unsigned int year,
              unsigned int month,
              unsigned int day )
{
   (this -> year) = year;
   (this -> month) = month;
   (this -> day) = day;
}
```



Usage of This

In a class method, the fields can be addressed by their names.

In case a local variable/paramter has the same name, the members can be addressed by `*this`, which unfortunately has type `X*` or `const X*`.

It should have been a reference, but references were introduced only later into C^{++}.

```
unsigned int getyear( );
unsigned int getmonth( );
unsigned int getday( );
unsigned int yearlength( );
unsigned int monthlength( );
bool isleapyear( );
```

- Increasement/descreasement $++, --$
- A function $-$ that computes the difference in days between two dates.
- Functions $+, -$ that can add an integer to a date?
- Operators `<`, `>`, `<=`, `>=`, `==`, `!=` ?
- Operators `+=`, `-=` ?

Summary

C^{++} has explicit memory management, which makes it a bit harder than Java and $C^{\#}$ say. But if you understand how the constructor/destructor mechanism works, it is not hard.

Always write good constructors, and a << operator.