C++ [exam] [2007]

[Based On The Lectures Of Moti Freiman && Ofir Pele]

Collected, extended upon and written by: Aviad Pines && Lev Faerman.

Thanks To: Dana Livni && Yuval Polachek

Contents {

};

Differences Between C && C++

General

First, every single piece of code written in C will compile in Cpp. Cpp provides backwards compatibility for C code, and sections of C code may be written inside Cpp code without any worries. This is not usually considered good programming form, unless there is a very good reason for this.

Header Name

All C Header files still exist in cpp and can be accessed by including them as usual. This is, however, not the Cpp way. Minor changes to support better code have been inserted to some of them. The newer files can be accessed by typing (This example includes the c assert.h):

```
#include <cassert>
```

Cpp standard library headers do not end with .h, this is done for convenience and to separate them from user built headers.

Procedural Vs. Object Oriented

Unlike C, Cpp is a language which can be made to act Object Oriented. This is not a must however, and the language leaves much control in the hands of the programmer (Unlike Java).

Many entities previously existing as structs or other types in C are now objects in Cpp. However, Cpp objects differ greatly from Java.

I/O

Even though printf and scanf still exist, they are obsolete and are replaced by std::cout and std::cin respectively. Syntax:

```
std::cout << <var_name1 << var_name2 << std::endl;

std::cin >> var_name1 >> var_name2 ;
```

As you can see we don't need to specify the type of the arguments we want to send or receive, since this is done via operator overloading (p. 24) The std::endl is the preferred way to enter a newline character since its also flushes the buffer

Cin has a special feature that it can enter fail mode, which will tell the programmer that an error had occurred during input retrieval and easily let the user decide the proper approach instead of going crazy, e.g:

```
if(std::cin.fail()) {

      std::cerr << “Error retrieving data” << std::endl

}
```

File I/O

File I/O acts exactly like regular I/O since we can redirect cin && cout to any place we wish.

Strings

Unlike in C, where a string is simply a null-terminated character array, in Cpp strings are their own type of Object. Almost all familiar methods from C have been ported to Cpp method form, and if a few were missed, the string class contains a constructor which accepts a C style string as a parameter. Coding C strings in Cpp, however, is considered bad form, unless it is for a very good reason. Cpp strings manage their own memory usage, freeing the programmer from the need to watch for buffer overrun.

Types && Qualifiers

General

All familiar types from C exist in Cpp. Additional types have been added as outlined below. In addition, while in C types are transferred only by value (Usually with bad results, especially in pointer containing structs) or by pointer and in Java everything except for primitives is passed by reference, in Cpp we are completely free to choose how we pass our information around: By value (even highly complex types, if required), by pointer, or by reference.

Objects

Cpp objects form their own type, similar to the way that a struct in C would define a type. This definition is automatic, and the name of the type is simply the name of the class. Unlike Java, Cpp objects do not share a single type hierarchy and a single master parent class like Object does not exist. Sometimes a similar concept exists within a given library. This is because unlike Java, Cpp is not a pure Object Oriented language, and not everything in Cpp belongs to a class. C style structs exist in Cpp and define a class (object) type of their own, also automatically. For the differences between Cpp structs and Cpp classes see page 23.

Boolean

Unlike C, where boolean variables were simply other variables, with zero denoting false and any other value denoting true, Cpp comes with a dedicated boolean type - *bool*. The bool type is identical to the Java boolean type.

Static

Static receives another meaning in Cpp over the two existent in C. A static class member behaves just like a static class member in Java, that is it is shared across instances of the same type who only point to it.

Members that are static && const can be initialized inside the .h file.

Static variables inside a function behaves just like in C, with one catch – if you declare a static variable inside a member function of a class, the variable will behave like a static class variable as well – it will be shared through all the instances of the class.

In C++ static variables have a special initialization – the declaration inside the class is merely a prototype declaration (to avoid collision of global variables) and the real declaration needs to be done inside the .cpp file which contain the class definition. e.g:

inside the header file:

```
class Fubar {

      static int foo;

      int stfoo;

}
```

inside the cpp file, at the beginning:

```
      int Fubar::foo;
```

or, if we want to initialize it:

```
      int Fubar::foo = 5;
```

Const

The const qualifier has another meaning in addition to the meanings it has in C. Const objects and const methods are now available to declare immutable objects and methods that do not change the properties of their objects. For more information, see page 10.

References

The way Java handles objects is by reference, and in cpp you have the choice to send your variables (primitives or objects) as references as well – sending the same instance via a function making the code more optimized and / or the variable mutable via the function itself.

Reference object are basically the same variable with different name, since it is the same memory address and changes applied to it will cause changes on the original as well.

This is essentially a const pointer, with all the annoying pointer usage such as * and -> omitted from it. We cannot perform any pointer arithmetic on it (just like a const pointer)

Syntax:

```
void printPersonData(Person &p) {

      p.doSomething();

}
```

The & tells the compiler that the variable sent to it is passed by reference. To optimize a program one can right that a variable will be sent through const reference, thus eliminating the need to copy that variable, while still not allowing any changes to be applied to it.

Be careful when sending a reference as the function's return value since the stack of the function collapses and the variable will point to uninitialized memory. So when returning a value be sure that the memory it points to will remain when function ends.

Important Note: You cannot make a reference to a temporal! For example, the following code will not compile:

```
void print(int &a, char &b) {

      std::cout << a << " " << b << std::endl;

}


int main() {

      print(3,'c'); // Reference cannot be made to a temporal type.

}
```

The relevant error:

```
test.cpp: In function ‘int main()’:

test.cpp:11: error: invalid initialization of non-const reference
of type ‘int&’ from a temporary of type ‘int’
```

Default Parameters

In general, if a function prototype declares that it receives a given number of parameters of given types, it cannot be called with less parameters. In Cpp, there is a way around that. We can use default parameters in the following fashion:

```
int addAndPrint(int a = 0 , int b);
```

This function prototype is supposed to add two numbers, print the result and return it. However, the following code is possible:

```
int result = addAndPrint(10);
```

The result would be the addition of 10 and 0 (The default value for a) and the printing of the result of 10. result will then receive the value 10, as well.

The rule for default parameters is once you declared it to at least one of your parameters, all parameters that follow it must be declared with default parameters as well.

Default parameters are written only inside the .h file (considered more elegant coding)

Casting

The old-fashioned C casting is still available but considered bad coding since it is more open to many kinds of bugs. The new Cpp casting system is much more syntax ugly and much more specific. See more information about casting on page 28.

Functions

Friend

Syntax:

```
friend <return_value> <function_name> ( <parameters> ) ;
```

e.g:

```
friend void printInt(int& x) ;
```

Inside a class private and protected members cannot be accessed from outside the same class in which they are declared. In some cases however, we would like that an outer function will have privileges to access the private data of the class (e.g. operator<<). This can be accomplished by declaring a function / class as a friend.

Friendship allows the friend function / class to access the data of a class that had declared them as friends as it was a member inside of the class itself (except that it cannot use the *this* variable).

Friends are functions or classes declared as such.

If we want to declare an external function as friend of a class, we do it by declaring a prototype of this external function within the class, and preceding it with the keyword friend:

```
class Person {

public:

      friend void setX(*this, int);

      private:

      int _x;

};


void setX(Person& p, int newX) {

      p._x = 5;  //we can do this since we declared friendship

}
```

As we can see now setX can access the private parts of class Person as if it was a member of the class.

Notice that the friend keyword is written in the function prototype inside the class declaration, and not in the global friend function itself.

Inline

Syntax:

```
inline <return_value> <function_name> ( <parameters> ) {...}
```

e.g:

```
inline void printInt(int& x) {...}
```

The inline keyword is used to suggest (as in not obligatory) to the compiler that a particular function be subjected to inline expansion; that is, it suggests that the compiler insert the complete body of the function in every context where that function is used and so it is used to avoid the unnecessary jumps from one place in code to another and back again to execute a short subroutine that is much better to be written explicitly in the code by the compiler before compilation.

Inline increase the code size, especially when not used in a smart manner. It can, however decrease by much the running time of the application.

One more important thing about inline function is the effect on linking: multiple definitions of an inline function are permitted as long as each is in a different module, and as long as *they are identical*. This allows inline function definitions to appear in header files, as oppose to non-inline functions which will almost certainly cause an error by doing that.

Inline can also be declared implicitly, by defining a member function inside a class/struct definition (which also explains the way we can define short methods inside the header file).

Notice that the inline keyword is written where we write the definition of the function, and not it's prototype.

Const

Syntax:

```
<return_value> <function_name> ( <parameters> ) const ;
```

e.g:

```
void printInt(int& x) const ;
```

The const keyword applies only to member function. The keyword is a binding contract which states that that function cannot modify the member variables of its class. The keyword is a heavy sign to the programmer that a given member function does not change the internal state of a class, and any attempt to change the class members will cause compilation errors.

You can overload function that their only difference is there const modifier, e.g:

```
void foo();

void foo() const;
```

are two different functions. The const function will be called only when const object are required whether we declared them as such or a function receives them as const parameters. The other function will be called at all other situations. This overloading is often used with iterators.

Virtual

Syntax: (written inside the class declaration in the header file)

```
virtual <return_value> <function_name> ( <parameters> );
```

e.g:

virtual void printInt(int& x) ;

C++ virtual function is a member function of a class, whose functionality can be over-ridden in its derived classes. Since in C++ polymorphism is not automatically built in, the virtual functions are the way to implement polymorphism in C++. For more information see the section about polymorphism, page 19.

Classes && Objects

General

The concept of Object Oriented programming is familiar to us from Java. The only thing in C that resembled OOP was the struct, but it was an extremely poor and primitive version.
C++ is, like Java, object oriented. Unlike Java however, it's not pure object oriented, thus not *everything* has to be placed inside classes. The classes of C++ are less pure, but much more powerful, allowing the user to gain more power from his application that was not available in Java programming.
Note that C++ supports the C style structs, and structs that are constructed with only C variables will be compiled as a C struct. In C++ the full strength of a struct is precisely like a class, only that the structs default visibility is public, whilst a class is private.

Writing A Class

Class writing in Java was an easy task and the most basic one, since every single tiny program is written inside a class. In C++ you have to work a little harder since the class is less natural there. Each class is divided into two separate files – The header file which contains its definition and prototype declarations, with few exceptions such as inline functions and templates (See Templates on page 34), and the definition of the methods and initialization of the static variables.
The syntax for declaring a class in the header file is:

```
class <name> {
public:
      <methods and members>
protected:
      <methods and members>
private:
        <methods and members>
    } ; // Note the semicolon! Code will not compile without it!
```

A more concrete example:

```
    class MySmarterStruct {
    public:
        int _num;
        long _bigNum;
        int getNum() const;
    } ;
```

The syntax for implementing the class methods in the .cpp file is:

```
    <return type> <name of class>::<name of function>(<parameters>);
```

Needless to say, that the implementation signature must be identical to the prototype declared in the header. For the example of MySmarterStruct it would be:

```
    int MySmarterStruct::getNum() const {
        return _num;
    }
```

Constructors && Destructors

Constructors

In C we were used to coding simple "poor man's" ctors which would use malloc() to allocate the memory for our struct and initialize members. These functions were not methods as they were not a part of the struct itself. Since in Cpp we have classes, it also offers us the ability to write a real constructor, like the one we are used to in Java, and as in java the ctor is a special function that's declared with a name identical to the class and without a return parameter, like this:

```
    class MySmarterStruct {
    public:
        MySmarterStruct(int num, long bigNum);   // constructor
        int _num;
        long _bigNum;
        int getNum() const;
    } ;
```

As you can see above, a constructor can accept parameters as any other function.
If we do not specify a constructor at all, a default constructor which receives no arguments will be

auto-generated.
Now we have a ctor, but how do we create an object? Note that Cpp differ from Java in this point, since there are several ways to do this. In java, writing:

```
MySmarterStruct a;
```

Would allocate a pointer to the heap, ready to contain a MySmarterStruct object. What would happen in Cpp ? The answer is: The default ctor of MySmarterStruct would be called and an object named 'a' of type MySmarterStruct would be allocated on the Stack. Often we did not quite mean for that to happen. In the specific example we have here, the code would not compile, since MySmartStruct has no ctor which receives no arguments. To allocate a MySmarterStruct as defined above we would have to write:

```
MySmarterStruct a(5,10); // Or some other values.
```

This creates a MySmarterStruct with the int value of 5 and the long value of 10 on the Stack.

How to duplicate the Java behavior ? We are already familiar with the concept of pointers, which apply to structs and classes, thus we will need to write the following:

```
MySmarterStruct * a;
```

This will create a pointer to a MySmarterStruct type, but it will not initialize anything.

IMPORTANT: Constructors are not inherited! (inheritance on page 17)

The Copy Constructor

A special kind of ctor is the copy ctor. A copy ctor receives a const reference to an object of the same type. In our example it would look like this:

```
MySmarterStruct(const MySmarterStruct& value);
```

(Notice the const reference as a parameter – this is a must!)
The purpose of the copy constructor is as its name suggest – to construct an object that is the identical clone of another object given as a parameter. As with the default constructor, if no copy constructor is specified, a default one will be created by the compiler, which will simply copy every member inside the given object *by value*. As in structs, we saw that copying by value is absolutely fine as long as we are dealing with primitives, thus the default copy constructor is great and even recommended in simple classes (note that the same trick with array copying with structs works in C ++ as well, with classes too). But what happens when we deal with pointers and other structs / classes?
Pointers can create a real mess when copied by value as we've encountered in C, and when copying other objects we need to rely on their copy constructors. This means that when we have pointers in our class, it is extremely recommended to create a copy constructor and not rely on the default one.
Copy constructors can be called explicitly or implicitly by the compiler when needed. Each time an object is passed by value, the copy constructor is invoked, so each time you send your object

through a function, you actually activating your copy constructor. To invoke the copy constructor explicitly:

```
MySmarterStruct a(3,5);

MySmarterStruct b(a);
```

Another way to call the cpy ctor:

```
MySmarterStruct b = a;
```

And an example of an implicit call:

```
MySmarterStruct doSomethingElse() {

        MySmarterStruct a(50,986);

        return a;           // Return by value invokes copy ctor !

}
```

In order to avoid the creation of objects in the previous examples (and in many cases avoiding the implementation of a copy ctor altogether) we can simple pass objects by reference instead of by value, for example:

```
void doSomething(MySmarterStruct &mss) {...}

MySmarterStruct a(3,5);

doSomething(a);     // cpy ctor not called - transfer is by reference
```

IMPORTANT: Since constructors are not inherited, the copy constructor is not effected by hierarchy. This means that if a child won't declare a copy constructor that calls the parent's copy constructor all of its members including the ones that are inherited will be copied by value! That's because the child will not call unless explicitly told to to the parent's copy constructor!

The Initialization List

In Java, the code block inside the constructor was responsible for initializing each and every member of the class, and, if needed, we could call another constructor or parent's constructor as well. In Cpp however, sometimes we need to initialize values even *before* we enter the constructor's code block.

Consider the following code:

```
Class A {
      A(int&);
}
```

```
Class B {
public:
      B()
private:
      A _a1, a2;
```

```
int main() {
      B b;
}
```

```
B::B() {
  _a1 = 7;
  _a2 = 5;
}
```

Similar code in Java will have no problem at all – B's constructor will be called, and inside of it we can initialize _a1 and _a2, and go on with our program.
C++ will not compile this code since A does not have a default constructor. Why? That's because in C++ when a class enters its constructor code block, all of the data members has to be initialized. In case of primitives and pointers, it's no problem since they will be initialized with junk. So, if we have a default constructor it will simply be called before we enter the code block, but what happens if we don't have one? As we've seen this will result in a compilation error.
The way we fix it is using the constructor's *initialization list*. This is, as the name implies, a list that comes before the code block allowing us to initialize all the data members we please (objects, pointers && primitives). This is a unique feature of the constructor and its considered elegant coding to use it for all data members, not only objects.
Syntax:

```
<class_name>::<class_name>(<param>):<data_member>(<Value>) {...}
```

So, to fix our problem above we would fix B's constructor to be:

```
B::B( ) : _a1(7), _a2(5) { }
```

Destructors

In Java we had the garbage collector to handle our allocated-but-not-needed-anymore memory, but in C++ since we are responsible for allocating and freeing memory, we have to destruct our object when we don't need it anymore. That's why we have the destructor. The closest thing we had to something like that in C sort of freeStruct() function we implemented on structs.
The destructor is called implicitly for every object that is created on the stack is called when that object's scope ends, or when an abnormal method termination in the form of an exception being thrown (See exceptions for specific information, page 40). This means that if we have created proper destructors we do not need to worry about memory leaks for stack objects (note that the objects themselves can allocate memory on the heap and release it in the destructors). However objects that were created on the dynamic heap needs to be freed from memory explicitly (outlined in next section)

The syntax for writing destructors is identical to that of writing ctors, but the name of the class is prefixed with a tilde ('~'):

```
class MySmarterStruct {
public:
      MySmarterStruct(int num, long bigNum);    // constructor
      ~MySmarterStruct();                       // destructor
      int _num;
      long _bigNum;
      int getNum() const;
} ;
```

Note the oppose to constructors, desctructors cannot receive any parameters, and thus cannot be overloaded within the same class.
What do we free in a destructor ? We should free everything we allocated on the heap during the creation of our object (and during other operations, if applicable). We should not try to free any members not initialized on the heap. These will be automatically freed when the destructor is called and we need not write anything to accomplish that.
IMPORTANT: Destructors are not inherited! (on inheritance and polymorphism involving destructors see page 23)

New && Delete

In C we had only a single way of allocating and freeing memory: Allocate by malloc() and free by free() (We also had the derived methods of calloc() and realloc() but they had specific uses). In Cpp these old functions still exist, but are not used (unless for a really specific task), and considered pretty much obsolete. The Cpp way of allocating and freeing memory is accomplished by the **'new'** and **'delete'** commands.
'new' signals to the compiler the the following type should be initialized on the heap. The result of the statement is a pointer to the relevant type which points to a location on the heap. This is the Cpp equivalent of malloc() and is smarter since it calls the constructor in case it was used on an object.
Example:

```
      MySmarterStruct * temp = new MySmarterStruct(5,4);
```

Note: Unlike Java, the new command can be used on anything, even primitives. The following is completely legal:

```
      int *numP = new int(267);
```

This allocates an int with the value of 267 on the heap, but in this case since it is not an object no constructor will be called.
'delete' signals to the compiler that it needs to free the memory that the argument takes up. It can only

be called on things allocated on the heap, doing the following:

```
int main() {
        int i = 5;
        delete &i;
}
```

Will result in a segmentation fault due to an invalid free memory operation.
In the case of a primitive, the memory is freed, in the case of an object the destructor of the object is called first. The success of the operation of course rests on whether the destructor correctly frees all the memory the object allocated on the heap by itself.
'new' and 'delete' work differently on arrays. For example, to allocate an array of its on the heap we write:

```
new int* arr = new int[10];
```

However, to free the array we use a different delete command:

```
delete [] arr;
```

This tells the compiler that the entity in question is an array and must be sequentially freed. Using 'delete' on an array may lead to memory leaks.
IMPORTANT: When initializing array of object, we can only call the default constructor. If you wish to allocate and array with anything but the default constructor in each and every cell, you'll have to create a array of pointers and allocating the cells manually.

Inheritance

Since C++ is supports object oriented programming, it also support the concept of inheritance like Java. The main difference between C++ and Java is that inheritance does not force polymorphism like in Java, but it is the user's decision (See the Polymorphism section below).
The basic inheritance syntax is:

```
class A {...};
class B : public A {...};
```

And this marks class B as an inheritor of class A.
In addition to declaring the inheritance public as written above, we may declare it protected or private with the following effects:

```
class B : public A {...};
```

This means that all methods and members of A retain the same access qualifiers when accessed through B, public remain public, protected remain protected, private remain private.

```
class B : protected A {...};
```

This means that all methods and members of A which were public are now protected methods in B, protected and private methods and members remain as they were in A.

```
class B : private A {...};
```

This means that all methods and members of A are now private when accessed through B and may only be called internally.

Initialization of Base Class Part

Lets examine the following inheritance hierarchy:

```
class Base {
public:
      Base(int a, int b);
      int _val0;
      int _val1;
}

class Derived : public Base {
public:
      Derived(int a, int b, int c);
      int _val2;
}
```

We declared a ctor in Derived which is supposed to initialize both the Base part of the type as well as the extended Derived part. But, will the following implementation in our .cpp file work ?

```
Derived::Derived(int a, int b, int c) {
      _val0 = a;
      _val1 = b;
      _val2 = c;
}
```

The answer is **no**. In the initialization list of each ctor of Derived there is an implicit call to the ctor Base(). Since no such ctor exists, this code does not compile with the following error:
no matching function for call to `Base::Base()' .
The correct ctor implementation would be:

```
Derived::Derived(int a, int b, int c) : Base::Base(a,b), _val2(c) {}
```

Notice the call to the parent class' constructor (C++ version to super())

Method && Member Hiding

In Cpp, just like in Java, we can hide members and methods in inheriting classes. Examine the following code:

```
class Base {
public:
      int a;
      void print();
};
```

```
class Derived : public base {
public:
      int a;
      void print();
};
```

Note that both the parent and child class have members with identical names and methods with identical names. In this example the member and method of Derived hides the ones of Base. How can we use Base's vars then?

To use the parent class' hidden method, or access a hidden member from inside the child class, write:

```
<parent_type_name>::<member_name>
<parent_type_name>::<method_name>(<arguments>)
```

Outside the child class, things get a bit more complicated. In order to utilize the two different members, we need to change our view of the object. By casting to the base type, we can access its member and by casting back to the child type we can access the child's member. They might be other ways, but this is the way we know that works.

As you can see, this gets overly complicated very quickly, and because of that reason it is seldom recommended to hide members. Hiding functions is also not recommended, but virtual functions (See Polymorphism) are an exception to the rule.

Polymorphism

General

Polymorphism in Cpp is an option and not a fact. Inheritance may come along with Polymorphism, but Inheritance may be non-polymorphic as well. A polymorphic class in Cpp is a class which has at least a single virtual method. Without a virtual method, a given inheritance tree is not Polymorphic. Once a single class is Polymorphic any class which inherits it is Polymorphic as well, since a we cannot un-virtual a method.

Virtual && Pure Virtual Functions && Classes

It should be noted, however, that the polymorphism only applies to the methods which are declared virtual. Thus a class may have several methods with polymorphic behavior and several more which lack it. For example:

```
class Base {
public:
      virtual void prtNfo();
      void prtMoreNfo();
};
```

```
class Derived {
public:
      void prtNfo();
      void prtMoreNfo();
};
```

In the above classes, only the prtNfo() method comes with polymorphic behavior, while the prtMoreNfo() method does not. This is completely different from Java, where any and all methods display polymorphic behavior. Note that in the class Derived we did not declare the prtNfo() method as virtual. This does not matter, since it shares the signature of the virtual method Base::prtNfo() it is now virtual as well. It is considered polite to write the virtual qualifier before all functions which are in fact virtual, even if we are not required to do so by the compiler.

A pure virtual function is a function which has no implementation in the class where it was declared, much like an abstract method in Java. Once a class has a pure virtual function it cannot be instantiated (as abstract class or interface in Java), but a constructor might still be required for it in order to be able to initialize the base part of any inheriting classes (Since inheriting classes will need to initialize the members declared in it). The syntax to declare a pure virtual function is as follows:

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

The ' = 0 ' tag marks the method (and the class Base) as pure virtual, with the previously explained results.

Important: Any polymorphic class must have a virtual destructor. While everything will compile fine without one the runtime results will not be pretty, since the wrong destructor may be called for the actual object, resulting in a memory leak.

The compile time and runtime mechanisms which choose which method to run are detailed below.

Static Resolution

Assuming Base and Derived are the two classes from the previous section, lets examine the following code:

```
Base * b = new Derived();
b->prtMoreNfo();
```

Which prtMoreNfo() will be called? The one from Base. Since prtMoreNfo is not a virtual function, it is called depending on our compile-time view of the object (the type of the variable and not the actual type of the object), which the compiler sees as a Base instance. This is called static resolution.

Dynamic Resolution

Again, let us inspect some code involving our earlier Base and Derived classes:

```
Base * b = new Derived();
b->prtNfo();
```

This time, the method that will be called will be Derived::prtNfo(). This is because prtNfo() is declared virtual. This virtual keyword instructs the compiler to determine the most specific type during run-time, rather than at compile time. This is called Dynamic Resolution and is less efficient since it imposes a run-time check to determine the exact method to run. How is this done ? By referencing the Virtual Table of the object.

Important Note: In Cpp the overriding of a virtual method in the child class must have precisely the same signature, not even sub-type relations will work ! The following code does not create a single virtual function with polymorphic behavior, but rather declares two different virtual functions, one in class Base and the other in class Derived. Here is the example:

```
class Base {
public:
virtual void printSomething(const Base &) const {...}
} ;

class Derived : public Base {
public:
virtual void printSomething(const Derived &) const {...}
} ;
```

Even though Derived is a sub-type of Base, the two functions will not display polymorphic behavior in the following code as they are not sharing the same signature:

```
Base *b = new Derived();
Base base;
Derived derived;
b->printSomething(base);
b->printSomething(derived);
```

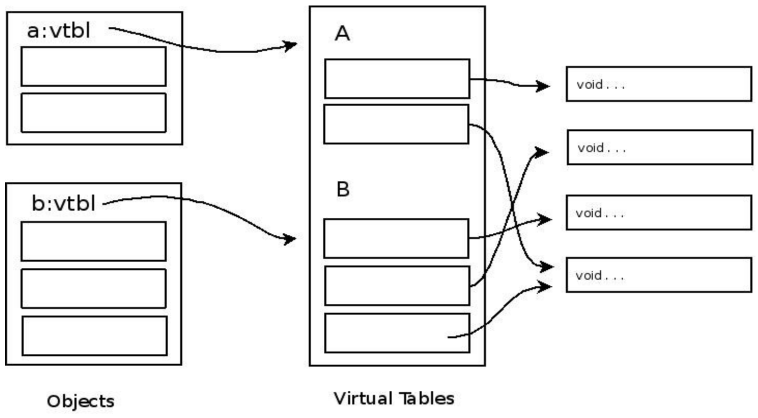
In both cases Base::printSomething will be the method that will be called. If we were to define Derived::printSomething as follows, however, polymorphic behavior would occur:

```
class Derived : public Base {
public:
virtual void printSomething(const Base &) const {...}
} ;
```

If we want to obtain polymorphic qualities in these functions, will have to define the derived printSomething as the latter and inside of it cast it into a Derived object. Remember to catch an exception in case you actually sent Base.

Virtual Tables

The virtual table is a table of pointers to functions in memory. Any function which is declared virtual is essentially an additional pointer member of our class which we cannot see or change. This pointer points to a location on the table, which in turn points to the actual function code. Each virtual class (each type, not each instance) has a virtual table which points the virtual functions to their actual implementation relevant to the class. An illustration:



Each class has has its own virtual table, and as we can see, each object has a special pointer referring to its class virtual table. The virtual tables themselves are arrays of pointers, which point to the actual function code blocks used by their representative classes. When a virtual function is called the function pointer of the object is dereferenced to the location on the virtual table, which is then dereferenced to the actual function code. This is far slower than a static resolution function call, efficiency wise, as it deals with memory access, so the overhead incurred by dynamic resolution can be significant. Here is an example of the process:

```
Base * b = new Derived();
b->prtNfo();
```

Upon the function call for the virtual function prtNfo() the hidden member pointer inside b is dereferenced to a location on the virtual table. This in turn is dereferenced to the location in memory containing the code of Derived::prtNfo() and the function is called.

Pointers to virtual table locations cannot be changed in any user-caused way. Neither copy ctor, nor an operator= and not even any type of cast change the value this pointer holds. This allows dynamic resolution to function as intended.

Virtual Constructors && Destructors

Simply put: Virtual ctors do not exist. Since at the moment of construction the object does not exist yet, then obviously the pointer to the virtual table location doesn't exist either, making the whole concept impossible.

Since destructors are essentially like any other function, virtual dtors are important. The moment we have inheritance, we immediately should declare a virtual dtor, except for some highly unusual cases. If we do not, the static resolution may call for the wrong dtor, leaving us with a memory leak that may be extremely difficult to find. It should also be noted that in implementing a child class' destructor, we do not need to call to the parent class' destructor since this is done automatically.

Classes Vs. Structs

A class is an extended version of a C struct, but both exist in Cpp and may both be used. In Cpp, a struct defines a class. This means that it can come with a ctor, a dtor, operators, a copy ctor and public/protected/private definition of functions and members. The only difference between a struct and a class (Other than that set in the mind of the programmer) is the default access permissions. If no permission is specified, then the default access to a struct member is public (To allow for backwards compatibility with older code), but in a class, the default access permission is private:

```
struct Silly {
      int _num; // public !!
} ;

class Sillier {
      int _num // private !!
} ;
```

Structs are usually used for inner classes (Such as nodes) which hold data, but require no special methods or treatment. If anything more complex is required, a class is usually declared. Note that this is a more conceptual use of structs and classes and is not forced by the compiler.

Nested Classes

It is possible to define a class within a class, or a struct within a class. This is called a nested class. If defined under the private header of the outer class, then it cannot be instantiated outside the outer class. If defined public, it can be instantiated anywhere. Unlike Java, there is no runtime connection between the nested class and the outer class. You cannot access the members of the outer class inside an inner class instance. This can be simulated by passing along a reference on creation of an instance and/or declaring the nested class as a friend class of the outer class. In addition, two nested classes of the same class have know knowledge of each other's existence. Nested classes serve as logical organizers of code, but have no meaning during actual run-time.

An example:

```
class Outer {
      public:
            class Inner1 {...};
      private:
            class Inner2 {...};
} ;
```

Class Inner1 can be instantiated anywhere, but class Inner2 can only instantiated within methods of Outer and within itself. Inner2 will be able to instantiate objects of inner1 type since it is visible to it (visible to all, not because of the nesting) but Inner1 will not be able to instantiate Inner2 since it does not know of its existence.

The syntax to implement methods of Inner1 and Inner2 is ugly. Assuming both have a void method called print, we would have to write the following to implement them:

```
void Outer::Inner1::print() {...}
void Outer::Inner2::print() {...}
```

Operators

General

We use operators all the time to perform different mathematical computations on primitives. In C we expanded the operators a bit when defining pointer arithmetic and creating non-mathematical operators such as * and -> to allow easy code for very commonly used functions. C++ takes another step and lets us overload operators like any other function. Almost every operator that you can think of can be overloaded to do whatever you implement it to do. Be careful however not to go wild with it – the golden rule of operator overloading is to make it work on your object in the same spirit that it works on primitives. e.g using the '+' operator to concatenate strings is logical, but using the ++ operator to clear a linked list has no logic into it and will surely cause problems.

Operators divide into two major types – member function operator which are defined inside a class and accept zero or one arguments (since the *this member is always sent to them) and global operator function which accept one or two types (left side or left and right side of the operator).

Note that you cannot define the built in operators defined in C++ such as on the primitive types.

Assignment Operator '='

This is a special operator and need to get its own spot.

We use the assignment operator all the time between primitives and pointers, using the function '='

which receives two arguments by reference and assigns the right side of the operator into its right side.

For example int assignment function will be:

```
int& operator=(int&, const int&)
```

The assignment operator is the only operator that has a default value for every object as well.

```
<object>& <object>::operator=(const object&)
```

The reason that the return value is by reference is to allow chaining such as:

```
a = b = c = 5;
```

The default operator simply copies all members of a given object by value, which can result in pointer and memory problems just like we encountered on the cpy ctor, so when defining large objects it is important to define this operator as well or disable it (will be elaborated on).

Assignment Operator Vs. Copy Constructor

As we can see there is a lot of similarity between the assignment operator and the copy constructor, and we need to define them both properly in order to make our objects tight proof against malicious errors and segmentation faults. First let's see when each of them is called:

```
Person B, A;
B = A;              // uses '=' operator
Person C = D;       // uses cpy ctor
Person E(F) ;       // uses cpy ctor
```

The only difference then, is that the assignment operator is called when the object that is calling already exist, and the copy constructor is called upon construction of an object. For this reason is it mostly considered proper coding to implement an init function that initializes and actually copies the object, and the only difference then between the cpy ctor and the '=' operator is that the operator will have to check if it has any allocated memory in its object, and free it before copying the values using the init function.

```
Foo::Foo(const Foo& other) {
        init();
}
```

```
Foo& Foo::operator=(const Foo& other) {
        if(memory_allocated_already)
                delete_memory;
        init();
        return *this;
}
```

Now, one more trap that we need to avoid is self copying. What happens if someone will right this redundant line:

```
NotFoo f;
f = f;
```

This can cause the memory to be deleted right at the beginning of the operator= function! The solution is short and simple:

```
Foo& Foo::operator=(const Foo& other) {
        if(this == other)
                return *this;
        if(memory_allocated_already)
                delete_memory
        init();
        return *this;
}
```

Disabling Copy

There are several ways of disabling the ability to copy the object that you've created. The usage of asserts and runtime errors is the way we have done things in C, and it is obviously a nasty way to go since the user of the class does not have any control over what happens if an error occurs. As we've seen, ignoring the problem by not defining the assignment operator and the copy constructor will not do us any good as well since default ones are created and will probably do more havoc.

The best way to go is to declare both the operator= and the copy constructor as private, thus getting a *compilation error* each time you attempt to copy in one way or another your object.

Assignment Operator && Inheritance

What happens if we using the assignment operator on two different but inherited types? e.g:

```
class Base {
public:
        int _a;
        float _x;
private:
        int _p;
}

class Derived: public Base {
private:
        int _t;
}
```

```
int main() {
       Base B;
       Derived D;
       B = D;
}
```

In this case, the default copy will simply copy to B all the values of D that were inherited to him by class Base, and leaving all D-specific members alone (since it does not know they exist and only knows the parts of D that it had inherited from Base).

IMPORTANT: Because a copy assignment operator is implicitly declared for
a class if not declared by the user, a base class assignment operator is always hidden by the assignment operator of the derived class, thus you cannot use it directly and rely on the assignment operator that you have declared in the base class but have to implement your own assignment operator fully.

Just like with copy constructor, the copy itself does not touch the virtual tables but only assign the values to the proper variables.

Casting Operators

It is possible to define casting for operators, to enable explicit and implicit casts from the object to any desired type. For example, if we want to assign a casting operator from our Person class so when casted into an integer it will return the ID number we will right

```
opration int() const;


Person::operator int() const {
       return _id;
}
```

And now every time that we want to cast our Person into a double, whether implicitly or explicitly, the result will be the ID number of the instance casted.

Operator Prototypes

Below are some examples for the prototypes of some of the operators that can be overloaded. at the end of this sections is the list of all operators which can be overloaded.

```
//member functions
<ClassName>& operator=(const <ClassName>&);
<ClassName>& operator=(const double&);
const <ClassName> operator-() const;
```

```
<ClassName>& operator+=(const <ClassName>&);
<ClassName>& operator*=(const <ClassName>&);
<ClassName>& operator-=(const <ClassName>&);
<ClassName>& operator/=(const <ClassName>&);

//global functions
friend const <ClassName> operator+ (const <ClassName>&, const <ClassName>&);
friend const <ClassName> operator* (const <ClassName>&, const <ClassName>&);
friend const <ClassName> operator- (const <ClassName>&, const <ClassName>&);
friend const <ClassName> operator/ (const <ClassName>&, const <ClassName>&);
friend bool operator< (const <ClassName>&, const <ClassName>&);
friend bool operator> (const <ClassName>&, const <ClassName>&);
friend bool operator<=(const <ClassName>&, const <ClassName>&);
friend bool operator>=(const <ClassName>&, const <ClassName>&);
friend bool operator==(const <ClassName>&, const <ClassName>&);
friend bool operator!=(const <ClassName>&, const <ClassName>&);
friend std::ostream& operator<<(std::ostream&, const <ClassName>&);
```

Casting && RTTI

General

In C we had a very simple cast system. Either we had automatic casting, such as from int to double, or we had to declare an explicit cast in the form of `(<cast_type>)<var_name>`. This way of casting was extremely unsafe and resulted in tedious run-time errors. In C++ casting is divided into more sections, is more code-specific and much, much more tedious. Implicit casts remain basically the same, and we have two additions: The tailored cast and the fact that we can add our own automatic casts to objects using operators.

Implicit Casts

Implicit casts, or automatic casts are exactly what they sound like. The user has no control over when they happen, nor does he need or want to. This may sound great in theory, but there are some cases where implicit casts can lead to serious issues. The basic implicit casts remain the same as in C, for example, in a statement requiring a double an int may be substituted freely, and similar casts occur automatically. In addition, any derived type can be cast to any of its higher types without stating an explicit casting operation, for example, assuming C inherits from B which inherits from A, the following will work:

```
void doSomething(A& thing) {
        ...some code...
}


int main() {

      C c;

      B b;

      doSomething(b);

      doSomething(c);

      return 0;

}
```

It should be noted, that in the function doSomething, only the methods and members of A can be used, regardless of the actual type that was cast into A on function call. Any virtual methods will still use dynamic resolution and call their relevant methods from class B or C, but any non-virtual methods, even if “overriden” in a lower class will be called from class A.

Tailored Casts

General

Tailored casting is an implicit cast. It's a mechanism of casting a certain type into another by using its constructor. This occurs automatically unless disabled by the user and can sometimes be extremely confusing to debug. Disabling is mentioned later in this section.

Usage

In an expression where type A is expected, the compiler will accept a type B, as long as type A has a ctor which accepts a single B type argument. The result of this will be a call to the ctor of A with the argument of type B and passing the resulting temporal object to the function.

Mistakes

It is possible to create errors by taking advantage of Tailored casting, for example:

```
class IntArray {

public:

      IntArray(int size);

      bool operator==(const IntArray &rhs) const;

      int operator[](int i) const;

      ...additional code...

};
```

Now the code taking advantage of our class:

```
IntArray A(5), B(5);

for (int i = 0; i < 5; i++ )

      if (A == B[i] ) { // Problem !!!

      ...

}

...
```

What is the issue ? The method operator== expects a const reference to an IntArray. B[i] is an int. Normally, such a statement will not compile. However, IntArray has a ctor which accepts a single int parameter. Because of that, the tailored cast mechanism takes over, and the code is actually equivalent to:

```
if (A == IntArray C(B[i]) )...
```

Obviously, this is not what we intended, but this can be very hard to locate and debug, since it compiles and is not immediately obvious.

Disable

Sometimes we might wish to disable the tailored casting to avoid such mistakes as the example above. To declare a certain constructor to be turned off to implicit casting, we simply need to write the reserved word `explicit` right before it, and that constructor will only be enabled when directly called upon.

Explicit Casts

static_cast

Syntax: `static_cast<type>(expression)`

Static casts are used to convert between related types such as one pointer type to another, enumeration to an integral type, or a floating point type to an integral type.
Works where implicit conversions exist, (int -> double, etc'). Failure to cast causes *compiler* error (no dynamic checking is done).

const_cast

Syntax: `const_cast<type>(expression)`

Changes the *const* status of a given variable (add/remove it). Failure to cast causes compiler error. Usually this signifies bad code design since you should design your code so you won't have to use this cast.

reinterpret_cast

Syntax: `reinterpret_cast<type>(expression)`

As the name implies, used to reinterpret byte patterns. Handles conversions between unrelated types such as from integer to a pointer, by changing the number of bytes accessed through the pointer. Mostly used to cast between pointers, e.g

```
reinterpret_cast<char*>(&int_variable)
reinterpret_cast<int*>(void*)
```

The result of the reinterpret_cast is guaranteed to be usable only if its result type is the exact type used to define the value involved.

Important: this cast circumvents the type checking of C++, which makes it extremely dangerous.

dynamic_cast

Syntax: `dynamic_cast<type>(expression)`

Dynamic casts are part of the RTTI (next section), and it enables type-checking during run-time. The cast can accept *only a pointer or a reference type* (Otherwise it won't compile) to a class (and only a class, it does not compile for primitives), and if the casting indeed can be done (upcast or downcast) a valid pointer (if a pointer was cast) or a reference (if a reference was cast) to the expression is returned.

If an error occurs, such as the following code:

```
Base * base = new Base();

Derived * derived = dynamic_cast<Derived *>(base);
```

(The error is because the run-time check can see that the object pointed to by base is not of type Derived) then a NULL pointer is returned or, if we sent a reference, a “bad_cast” exception is thrown.

Non-Polymorphic classes (Without at least a single virtual function), even if inheriting each other, cannot be cast with dynamic_cast. Dynamically casting non-polymorphic types will result in a compilation error.

This is extremely confusing cast, let's view some examples:

```
class Base {
      public:
      virtual int foo() {

            std::cout << "Base::foo()";

      }
};
class Derived : public Base {
      public:
      virtual int foo() {
            std::cout << "Derived::foo()";
      }
};
```

First Attempt:

```
int main() {
      Derived der;
      Base * b = dynamic_cast<Base *>(&der);
      Derived * d = dynamic_cast<Derived *>(b);
      d->foo();
}
```

The above works and does so as intended, because the pointer b is actually pointing to a Derived object, even if its static type is Base. The following example crashes on runtime (The cast fails and d is a NULL), but compiles:

```
int main() {
      Base * b = new Base();

      Derived * d = dynamic_cast<Derived *>(b);
      d->foo();
}
```

The object pointed to by d is a Base object, but we requested a function belonging to the Derived part. The compiler has no way of knowing that cast is not correct – it sees both classes in the same inheritance tree and believes us, allowing the code to compile. During runtime, however, this crashes as the cast returns a NULL pointer.

RTTI

General

RTTI stands for “Run Time Type Information”. RTTI refers to the ability of the system to report on the dynamic type of an object and to provide information about that type at runtime (as opposed to at compile time). It's the C++ version to Java's *instanceof* function and they serve similar purpose. It's used to determine whether down casting is possible by using the *dynamic cast* and the RTTI operators.

The typeid Operator && type_info Class

Obtains information about an object or an expression, like we use the 'sizeof' functions.

Syntax: `typeid(object)`

The result will be an object of the *type_info* class, which is a simple class:

```
class type_info {

public:

      bool operator==(type_info const&) const;

      char const* name() const;

private:

      type_info(type_info const&)

      type_info& operator=(type_info const&)

};
```

name() simply returns the name of the class the object belongs to.

RTTI Misuse

Consider the following code:

```
void rotate(shape const& s) {

      if(typeid(s) == typeid(Circle))

            // Rotate Circle

      else if(typeid(s) == typeid(Triangle))

            // Rotate Triangle

      ...

}
```

This is a classic case of polymorphism, and the code above is extremely unnecessary. Use polymorphism when you can!

An example of good usage for RTTI is when you need to know the type of the object before storing it on the server.

Templates

General

templates are a feature of C++ that allow code to be written without consideration of the data type with which it will eventually be used. Templates support generic programming C++. Templates are a mechanism of the language which receives a type and return non-generic code for the type specified. Usage example, the generic swap function:

```
template <class T> void swap(T& a, T& b) {
      T temp = a;
      a = b;
      b = temp;
}
```

Now, writing:

```
int a = 5, b = 10;
swap(a,b);
```

Will generate the following function, which will be attached to the place where the swap call was made (The object file on compilation). The generated code will be identical to:

```
void swap(int& a, int& b) {

      int temp = a;

      a = b;

      b = temp;

}
```

The Compiler Perspective

When a call to swap(int, int) is made in the code, the compiler looks for a matching function signature. It looks in all scopes known to it and only if it does not find one, it looks for a matching template function (this is called template specialization). If one is found, it then copies the code, changing the templated type to the type used in the specific call and places the generated code in the module where it was called.

Following the previous example of swap, if another call to swap is made with (char, char), or any other type, another swap function will be generated for that type. If two calls for swap with the same type are done in the same scope, the generated code will be reused. In two different modules, however, the same code will be generated twice.

Since the template must be known to the code requesting it, it must be written in the header file. In fact, the entire implementation of the template function must be located in the header file (this is an exception to the separation between forward declaration and implementation of functions).

Templates can receive also constant arguments within the template syntax. For more information, see last section about 'syntax && examples'.

Generic Functions

A generic function is a template which receives a type or a generic argument and returns a function specific for the type given.

Possible uses:

```
template <class T> void printArg(T& elem) {

      std::cout << elem;

}
```

This function, when called to with a concrete type will send the argument received into the standard output stream. If the << operator is not implemented with regard to the relation between cout and the type T specified, a compiler error will result.

This is likely the most common use of a generic function, but there is another:

```
template <int sillyNum> void sillyPrint() {

      std::cout << sillyNum;

}
```

This function must be called to with a special syntax, for example, as follows:

```
sillyPrint<10>();
```

The result would look as if 10 was sent to the standard output stream. There is usually no logic in writing code as the above, as passing the integer as a regular parameter would work just as well and not cause many different sillyPrint codes created for each specific number. You cannot use a variable to define sillyNum, this value must be known at compile time.

O fcourse, combining both is possible and any number of template parameters is possible:

```
template <class T, class S, int x, int y> void sillyMult(T& t, S& s) {

      t = t * x;

      s = s * y;

} // Again, the binary operator * must be implemented for T and S...
```

The syntax to use this template would be:

```
int a = 3;

double b = 4.5;

sillyMult<int,double,3,4>(a,b);
```

Again, this is possible, but as far as we know there is no real reason to use this style.

Generic Classes

Similar to generic C++ functions and also similar to Java's generic classes, we can create a class template. The class template is defined by one (or more) generic types or values as follows:

```
#define DEFAULT_SIZE 10

template <class T, int size = DEFAULT_SIZE> class MyArr {

public:

      ... Insert relevant contents...

private:

      T[size] _myArr;

}
```

The above is a generic class (and a generic type) that implements an array which is an object (A bit like Java). The template depends upon a given type and a given size. If the size is not provided, then it defaults to DEFAULT_SIZE (10). It should be noted that the following two types are completely identical:

```
MyArr<int,10>

MyArr<int>
```

Since the default value will create the equivalent of the first type. Thus if the template was used once with <int,10> then the used with <int> it will not create duplicate code, but <char,10> will create another class from the template.

The type T can be reused in the value field, for example:

```
template <class T, T val> class MyClass {}
```

Template specialization does not work for classes, the following does not compile:

```
template <class T> class MyArr {

public:

      T _myArr[5];
} ;

class MyArr {

public:

      int _myArr[5];

} ;


int main() {

      MyArr test;

      MyArr<int,5> test2;

}
```

Nested Templates

One can use template methods in non-template classes, or template methods within template classes, a nested template class within a non-template class or even use a nested template class inside another template class (Every possible combination of template and non-template applies).

Some general rules:

A nested class can use the template type of its parent without declaring its own:

```
template <class T> class OuterFoo {

public:

      void print();

private:

      class InnerFoo {

      public:

            notFoo();

      private:

            T _elementTwo;

      }

      T _element;

};
```

This essentially creates the InnerFoo as a template “for free”.

It can also be a separate template:

```
template <class T> class OuterFoo {

public:

      void print();

private:

      template <class S> class InnerFoo {

      public:

            void notFoo();

      private:

            S _elementTwo;

            T _elementThree;   // This is also legal !!

      };

      T _element;

};
```

Now, S and T are two completely different templates and can take two different types.

Writing template methods inside a non-template (or a template) class follows the conventions for regular template functions (see above).

The syntax for implementing an inner template class' functions is somewhat different from usual. In the example above, to implement foo we would have to write the following:

```
template <class T> template <class S>

void OuterFoo<T>::InnerFoo<S>::notFoo() {}
```

In the example before that (nested class inside a template class) we would have to write:

```
template <class T> void OuterFoo<T>::InnerFoo::notFoo() {}
```

The typename Keyword

The typename keyword has two uses. First, it can replace the keyword “class” in template definitions. The following are equal:

```
template <class T> class Silly {}

template <typename T> class Silly {}
```

Another use is accessing an inner type of the type T specified in the template, for example:

```
template <typename T> MyClass {

private:

      T::InnerMyClass _y;

}
```

The above code will not compile, since the compiler needs to be specifically told that T is not a simple type, but a more complex one. The correct way to write this is:

```
template <typename T> MyClass {

private:

      typename T::InnerMyClass _y;

}
```

This instructs the compiler that T has a nested type. The compiler believes you at this point, but if you try to write the following, for example:

```
MyClass<int> test;
```

Will result in a compilation error, since int::InnerMyClass does not exist.

Polymorphism Vs. Templates

you can look at templates as compile-time polymorphism, allowing us to generate generic code (in polymorphism we accomplished this via dynamic resolution). So if both can accomplish the same things, which one is better?

Templates can really inflate the code, thus the compile time will be much longer, but since there are no virtual tables and/or dynamic resolution, the running time of the application will be much quicker (combining it with inlining will make the application even faster).

Applying polymorphism to everything will result in an unclear application, code wise and logic wise, so the best way to use polymorphism and inheritance is when logically in the application such things are required (as with Student, Person, etc').

Exceptions

General

The exception mechanism in Cpp works very similar to the exception mechanism in Java. But just as there is no master Object class, there is also has no master Exception class. Each library or program defines its own exception hierarchy, if a hierarchy is approporiate. Another important thing to note is that in Cpp anything and everything can be thrown, not only exceptions. This includes primitives, standard Cpp objects, C style structs, pointers, references and anything else you can imagine. The syntax of try and catch block is identical to Java (but there is no finally clause in Cpp). Finally, unless defined manually, an exception carries no additional information (Unlike Java and its Call Stack description), so it is not easy to know which particular line of code the exception came from.

Example:

```
class Stone {
public:
      void splat();
};


void notFoo() {
      ... do things...
      if(error_occured)
            throw Stone();
}
int main() {
      try {
            notFoo();
      }
      catch (Stone& rock) {
            rock.splat();
      }
}
```

In the above example, if an error occurs in notFoo() a Stone is thrown and caught by the catch clause of the main method.

Note: Exceptions do not support Tailored casting in neither throwing nor catching.

Stack Unwinding

Lets examine the following code:

```
h() {
      throw 10;
}

g() {
      h();
}

f() {
      try {
             g();
      }
      catch (int num) {
            std::cout << num;
      }
}

int main() {
      f();
}
```

Lets examine program flow:

1) The main is started and f() is called.
2) g() is called from the try block in f().
3) g() calls to h().
4) h() throws a primitive of type int with the value of 10.
5) The stack of h() collapses, all local variables (Note: Things on the heap are not freed ! How to smartly deal with such a situation ? (See the smart pointer section, p. 46) inside h() are deleted (their destructors are called) and an int with the value of 10 is created on the stack of g().

   Important: Whether the exception is caught by reference or by value, the compiler still copies the exception object via the copy ctor. This means that an object with a private cpy ctor *cannot be thrown*. Also you need to be careful when throwing objects and make sure that they will not cause any memory problems after being copied.

6) g() does not catch anything, thus the line which called to h() re-throws the value thrown by h(). Again, all local variables inside g() are deleted and the throw value of 10 is now copied to the stack of f().

7) The catch block of f() catches the primitive and prints its value. After this f() terminates normally.

8) main() terminates normally.

Things to look out for: If we wish to throw an object, it needs to implement a copy constructor as it is thrown by value. An alternative is to throw a pointer (throw new MyObject()), but this is ugly and requires to manually delete the object allocated. The correct approach is to throw an object, but catch a reference, like this:

```
throw MyObject();
...
catch (MyObject &gotcha) {}
```

This enables us even a more important feature which is using the *polymorphic attributes* of the thrown objects if those exist.

Exceptions Vs. Other Error Handling Techniques

Unlike in Java, where exceptions were the normal form of error handling, and were thrown in ever increasing numbers, in Cpp we do not use exceptions for everything. In addition, unlike C, the exception mechanism does offer an alternative to return values and global variables such as errno.

Usually, we will use exceptions in the following places:

1) Handling errors in ctors. This is because a ctor has no return value, and thus our other options are limited.

2) Handling memory allocation errors.

3) Mathematical and Logical errors (Such as division by zero).

In comparison to the other error handling methods, exceptions are convenient and allow us to write more coherent code and allows you to handle errors where you would like to. But, they impose additional run-time operations on your program. However, exceptions are an expensive mechanism (run-time wise). If you wish to write error handling for debugging purposes, then assert is definitely a better choice (as it is easier to disable, and takes less time to implement).

Re-throwing An Exception

Sometimes we want to catch an exception in one place, do some things to terminate properly, and then re-throw the exception to continue the stack unwiding. This action can be accomplished easily using the command throw, e.g:

```
void f() {
      throw fubar_exception(__FILE__, __LINE__);
}

void g() {
      try {
            f()
      }
      catch(fubar_exception e) {
            std::cout << "fubar, exiting" << std::endl;
            throw;            // re-throwing the exception
      }
}
```

Exceptions && Polymorphism

What happens when we have:

```
class Base {
public:
      virtual void printErr();
...code...
};

class Derived : public Base {
      virtual void printErr();
...more code...
};

int main() {
      try {
            throw Derived;
      }
      catch (Base gotcha) {
            gotcha.printErr();
      }
```

```
        }
```

Even though we probably meant to get the error message relevant to Derived, we will get the error message of class Base. This is because the catch by value called to the copy ctor of Base, and not Derived. Hence, the Derived portion of the thrown object is not copied by it (See ctors and polymorphism for more information on polymorphism and copying, page 23).

However, if the catch above was done by reference, rather than by value the virtual function would have performed as expected. This is another reason to catch things by reference, rather than by value.

Catch Without Distinction

In addition to polymorphic behavior, there is a way to define a universal catch:

```
        catch (...) {}
```

This catches anything and eveything thrown from the try clause. Just like catch(Exception e) in Java, this is considered as bad form when used without very, very, very good reason.

Disabling && Limiting throws

In general we do not have to declare our function as throwing anything to actually throw things. But, this creates messy code in which it is very hard to know what might be thrown from a given function. It is considered polite to explicitly state in the header file what might be thrown from a function, like this:

```
        void notFoo() throw(int,double, Rock, Stone, Brick);
```

This also limits the person doing the actual implementation to throwing only the declared types.

In order to declare a function which does not throw anything at all (And cannot throw anything, even if we try) we use the syntax:

```
        void notFoo() throw();
```

Constructors, Destructors && Exceptions

If a constructor throws something during its normal execution then it is aborted, all local variables are deleted and the object is not allocated, which means that even if the allocation was an attempt to allocate on the heap, using the 'new' command, there is no need to delete the object after creation failure. However, if the object we were trying to create has pointers allocated by the 'new' command, memory leaks may occur.

Throwing exceptions from a ctor is a good way to notify the user that the creation of the object has failed for some reason. This is especially true because unlike regular functions, we do not have the option of using the return value to indicate an error.

In general, throwing exceptions from a dtor is an extremely bad idea. It is important to note that a destructor is called:

- as part of a normal deallocation (exit from a scope, delete)
- as part of a stack unwinding that handles a previously thrown exception.

In the former case, throwing an exception inside a destructor can simply cause memory leaks due to incorrectly deallocated object (Not all of the normal free operations have completed normally). In the latter, the code must be more clever. If an exception was thrown as part of the stack unwinding caused by another exception, there is no way to choose which exception to handle first. This is interpreted as a failure of the exception handling mechanism and that causes the program to simply crash.

Memory Allocation Exception

If we do not succeed in allocating needed memory in our application via our new command, a "bad_alloc" exception will be cast. In order to deal and catch this exception we need to #include <new>

Iterators

General

In Java, an iterator was a class simply designed to iterate over another class, implementing the Iterator interface. C++ uses a different kind of iterators.

In C++, an iterator is a concept rather than a specific type. The iterators in C++ mimic pointers, overriding the operators, to create an artificial pointer and by that iterate over the object that has them (iterators are declared as nested classes) just like we do with pointers and pointer arithmetic. We can look at a pointer as a primitive iterator.

Iterator Types

Iterators are categorized according to the operators and functions that they support. The most rough categories are:

- Output Iterators – Write only iterators, to view the content of the container.
- Input Iterators – Read only iterators, enable the programmer to insert data into the container

Besides that the iterators have more categories describing their purpose:

- Forward Iterators – Can only go forward in the container. Supports both reading and writing.
- Bidirectional Iterators – Can move backward and forward in the container. Supports both reading and writing.
- Random Access Iterators – Supports random access (just like a pointer). Supports both reading and writing.

In the course we defined the output and input iterators as forward iterators.

Const / Not-const Iterators

Each one of the iterators above (except output iterator) can be declared as a *const iterator*, meaning that it cannot change the values of the container it iterates upon, and all of its operators are declared const, which allows us to iterate over const objects.

It is accustomed to overload the begin() and end() functions, which return iterators to the proper places in the container to be of const and non-const manner, so it will be able to give the best iterator to the calling object.

Required Operation

Each iterator name means that the iterator implemented various operators:

	Output	**Input**	**Forward**	**Bidirectional**	**Random Access**
Read		x = *it	x = *it	x = *it	x = *it
Write	*it = x		*it = x	*it = x	*it = x
Iteration	++	++	++	++ , --	++ , --, + , - , += , -=
Comparison		== , !=	== , !=	== , !=	== , != , < , > , <= , >=

Iterator Validity

Notice that some container which support iterator might have unexpected effects. If we had an iterator to a linked list which points to a node in the list, and we remove that list, the iterator will now point at an uninitialized junk memory, and usage of it will cause unexpected behavior at best. It is important to look at the properties and documentation of the container we are using to see what will happen in such scenarios and adjust iterators of our on creation to such scenarios as well.

Examples: Removal of an element via iterator to *set, list and map* will invalidate the iterator. Removing an element of a *vector* via an iterator will simply cause the iterator to point to an element *after* the one removed.

Smart Pointers && Memory Management

Why?

As we've seen in the case of exceptions, sometimes freeing memory allocated by us can be tedious to impossible. C++ has a mechanism that can do all the dirty work of deallocating all the memory when an exception was thrown, or for one reason or another a pointer reaches the end of its scope. This mechanism is referred to as *smart pointers.* The smart pointers gives us a way to ensure that allocated memory will be freed when its no longer needed.

Ways To Manage Memory

General

One of the hardest, annoying, most tedious and extremely repetitive tasks of a C++ programmer is to be sure that all the memory allocated by the program is smartly and safely freed. There are two main concepts regarding memory management – ownership and references counting. An owner is defined as the object / pointer / being that is responsible for freeing the memory allocated. The ownership concept basically says that at all times, there is an owner, whether known or unknown, fixed or dynamic, that is solely responsible to free the allocated memory. The references counting technique does not establish ownership but uses a different method for solving the problem.

Fixed Ownership

As the name implies, this strategy states that each allocated pointer has one and only one owner (function or object), which is responsible for freeing its memory when it's done. The memory allocated by this pointer cannot be accessed nor modified by any one else. A good example for this kind of strategy is the std::string class, which handles the memory allocated by char*, and encapsulates all the relevant details to it. The user itself cannot handle the char* itself, only via the string class who owns it. The user does not have any access to the pointer itself.

Dynamic Ownership (Compile Time)

This strategy states that the user has only one owner like the fixed ownership, but it can be changed during the program flow. This is basically regular pointer handling since when we return allocated memory we need to remember to free it. The responsibility of freeing the memory is on the programmer.

Dynamic Ownership (Runtime)

General

In some cases we cannot know who will own a pointer, and in some cases we just don't need / care / want to know. Regular pointers cannot help us in figuring out the current owner of the allocated memory, since they remember only the memory address they are currently pointing at. This way of memory management called dynamic ownership, and we resolve it using *smart pointers.*

Smart Pointer

A Smart pointer is a pointer that can do everything that the regular ('dumb') pointer can do, and in addition it can free itself when no one uses it anymore (e.g. when it reaches the end of its scope). Let's see what we need to implement to get such a marvelous pointer:

```
//We use T as a template for anything

pointer::pointer(T*)

pointer& pointer(const pointer&)

pointer& operator=(const pointer&)

T& operator*(const pointer&)

T const& pointer::operator*() const

T* pointer::operator->()

T const* pointer::operator->() const

pointer::operator void const*() const
```

The class will be a wrapper to our original dumb pointer:

```
template <typename T> class pointer {

public:

      pointer(T* p = NULL) : _ptr(p) {}

      T& operator*(const pointer&)

      ...

private:

      T* _ptr;

};
```

Now, for the final touch, we only need to add a destructor that will destruct the memory pointed by the wrapped pointer, which will guarantee that when the smart pointer reaches the end of its scope, it will free the needed memory as well:

```
pointer::~pointer() {

      delete _ptr;

}
```

An implementation of such pointer is available to us in the standard library and is called auto_ptr.

auto_ptr

auto_ptr is essentially a wrapper for any kind of pointer, which automatically deletes the object it points at when it reaches the end of its scope.

Syntax:

```
#include <memory>


auto_ptr<myClass> p(new MyClass());

...

return p.release();
```

If an exception occurs inside the function, auto_ptr is destroyed and it automatically invokes myClass's destructor, freeing it from the memory. If, however, no exception was thrown and everything went smoothly, the *release* command returns the myClass object, and tells the auto_ptr not to delete the object upon stack collapse.

What happens when we assign auto_ptr to another auto_ptr? Since auto_ptr allows only one owner, it simply transfers ownership to the other auto_ptr, and stops being the owner (points to NULL):

```
auto_ptr( auto_ptr& rhs ) : _ptr( rhs.release() ) {}


auto_ptr& operator=( auto_ptr& rhs ) {

      reset( rhs.release() );    // rhs now points to NULL

      return *this;

}


void reset(T* p = NULL)    {

      if( _ptr != p ) delete _ptr;

      _ptr = p;

}


T* release() {

      T* tmp = _ptr;

      _ptr = NULL;

      return tmp;

}
```

auto_ptr provides a smart mechanism for ownership transfer. It does not however, eliminates the need for our loved dumb pointers, and for that we have the method

```
T* auto_ptr<T>::get()
```

which allows us access to the dumb pointer hidden inside the auto_ptr (very dangerous!)

Reference Counting

This memory tracking technique does not point to or specify a particular owner per allocated memory block. Instead, it keeps track of the total number of pointers towards a single memory block. Upon creation of yet another pointer to the same memory block, we simply increment the number of pointers by one, and the opposite takes place on deletion of a pointer. Upon the deletion of the last pointer to the block (the counter reaches zero), the block is deleted.

How do we know that an identical value block is located somewhere in memory ? In short, we simply do not. Instead, the reference counting is embedded into places in which we definitely know that such a memory block exists and we have a way of accessing it (Say, a pointer). Usually this means that the mechanism will be coded into the copy ctor and the operator= for a specific object, but a generic version to count references to any kind of object is possible.

This is all nice and comfortable while we do not try to change the value held within one of the objects. Since they all point to essentially the same place in memory, changing one without additional work automatically changes all the others as well. This is not what we want. Consider the following example:

We have a class SmartString which counts references by embedding the mechanism in its copy ctor and its operator=(). It also implements the operator[] to allow access to the individual characters of the SmartString. Now lets see some code:

```
SmartString a = “snafu”;

SmartString b = a;

SmartString c = b;
```

In effect only a single memory block containing “snafu” exists in memory and all the SmartStrings actually point to it. But what happens when we do:

```
b[2] = 'A';
```

We do not intend to change a,b and c at once. So now we have a problem.

The solution that follows is usually embedded into every method that changes the object (any non const method): If we wish to change the value of the object and the refernce counter is greater than one we:

1)Allocate another section in memory and copy the shared value to it.

2) Decrement the reference counters for the old value by one.

3) Point our object to the new memory and perform the change required.

This is also known as “lazy evaluation”, since time-consuming work is only done when it is really needed.

An example involving strings is below:

```
char const & operator[](int i) const {
   return innerValue->theString[i];
```

```
    }

    char & operator[](int i) {

      if( innerValue->refCounter > 1 ) { // More than just this
                                           // point to the same
                                           // memory

     innerValue->refCounter--;             // Decrement ref counter
                                           // by 1.

    innerValue = new StringValue(*innerValue); // Copy value
                                                  // to a new
                                                  // memory location
      }
      return innerValue->theString[i]; // Return a ref to the newly
                                         // copied string at the
                                         // right index.
    }
  };
```

Can this be done in a generic fashion ? The answer is yes. We can wrap the intended class in a special pointer type that will point to our class, as well as keep a reference counter by adding additional code wrapped around calls to our object's methods.

We should differentiate between const and non-const methods by adding the additional counting operations only to methods which actually change the object we wrap, he is an example (With our class being named RCPointer):

Read only access:

```
T const& RCPointer<T>::operator*() const;
T const* RCPointer<T>::operator->() const;
```

Read and Write access:

```
T& RCPointer<T>::operator*();
T* RCPointer<T>::operator->();
```

Are there cases in which reference counting just does not work ? Yes, in the case of data structures which include cycles. i.e an object which refers directly or indirectly to itself. A mechanism relying purely on reference counts will never consider cyclic chains of objects for deletion, since their reference count is guaranteed to stay nonzero.

Example: Assume that A has a member which is a pointer to B, and B has a member which is a pointer to A. After initialization of A and B, we set to pointers on each, to point to the other object, resulting in a cycle.

When To Use What

auto_ptr / Ownership transfer:

Simple yet problematic, as auto_ptrs are not convenient to work with if passing them around is required. Good for local variables , to ensure they are freed on exceptions.

Reference counting:

Simple, memory safe, quick, problematic when modifying the object. Very good for large objects, and STL containers providing these do not change the objects, since that may change more than a single element .

Reference counting with Clone On Write ("Lazy Evaluation"):

Optimal for STL and for large objects where copying them all the time is extremely inefficient. Problematic for data structures with cycles.

Namespaces

General

Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in "sub-scopes", each one with its own name. Basically namespaces are a mechanism of C++ for logical grouping of declarations and definitions into a common scope, creating an aggregation of related code under one name.

The usage of namespace cause no runtime differences at all, and is used for logic relations alone such in libraries.

Creating Namespaces

To create a namespace, you need to write:

```
namespace <namespace_name> {
      contents
}
```

All the contents inside the braces will be part of the namespace. The contents can be variables, methods, classes, and everything you wish to enter. The namespace has no power like the package in Java, and is only used for organization of data – Libraries, for example, are usually written under a namespace of their own (e.g. std).

You can create several files under the same namespace – the compiler will recognize the same namespace and will compile all of the files under it.

Namespace can be created in the header file and / or the source file.

Using Namespaces

There are several ways to use the contents of the namespace:

1. Using the direct way, e.g. *std::cout* will enter the std namespace and use the cout function inside. This is the safest way since even if we named a function in the same name and / or imported other namespaces containing that function, the compiler will know exactly what we mean.
2. Writing at the beginning of the file

   ```
   using std::cout ;
   ```

   will tell the compiler that when we are writing cout we mean the cout inside the std namespace. This system saves typing and is a little bit more dangerous than the first option since we need to be sure we have no other name duplications in our code.
3. Writing at the beginning of the file

   ```
   using std ;
   ```

   will tell the compiler that every function that we use that exist in the std namespaces, extract it directly from there. That's a very risky method since we do not know the entire library and the possibility for code duplication and cryptic error messages is big.

Important: Do not use other namespaces inside a header file! By doing this you are forcing more elements on the importing user, making the risk for unresolved errors and huge frustration much higher.

Function Objects && Comparators

A function object, or a functor is an object that has an operator (). The STL uses functors in many ways, and an important one is the comparator. The comparator uses the power of a class to compare between two objects inside a container, allowing us to compare our objects as we desire, thus giving even more strength to the template containers inside the STL.

To use the comparator's function call operator() takes two potential elements of the desired class as arguments and returns a Boolean value. The returned value should be true if the first argument must precede the second within the collection class, and false otherwise.

most of the times it is easiest to use one of the functors classes provided by the STL in the header file *<functional>*. In particular, use *less<T>* to maintain elements in increasing order, or *greater<T>* to maintain them in decreasing order.

References

Additional to the lectures, we used the following references:

- “C++ Programming Language”, Bjarne Stroustrup,Third Edition.
- C++ Programming” Wikibook, (http://en.wikibooks.org/wiki/C%2B%2B_Programming)
- www.cplusplus.com
- www.cppreference.com
- Google.

```
<Segmentation Fault. Core Dumped>
```