C++ Programming/Classes

Wikibooks.org

April 17, 2012

Contents

0.1 Classes

Classes are used to create *user defined types*. An instance of a class is called an *object* and programs can contain any number of classes. As with other types, object types are case-sensitive.

Classes provide *encapsulation* as defined in the Object Oriented Programming (OOP) paradigm. A class can have both data members and functions members associated with it. Unlike the built-in types, the class can contain several variables and functions, those are called members.

Classes also provide flexibility in the "DIVIDE AND CONQUER[1]" scheme in program writing. In other words, one programmer can write a class and guarantee an interface. Another programmer can write the main program with that expected interface. The two pieces are put together and compiled for usage.

Note:
From a technical viewpoint, a struct and a class are practically the same thing. A struct can be used anywhere a class can be and vice-versa, the only technical difference is that class members default to *private* and struct members default to *public*. Structs can be made to behave like classes simply by putting in the keyword private at the beginning of the struct. Other than that it is mostly a difference in convention.
The C++ standard does not have a definition for *method*. When discussing with users of other languages, the use of the word *method* to represent a member function can at times become confusing or raise problems to interpretation, like referring to a static member function as a static method. It is even common for some C++ programmers to use the term method to refer specifically to a virtual member functions in an informal context.

0.1.1 Declaration

A class is *defined* by:

1 HTTP://EN.WIKIPEDIA.ORG/WIKI/DIVIDE%20AND%20CONQUER

```
class MyClass
{
 /* public, protected and private
 variables, constants, and functions */
};
```

An object of type *MyClass* (case-sensitive) is *declared* using:

```
MyClass object;
```

- by default, all class members are initially *private*.
- keywords *public* and *protected* allow access to class members.
- classes contain not only data members, but also functions to manipulate that data.
- a class is used as the basic building block of OOP (this is a distinction of convention, not of language-enforced semantics).

A class can be created

- before main() is called.
- when a function is called in which the object is declared.
- when the "new" operator is used.

Class Names

- Name the class after what it is. If you can't determine a name, then you have not designed the system well enough.
- Compound names of over three words are a clue your design may be confusing various entities in your system. Revisit your design. Try a CRC card session to see if your objects have more responsibilities than they should.
- Avoid the temptation of naming a class something similar to the class it is derived from. A class should stand on its own. Declaring an object with a class type doesn't depend on where that class is derived from.
- Suffixes or prefixes are sometimes helpful. For example, if your system uses agents then naming something DownloadAgent conveys real information.

Data Abstraction

A fundamental concept of Object Oriented (OO) recommends an object should not expose any of its implementation details. This way, you can change the implementation without changing the code that uses the object. The class, by design, allows its programmer to hide (and also prevents changes as to) how the class is implemented. This powerful tool allows the programmer to build in a 'preventive' measure. Variables within the class often have a very significant role in what the class does, therefore variables can be secured within the *private* section of the class.

0.1.2 Access labels

The access labels **Public**, **Protected** and **Private** are used within classes to set access permissions for the members in that section of the *class*. All class members are initially *private* by default. The labels can be in any order. These labels can be used multiple times in a class declaration for cases where it is logical to have multiple groups of these types. An access label will remain active until another access label is used to change the permissions.

We have already mentioned that a class can have member functions "inside" it; we will see more about them later. Those member functions can access and modify all the data and member function that are inside the class. Therefore, permission labels are to restrict access to member function that reside outside the class and for other classes.

For example, a class "Bottle" could have a private variable *fill*, indicating a liquid level 0-3 dl. *fill* cannot be modified directly (compiler error), but instead *Bottle* provides the member function sip() to reduce the liquid level by 1. Mywaterbottle could be an instance of that class, an object.

```
/* Bottle - Class and Object Example */
#include <iostream>
#include <iomanip>

using namespace std;

class Bottle
{
  private:      // variables are modified by member functions of class
  int iFill;    // dl of liquid

  public:
    Bottle()    // Default Constructor
    : iFill(3)  // They start with 3 dl of liquid
      {
        // More constructor code would go here if needed.
      }

     bool sip() // return true if liquid was available
     {

       if (iFill > 0)
       {
         --iFill;
         return true;
       }
       else
       {
         return false;
       }

     }

     int level() const  // return level of liquid dl
     {
         return iFill;
     }
 };  // Class declaration has a trailing semicolon

int main()
{
  // terosbottle object is an instance of class Bottle
  Bottle terosbottle;
  cout << "In the beginning, mybottle has "
       << terosbottle.level()
       << "  dl of liquid"
       << endl;

  while (terosbottle.sip())
  {
     cout << "Mybottle has "
          << terosbottle.level()
          << " dl of liquid"
          << endl;
  }

  return 0;
}
```

These keywords, *private, public, and protected,* affect the permissions of the members -- whether functions or variables.

public

This label indicates any members within the 'public' section can accessed freely anywhere a declared object is in scope.

> **Note:**
> Avoid declaring public data members, since doing so would contribute to create unforeseen disasters.

private

Members defined as private are only accessible within the class defining them, or friend classes. Usually the domain of member variables and helper functions. It's often useful to begin putting functions here and then moving them to the higher access levels as needed so to reduce complexity.

> **Note:**
> It's often overlooked that different instances of the same class may access each others' private or protected variables. A common case for this is in copy constructors.

(This is an example where the default copy constructor will do the same thing.)

```
class Foo
{
 public:
   Foo(const Foo &f)
   {
     m_iValue = f.m_iValue; // perfectly legal
   }

 private:
   int m_iValue;
};
```

protected

The protected label has a special meaning to inheritance, protected members are accessible in the class that defines them and in classes that inherit from that base class, or friends of it. In the section on inheritance we will see more about it.

> **Note:**
> Other instances of the same class can access a protected field - provided the two classes are of the same type. However, an instance of a child class cannot access a protected field or method of an instance of a parent class.

0.1.3 Inheritance (Derivation)

As we have seen early as we introduced PROGRAMMING PARADIGMS[2], **INHERITANCE[3]** is a property that describes a relationship between two (or more) types, or classes, of objects in OOP and C++ classes share this property. This in it self in not an abstraction but a characteristic of OOP.

Derivation is the action of creating a new class using the inheritance property of the C++ programming language. It is possible to derive one class from another or even several (**MULTIPLE INHERITANCE[4]**), like a tree we can call base class to the root and child class to any leaf; in any other case the parent/child relation will exist for each class derived from another.

Base Class

A base class is a class that is created with the intention of deriving other classes from it.

Child Class

A child class is a class that was derived from another, that will now be the parent class to it.

Parent Class

A parent class is the closest class that we derived from to create the one we are referencing as the child class.

As an example, suppose you are creating a game, something using different cars, and you need specific type of car for the policemen and another type for the player(s). Both car types share similar properties. The major difference (on this example case) would be that the policemen type would have sirens on top of their cars and the players' cars will not.

2 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FPROGRAMMING%20LANGUAGES%2FPARADIGMS
3 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FPROGRAMMING%20LANGUAGES%2FPARADIGMS%2FINHERITANCE
4 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FCLASSES%2FINHERITANCE%23MULTIPLE_INHERITANCE

One way of getting the cars for the policemen and the player ready is to create separate classes for policemen's car and for the player's car like this:

```
class PlayerCar {
private:
  int color;

public:
  void driveAtFullSpeed(int mph){
    // code for moving the car ahead
  }

};

class PoliceCar {
private:
  int color;
  bool sirenOn;  // identifies whether the siren is on or not
  bool inAction; // identifies whether the police is in action (following the
 player) or not

public:
  bool isInAction(){
    return this->inAction;
  }

  void driveAtFullSpeed(int mph){
    // code for moving the car ahead
  }

};
```

and then creating separate objects for the two cars like this:

```
PlayerCar player1;
PoliceCar policemen1;
```

So, except for one thing that you can easily notice: there are certain parts of code that are very similar (if not exactly the same) in the above two classes. In essence, you have to type in the same code at two different locations! And when you update your code to include methods (functions) for `handBrake()` and `pressHorn()`, you'll have to do that in both the classes above.

Therefore, to escape this frustrating (and confusing) task of writing the same code at multiple locations in a single project, you use Inheritance.

Now that you know what kind of problems Inheritance solves in C++, let's examine how to implement Inheritance in our programs. As its name suggests, Inheritance lets us create new classes which automatically have all the code from existing classes. It means that if there is a class called `MyClass`, a new class with the name `MyNewClass` can be created which will have all the code present inside the `MyClass` class. The following code segment shows it all:

```
class MyClass {
  protected:
        int age;
  public:
        void sayAge(){
            this->age = 20;
            cout << age;
        }
};
```

```
class MyNewClass : public MyClass {

};

int main() {

  MyNewClass *a = new MyNewClass();
  a->sayAge();

  return 0;

}
```

As you can see, using the colon ':' we can inherit a new class out of an existing one. It's that simple! All the code inside the `MyClass` class is now available to the `MyNewClass` class. And if you are intelligent enough, you can already see the advantages it provides. If you are like me (i.e. not too intelligent), you can see the following code segment to know what I mean:

```
class Car {
  protected:
        int color;
        int currentSpeed;
        int maxSpeed;
  public:
        void applyHandBrake(){
            this->currentSpeed = 0;
        }
        void pressHorn(){
            cout << "Teeeeeeeeeeeeent"; // funny noise for a horn
        }
        void driveAtFullSpeed(int mph){
             // code for moving the car ahead;
        }
};

class PlayerCar : public Car {

};

class PoliceCar : public Car {
  private:
        bool sirenOn;  // identifies whether the siren is on or not
        bool inAction; // identifies whether the police is in action (following
 the player) or not
  public:
        bool isInAction(){
            return this->inAction;
        }
};
```

In the code above, the two newly created classes *PlayerCar* and *PoliceCar* have been inherited from the *Car* class. Therefore, all the methods and properties (variables) from the *Car* class are available to the newly created classes for the player's car and the policemen's car. Technically speaking, in C++, the *Car* class in this case is our "Base Class" since this is the class which the other two classes are based on (or inherit from).

Just one more thing to note here is the keyword *protected* instead of the usual *private* keyword. That's no big deal: We use *protected* when we want to make sure that the variables we define in our base class should be available in the classes that inherit from that base class. If you use *private* in the class definition of the *Car* class, you will not be able to inherit those variables inside your inherited classes.

There are three types of class inheritance: public, private and protected. We use the keyword *public* to implement public inheritance. The classes who inherit with the keyword public from a base class, inherit all the public members as public members, the protected data is inherited as protected data and the private data is inherited but it cannot be accessed directly by the class.

The following example shows the class Circle that inherits "publicly" from the base class Form:

```
class Form {
private:
  double area;

public:
  int color;

  double getArea(){
    return this->area;
  }

  void setArea(double area){
    this->area=area;
  }

};

class Circle: public Form {
public:
  double getRatio() {
    double a;
    a= getArea();
    return sqrt(a/2*3.14);
  }

  void setRatio(double diameter) {
    setArea( pow( diameter * 0.5, 2) * (3.14));
  }

  bool isDark() {
    return color>10;
  }

};
```

The new class Circle inherits the attribute area from the base class Form (the attribute area is implicitly an attribute of the class Circle), but it cannot access it directly. It does so through the functions getArea and setArea (that are public in the base class and remain public in the derived class). The color attribute, however, is inherited as a public attribute, and the class can access it directly.

The following table indicates how the attributes are inherited in the three different types of inheritance:

	private	protected	public
private inheritance	The member is inaccessible.	The member is private.	The member is private.
protected inheritance	The member is inaccessible.	The member is protected.	The member is protected.
public inheritance	The member is inaccessible.	The member is protected.	The member is public.

As the table above shows, protected members are inherited as protected methods in public inheritance. Therefore, we should use the protected label whenever we want to declare a method inaccessible outside the class and not to lose access to it in derived classes. However, losing accessibility can be useful sometimes, because we are encapsulating details in the base class.

Let's imagine that we have a class with a very complex method "m" that invokes many auxiliary methods declared as private in the class. If we derive a class from it, we should not bother about those methods because they are inaccessible in the derived class. If a different programmer is in charge of the design of the derived class, allowing access to those methods could be the cause of errors and confusion. So, it is a good idea to avoid the protected label whenever we can have a design with the same result with the private label.

Multiple inheritance

MULTIPLE INHERITANCE[5] allows the construction of classes that inherit from more than one type or class. This contrasts with single inheritance, where a class will only inherit from one type or class.

Multiple inheritance can cause some confusing situations, and is much more complex than single inheritance, so there is some debate over whether or not its benefits outweigh its risks. Multiple inheritance has been a touchy issue for many years, with opponents pointing to its increased complexity and ambiguity in situations such as the "DIAMOND PROBLEM[6]". Most modern OOP languages do not allow multiple inheritance.

The declared order of derivation is relevant for determining the order of default initialization by constructors and destructors cleanup.

```
class One
{
  // class internals
}

class Two
{
  // class internals
}

class MultipleInheritance : public One, public Two
{
  // class internals
}
```

Note:
Remember that when creating classes that will be derived from, the destructor may require further considerations.

EXPANDTEMPLATES[7]

5 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FPROGRAMMING%20LANGUAGES%2FPARADIGMS%2FINHERITANCE%2FMULTIPLE%20INHERITANCE
6 HTTP://EN.WIKIPEDIA.ORG/WIKI/DIAMOND%20PROBLEM
7 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

0.1.4 Data members

Data members are declared in the same way as a global or function variable, but as part of the class definition. Their purpose is to store information for that class and may include members of any type, even other user-defined types. They are usually hidden from outside use, depending on the coding style adopted, external use is normally done through SPECIAL MEMBER FUNCTIONS[8].

Note:
Explicit initializers are not allowed inside the class definition, except if they are `const static int` or enumeration types, these may have an explicit initializer.

To do:
Add more info

this pointer

The *this* keyword acts as a pointer to the class being referenced. The *this* pointer acts like any other pointer, although you can't change the pointer itself. Read the section concerning POINTERS AND REFERENCES[9] to understand more about general pointers.

The *this* pointer is only accessible within nonstatic member functions of a **class**, **union** or **struct**, and is not available in static member functions. It is not necessary to write code for the *this* pointer as the compiler does this implicitly. When using a debugger, you can see the *this* pointer in some variable list when the program steps into nonstatic class functions.

In the following example, the compiler inserts an implicit parameter *this* in the nonstatic member function int getData(). Additionally, the code initiating the call passes an implicit parameter (provided by the compiler).

```
class Foo
{
private:
    int iX;
public:
    Foo(){ iX = 5; };

    int getData()
    {
        return this->iX;  // this is provided by the compiler at compile time
    }
};
```

8 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2FClasses%23Accessors_and_Modifiers_.28Setter.2FGetter.29
9 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2FProgramming%20Languages%2FC%2B%2B%2FCode%2FStatements%2FVariables%2FOperators%23Pointers.2C%20.22.2A.22%20and%20References.2C%20.22.26.22

```
int main()
{
    Foo Example;
    int iTemp;

    iTemp = Example.getData(&Example);  // compiler adds the &Example reference
 at compile time

    return 0;
}
```

There are certain times when a programmer should know about and use the *this* pointer. The *this* pointer should be used when overloading the assignment operator to prevent a catastrophe. For example, add in an assignment operator to the code above.

```
class Foo
{
private:
    int iX;
public:
    Foo() { iX = 5; };

    int getData()
    {
        return iX;
    }

    Foo& operator=(const Foo &RHS);
};

Foo& Foo::operator=(const Foo &RHS)
{
    if(this != &RHS)
    {    // the if this test prevents an object from copying to itself (ie. RHS =
 RHS;)
        this->iX = RHS.iX;     // this is suitable for this class, but can be
 more complex when
                               // copying an object in a different much larger
 class
    }

    return (*this);            // returning an object allows chaining, like a = b
 = c; statements
}
```

However little you may know about *this*, it is important in implementing any class.

EXPANDTEMPLATES[10]

static data member

The use of the `static`[11] specifier in a data member, will cause that member to be shared by all instances of the owner class and derived classes. To use static data members you must declare the data member as static and initialize it outside of the class declaration, at file scope.

When used in a class data member, all instantiations of that class share one copy of the variable.

10 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3ASPECIAL%3AEXPANDTEMPLATES
11 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FPROGRAMMING%20LANGUAGES%2FC%2B%2B%2FCODE%2FKEYWORDS%2FSTATIC

```
class Foo {
public:
  Foo() {
    ++iNumFoos;
    cout << "We have now created " << iNumFoos << " instances of the Foo
 class\n";
  }
private:
  static int iNumFoos;
};

int Foo::iNumFoos = 0;  // allocate memory for numFoos, and initialize it

int main() {
  Foo f1;
  Foo f2;
  Foo f3;
}
```

In the example above, the static class variable numFoos is shared between all three instances of the *Foo* class (*f1*, *f2* and *f3*) and keeps a count of the number of times that the Foo class has been instantiated.

12

0.1.5 Member Functions

Member functions can (and should) be used to interact with data contained within user defined types. User defined types provide flexibility in the "DIVIDE AND CONQUER[13]" scheme in program writing. In other words, one programmer can write a user defined type and guarantee an interface. Another programmer can write the main program with that expected interface. The two pieces are put together and compiled for usage. User defined types provide *encapsulation* defined in the Object Oriented Programming (OOP) paradigm.

Within classes, to protect the data members, the programmer can define functions to perform the operations on those data members. Member functions and functions are names used interchangeably in reference to classes. Function prototypes are declared within the class definition. These prototypes can take the form of non-class functions as well as class suitable prototypes. Functions can be declared and defined within the class definition. However, most functions can have very large definitions and make the class very unreadable. Therefore it is possible to define the function outside of the class definition using the scope resolution operator "**::**". This scope resolution operator allows a programmer to define the functions somewhere else. This can allow the programmer to provide a header file *.h* defining the class and a *.obj* file built from the compiled *.cpp* file which contains the function definitions. This can hide the implementation and prevent tampering. The user would have to define every function again to change the implementation. Functions within classes can access and modify (unless the function is constant) data members without declaring them, because the data members are already declared in the class.

Simple example:

file: Foo.h

12 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3ASPECIAL%3AEXPANDTEMPLATES
13 HTTP://EN.WIKIPEDIA.ORG/WIKI/DIVIDE%20AND%20CONQUER

```
// the header file named the same as the class helps locate classes within a
 project
// one class per header file makes it easier to keep the
// header file readable (some classes can become large)
// each programmer should determine what style works for them or what programming
 standards their
// teacher/professor/employer has

#ifndef FOO_H
#define FOO_H

class Foo{
public:
  Foo();                 // function called the default constructor
  Foo( int a, int b );   // function called the overloaded constructor
  int Manipulate( int g, int h );

private:
  int x;
  int y;
};

#endif
```

file: Foo.cpp

```
#include "Foo.h"

/* these constructors should really show use of initialization lists
Foo::Foo() : x(5), y(10)
{
}
Foo:Foo(int a, int b) : x(a), y(b)
{
}
*/
Foo::Foo(){
  x = 5;
  y = 10;
}
Foo::Foo( int a, int b ){
  x = a;
  y = b;
}

int Foo::Manipulate( int g, int h ){
  x = h + g*x;
  y = g + h*y;
}
```

Overloading

Member functions can be overloaded. This means that multiple member functions can exist with the same name on the same scope, but must have different signatures. A member function's signature is comprised of the member function's name and the type and order of the member function's parameters.

Due to name hiding, if a member in the derived class shares the same name with members of the base class, they will be hidden to the compiler. To make those members visible, one can use declarations to introduce them from base class scopes.

Constructors and other class member functions, except the Destructor, can be overloaded.

Constructors

A constructor is a special member function which is called whenever a new instance of a class is created. The compiler calls the constructor after the new object has been allocated in memory, and converts that "raw" memory into a proper, typed object. The constructor is declared much like a normal member function but it will share the name of the class and it has no return value.

Constructors are responsible for almost all of the run-time setup necessary for the class operation. Its main purpose becomes in general defining the data members upon object instantiation (when an object is declared), they can also have arguments, if the programmer so chooses. If a constructor has arguments, then they should also be added to the declaration of any other object of that class when using the **new** operator. Constructors can also be overloaded.

```
Foo myTest;                // essentially what happens is:  Foo myTest = Foo();
Foo myTest( 3, 54 );       // accessing the overloaded constructor
Foo myTest = Foo( 20, 45 ); // although a new object is created, there are some
 extra function calls involved
                           // with more complex classes, an assignment operator
 should
                           // be defined to ensure a proper copy (includes
 ''deep copy'')
                           // myTest would be constructed with the default
 constructor, and then the
                           // assignment operator copies the unnamed Foo( 20, 45
 ) object to myTest
```

using **new** with a constructor

```
Foo* myTest = new Foo();          // this defines a pointer to a dynamically
 allocated object
Foo* myTest = new Foo( 40, 34 );   // constructed with Foo( 40, 34 )
// be sure to use delete to avoid memory leaks
```

Note:
While there is no risk in using **new** to create an object, it is often best to avoid using memory allocation functions within objects' constructors. Specifically, using new to create an array of objects, each of which also uses new to allocate memory during its construction, often results in runtime errors. If a class or structure contains members which must be pointed at dynamically created objects, it is best to sequentially initialize these arrays of the parent object, rather than leaving the task to their constructors.
This is especially important when writing code with exceptions (in EXCEPTION HANDLING[14]), if an exception is thrown before a constructor is completed, the associated destructor will not be called for that object.

A constructor can't delegate to another. It is also considered desirable to reduce the use of default arguments, if a maintainer has to write and maintain multiple constructors it can result in code

14 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FEXCEPTION%20HANDLING

duplication, which reduces maintainability because of the potential for introducing inconsistencies and even lead to code bloat.

Default Constructors

A default constructor is one which can be called with no arguments. Most commonly, a default constructor is declared without any parameters, but it is also possible for a constructor with parameters to be a default constructor if all of those parameters are given default values.

In order to create an array of objects of a class type, the class must have an accessible default constructor; C++ has no syntax to specify constructor arguments for array elements.

Overloaded Constructors

When an object of a class is instantiated, the class writer can provide various constructors each with a different purpose. A large class would have many data members, some of which may or may not be defined when an object is instantiated. Anyway, each project will vary, so a programmer should investigate various possibilities when providing constructors.

These are all constructors for a class myFoo.

```
myFoo(); // default constructor, the user has no control over initial values
        // overloaded constructors

myFoo( int a, int b=0 ); // allows construction with a certain 'a' value, but
 accepts 'b' as 0
                     // or allows the user to provide both 'a' and 'b' values
 // or

myFoo( int a, int b ); // overloaded constructor, the user must specify both
 values

class myFoo {
private:
  int Useful1;
  int Useful2;

public:
  myFoo(){                   // default constructor
          Useful1 = 5;
          Useful2 = 10;
         };

  myFoo( int a, int b = 0 ) { // two possible cases when invoked
        Useful1 = a;
        Useful2 = b;
  };

};

myFoo Find;          // default constructor, private member values Useful1 = 5,
 Useful2 = 10
myFoo Find( 8 );     // overloaded constructor case 1, private member values
 Useful1 = 8, Useful2 = 0
myFoo Find( 8, 256 ); // overloaded constructor case 2, private member values
 Useful1 = 8, Useful2 = 256
```

Constructor initialization lists

Constructor initialization lists (or member initialization list) are the only way to initialize data members and base classes with a non-default constructor. Constructors for the members are included between the argument list and the body of the constructor (separated from the argument list by a colon). Using the initialization lists is not only better in terms of efficiency but also the simplest way to guarantee that all initialization of data members are done before entering the body of constructors.

```
// Using the initialization list for _myComplexMember
MyClass::MyClass(int mySimpleMember, MyComplexClass myComplexMember)
: _myComplexMember(myComplexMember) // only 1 call, to the copy constructor
{
 _mySimpleMember=mySimpleMember; // uses 2 calls, one for the constructor of the
 mySimpleMember class
                                 // and a second for the assignment operator of
 the MyComplexClass class
}
```

This is more efficient than assigning value to the complex data member inside the body of the constructor because in that case the variable is initialized with its corresponding constructor.

Note that the arguments provided to the constructors of the members do not need to be arguments to the constructor of the class; they can also be constants. Therefore you can create a default constructor for a class containing a member with no default constructor.

Example:

```
MyClass::MyClass() : _myComplexMember(0) { }
```

It is useful to initialize your members in the constructor using this initialization lists. This makes it obvious for the reader that the constructor does not execute logic. The order the initialization is done should be the same as you defined your base-classes and members. Otherwise you can get warnings at compile-time. Once you start initializing your members make sure to keep all in the constructor(s) to avoid confusion and possible 0xbaadfood.

It is safe to use constructor parameters that are named like members.

Example:

```
class MyClass : public MyBaseClassA, public MyBaseClassB {
  private:
    int c;
    void *pointerMember;
  public:
    MyClass(int,int,int);
};
/*...*/
MyClass::MyClass(int a, int b, int c):
 MyBaseClassA(a)
,MyBaseClassB(b)
,c(c)
,pointerMember(NULL)
,referenceMember()
{
 //logic
}
```

Note that this technique was also possible for normal functions but it is now obsoleted and is classified as an error in such case.

Note:
It is a common misunderstanding that initialization of data members can be done within the body of constructors. All such kind of so-called "initialization" are actually assignments. The C++ standard defines that all initialization of data members are done before entering the body of constructors. This is the reason why certain types (const types and references) cannot be assigned to and must be initialized in the constructor initialization list.
One should also keep in mind that class members are initialized in the order they are declared, not the order they appear in the initializer list. One way of avoiding CHICKEN AND EGG PARADOXES[15] is to always add the members to the initializer list in the same order they're declared.

Destructors

Destructors like the Constructors are declared as any normal member functions but will share the same name as the Class, what distinguishes them is that the Destructor's name is preceded with a "~", it can not have arguments and can't be overloaded.

Destructors are called whenever an Object of the Class is destroyed. Destructors are crucial in avoiding resource leaks (by deallocating memory), and in implementing the RAII idiom. Resources which are allocated in a Constructor of a Class are usually released in the Destructor of that Class as to return the system to some known or stable state after the Class ceases to exist.

The Destructor is invoked when Objects are destroyed, after the function they were declared in returns, when the **delete** operator is used or when the program is over. If an object of a derived type is destructed, first the Destructor of the most derived object is executed. Then member objects and base class subjects are destructed recursively, in the reverse order their corresponding Constructors completed. As with structs the compiler implicitly-declares a Destructor as a inline public member of its class if the class doesn't have a user-declared Destructor.

The DYNAMIC TYPE[16] of the object will change from the most derived type as Destructors run, symmetrically to how it changes as Constructors execute. This affects the functions called by virtual calls during construction and destruction, and leads to the common (and reasonable) advice to avoid calling virtual functions of an object either directly or indirectly from its Constructors or Destructors.

`inline`[17]

Sharing most of the concepts we have seen before on the introduction to INLINE FUNCTIONS[18], when dealing with member function those concepts are extended, with a few additional considerations.

15 HTTP://EN.WIKIPEDIA.ORG/WIKI/CHICKEN%20OR%20THE%20EGG
16 HTTP://EN.WIKIPEDIA.ORG/WIKI/DYNAMIC%20TYPE
17 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FPROGRAMMING%20LANGUAGES%2FC%2B%2B%2FCODE%2FKEYWORDS%2FINLINE
18 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FCODE%2FSTATEMENTS%2FFUNCTIONS%23INLINE

If the member functions definition is included inside the declaration of the class, that function is by default made implicitly inline. Compiler options may override this behavior.

Calls to virtual functions cannot be inlined if the object's type is not known at compile-time, because we don't know which function to inline.

`static`[19]

The **static** keyword can be used in four different ways:

- TO CREATE PERMANENT STORAGE FOR LOCAL VARIABLES IN A FUNCTION[20].
- TO SPECIFY INTERNAL LINKAGE[21].
- TO DECLARE MEMBER FUNCTIONS THAT ACT LIKE NON-MEMBER FUNCTIONS[22].
- TO CREATE A SINGLE COPY OF A DATA MEMBER[23].

To do:
Alter the above links from subsection to book locations after the structure is fixed.

static member function
Member functions or variables declared static are shared between all instances of an object type. Meaning that only one copy of the member function or variable does exists for any object type.

member functions callable without an object

When used in a class function member, the function does not take an instantiation as an implicit `this`[24] parameter, instead behaving like a free function. This means that static class functions can be called without creating instances of the class:

```
class Foo {
public:
  Foo() {
    ++numFoos;
    cout << "We have now created " << numFoos << " instances of the Foo class\n";
```

19 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FPROGRAMMING%20LANGUAGES%2FC%2B%2B%2FCODE%2FKEYWORDS%2FSTATIC
20 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FPROGRAMMING%20LANGUAGES%2FC%2B%2B%2FCODE%2FKEYWORDS%2FSTATIC%2FPERMANENT%20STORAGE
21 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FPROGRAMMING%20LANGUAGES%2FC%2B%2B%2FCODE%2FKEYWORDS%2FSTATIC%2FINTERNAL%20LINKAGE
22 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FPROGRAMMING%20LANGUAGES%2FC%2B%2B%2FCODE%2FKEYWORDS%2FSTATIC%2FMEMBER%20FUNCTION
23 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FPROGRAMMING%20LANGUAGES%2FC%2B%2B%2FCODE%2FKEYWORDS%2FSTATIC%2FDATA%20MEMBER
24 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FPROGRAMMING%20LANGUAGES%2FC%2B%2B%2FCODE%2FKEYWORDS%2FTHIS

```
  }
  static int getNumFoos() {
    return numFoos;
  }
private:
  static int numFoos;
};

int Foo::numFoos = 0;  // allocate memory for numFoos, and initialize it

int main() {
  Foo f1;
  Foo f2;
  Foo f3;
  cout << "So far, we've made " << Foo::getNumFoos() << " instances of the Foo
 class\n";
}
```

Named constructors

Named constructors are a good example of using static member functions. *Named constructors* is the name given to functions used to create an object of a class without (directly) using its constructors. This might be used for the following:

1. To circumvent the restriction that constructors can be overloaded only if their signatures differ.
2. Making the class non-inheritable by making the constructors private.
3. Preventing stack allocation by making constructors private

Declare a static member function that uses a private constructor to create the object and return it. (It could also return a pointer or a reference but this complication seems useless, and turns this into the FACTORY PATTERN[25] rather than a conventional named constructor.)

Here's an example for a class that stores a temperature that can be specified in any of the different temperature scales.

```
class Temperature
{
    public:
        static Temperature Fahrenheit (double f);
        static Temperature Celsius (double c);
        static Temperature Kelvin (double k);
    private:
        Temperature (double temp);
        double _temp;
};

Temperature::Temperature (double temp):_temp (temp) {}

Temperature Temperature::Fahrenheit (double f)
{
    return Temperature ((f + 459.67) / 1.8);
}

Temperature Temperature::Celsius (double c)
{
    return Temperature (c + 273.15);
}
```

25 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FCODE%2FDESIGN%20PATTERNS

```
Temperature Temperature::Kelvin (double k)
{
   return Temperature (k);
}
```

26

const

This type of member function cannot modify the member variables of a class. It's a hint both to the programmer and the compiler that a given member function doesn't change the internal state of a class; however, any variables declared as `mutable` can still be modified.

Take for example:

```
class Foo
{
 public:
   int value() const
   {
     return m_value;
   }

   void setValue( int i )
   {
     m_value = i;
   }

 private:
   int m_value;
};
```

Here `value()` clearly does not change m_value and as such can and should be const. However `setValue()` does modify m_value and as such cannot be const.

Another subtlety often missed is a `const` member function cannot call a non-const member function (and the compiler will complain if you try). The `const` member function cannot change member variables and a non-const member functions can change member variables. Since we assume non-const member functions do change member variables, `const` member functions are assumed to never change member variables and can't call functions that do change member variables.

The following code example explains what `const` can do depending on where it is placed.

```
class Foo
{
public:
   /*
     * Modifies m_widget and the user
     * may modify the returned widget.
     */
   Widget *widget();

   /*
     * Does not modify m_widget but the
     * user may modify the returned widget.
     */
```

26 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3ASPECIAL%3AEXPANDTEMPLATES

```
    Widget *widget() const;

    /*
      * Modifies m_widget, but the user
      * may not modify the returned widget.
      */
    const Widget *cWidget();

    /*
      * Does not modify m_widget and the user
      * may not modify the returned widget.
      */
    const Widget *cWidget() const;

private:
    Widget *m_widget;
};
```

Accessors and Modifiers (Setter/Getter)

What is an accessor?

An accessor is a member function that does not modify the state of an object. The accessor functions should be declared as CONST[27].

Getter is another common definition of an accessor due to the naming (`GetSize()`) of that type of member functions.

What is a modifier?

A modifier, also called a modifying function, is a member function that changes the value of at least one data member. In other words, an operation that modifies the state of an object. Modifiers are also known as 'mutators'.

Setter is another common definition of a modifier due to the naming (`SetSize( int a_Size )`) of that type of member functions.

Note:
These are commonly used reference labels (not defined on the standard language).

Dynamic polymorphism (Overrides)

So far, we have learned that we can add new data and functions to a class through inheritance. But what about if we want our derived class to inherit a method from the base class, but to have a different implementation for it? That is when we are talking about polymorphism, a fundamental concept in OOP programming.

27 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FCLASSES%2FMEMBER%20FUNCTIONS%23CONST

As seen previously in the PROGRAMMING PARADIGMS SECTION[28], POLYMORPHISM[29] is subdivided in two concepts *static polymorphism* and *dynamic polymorphism*. This section concentrates on dynamic polymorphism, which applies in C++ when a derived class overrides a function declared in a base class.

We implement this concept redefining the method in the derived class. However, we need to have some considerations when we do this, so now we must introduce the concepts of dynamic binding, static binding and virtual methods.

Suppose that we have two classes, `A` and `B`. `B` derives from `A` and redefines the implementation of a method `c()` that resides in class `A`. Now suppose that we have an object `b` of class `B`. How should the instruction `b.c()` be interpreted?

If `b` is declared in the stack (not declared as a pointer or a reference) the compiler applies static binding, this means it interprets (at compile time) that we refer to the implementation of `c()` that resides in `B`.

However, if we declare `b` as a pointer or a reference of class `A`, the compiler could not know which method to call at compile time, because `b` can be of type `A` or `B`. If this is resolved at run time, the method that resides in `B` will be called. This is called dynamic binding. If this is resolved at compile time, the method that resides in A will be called. This is again, static binding.

Virtual member functions

The `virtual` member functions is relatively simple, but often misunderstood. The concept is an essential part of designing a class hierarchy in regards to sub-classing classes as it determines the behavior of overridden methods in certain contexts.

Virtual member functions are class member functions, that can be overridden in any class derived from the one where they were declared. The member function body is then replaced with a new set of implementation in the derived class.

Note:
When overriding virtual functions you can alter the private, protected or public state access state of the member function of the derived class.

By placing the keyword `virtual` before a method declaration we are indicating that when the compiler has to decide between applying static binding or dynamic binding it will apply dynamic binding. Otherwise, static binding will be applied.

Note:
While it is not required to use the virtual keyword in our subclass definitions (since if the base class function is virtual all subclass overrides of it will also be virtual) it is good style to do so when producing code for future reutilization (for use outside of the same project).

Again, this should be clearer with an example:

28 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FPROGRAMMING%20PARADIGMS
29 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FPROGRAMMING%20LANGUAGES%2FPARADIGMS%2FPOLYMORPHISM

```
class Foo
{
public:
  void f()
  {
    std::cout << "Foo::f()" << std::endl;
  }
  virtual void g()
  {
    std::cout << "Foo::g()" << std::endl;
  }
};

class Bar : public Foo
{
public:
  void f()
  {
    std::cout << "Bar::f()" << std::endl;
  }
  virtual void g()
  {
    std::cout << "Bar::g()" << std::endl;
  }
};

int main()
{
  Foo foo;
  Bar bar;

  Foo *baz = &bar;
  Bar *quux = &bar;

  foo.f(); // "Foo::f()"
  foo.g(); // "Foo::g()"

  bar.f(); // "Bar::f()"
  bar.g(); // "Bar::g()"

  // So far everything we would expect...

  baz->f();  // "Foo::f()"
  baz->g();  // "Bar::g()"

  quux->f(); // "Bar::f()"
  quux->g(); // "Bar::g()"

  return 0;
}
```

Our first calls to `f()` and `g()` on the two objects are straightforward. However things get interesting with our baz pointer which is a pointer to the Foo type.

`f()` is not `virtual` and as such a call to `f()` will always invoke the implementation associated with the pointer type -- in this case the implementation from Foo.

Note:
Remember that OVERLOADING[30] and OVERRIDING[31] are distinct concepts.

30 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FCLASSES%2FMEMBER%20FUNCTIONS%23OVERLOADING
31 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FCLASSES%2FPOLYMORPHISM

Virtual function calls are computationally more expensive than regular function calls. Virtual functions use pointer indirection, invocation and will require a few extra instructions than normal member functions. They also require that the constructor of any class/structure containing virtual functions to initialize a table of pointers to its virtual member functions.

All this characteristics will signify a trade-off between performance and design. One should avoid preemptively declaring functions virtual without an existing structural need. Keep in mind that virtual functions that are only resolved at run-time cannot be inlined.

To do:
Example of issue of virtual and inline.

Note:
Some of the needs for using virtual functions can be addressed by using a class template. This will be covered when we introduce TEMPLATES[32].

Pure virtual member function

There is one additional interesting possibility. Sometimes we don't want to provide an implementation of our function at all, but want to require people sub-classing our class to provide an implementation on their own. This is the case for *pure* virtuals.

To indicate a pure `virtual` function instead of an implementation we simply add an "`= 0`" after the function declaration.

Again -- an example:

```
class Widget
{
public:
   virtual void paint() = 0;
};

class Button : public Widget
{
public:
   void paint() // is virtual because it is an override
   {
      // do some stuff to draw a button
   }
};
```

Because `paint()` is a pure `virtual` function in the Widget `class` we are required to provide an implementation in all concrete subclasses. If we don't the compiler will give us an error at build time.

This is helpful for providing interfaces -- things that we expect from all of the objects based on a certain hierarchy, but when we want to ignore the implementation details.

32 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FTEMPLATES

So why is this useful?

Let's take our example from above where we had a pure `virtual` for painting. There are a lot of cases where we want to be able to do things with widgets without worrying about what kind of widget it is. Painting is an easy example.

Imagine that we have something in our application that repaints widgets when they become active. It would just work with pointers to widgets -- i.e. `Widget *activeWidget() const` might be a possible function signature. So we might do something like:

```
Widget *w = window->activeWidget();
w->paint();
```

We want to actually call the appropriate paint member function for the "real" widget type -- not `Widget::paint()` (which is a "pure" `virtual` and will cause the program to crash if called using virtual dispatch). By using a `virtual` function we insure that the member function implementation for our subclass -- `Button::paint()` in this case -- will be called.

To do:
Mention interface classes

Covariant return types

Covariant return types is the ability for a virtual function in a derived class to return a pointer or reference to an instance of itself if the version of the method in the base class does so. e.g.

```
class base
{
public:
  virtual base* create() const;
};

class derived : public base
{
public:
  virtual derived* create() const;
};
```

This allows casting to be avoided.

Note:
Some older compilers do not have support for covariant return types. Workarounds exist for such compilers.

virtual Constructors

There is a hierarchy of classes with base class `Foo`. Given an object `bar` belonging in the hierarchy, it is desired to be able to do the following:

1. Create an object `baz` of the same class as `bar` (say, class `Bar`) initialized using the default constructor of the class. The syntax normally used is:

   ```
   Bar* baz = bar.create();
   ```
2. Create an object `baz` of the same class as `bar` which is a copy of `bar`. The syntax normally used is:

   ```
   Bar* baz = bar.clone();
   ```

In the class `Foo`, the methods `Foo::create()` and `Foo::clone()` are declared as follows:

```
class Foo
{
    // ...

    public:
        // Virtual default constructor
        virtual Foo* create() const;

        // Virtual copy constructor
        virtual Foo* clone() const;
};
```

If `Foo` is to be used as an abstract class, the functions may be made pure virtual:

```
class Foo
{
   // ...

    public:
        virtual Foo* create() const = 0;
        virtual Foo* clone() const = 0;
};
```

In order to support the creation of a default-initialized object, and the creation of a copy object, each class `Bar` in the hierarchy must have public default and copy constructors. The virtual constructors of `Bar` are defined as follows:

```
class Bar : ... // Bar is a descendant of Foo
{
    // ...

    public:
    // Non-virtual default constructor
    Bar ();
    // Non-virtual copy constructor
    Bar (const Bar&);

    // Virtual default constructor, inline implementation
    Bar* create() const { return new Foo (); }
    // Virtual copy constructor, inline implementation
    Bar* clone() const { return new Foo (*this); }
};
```

The above code uses COVARIANT RETURN TYPES[33]. If your compiler doesn't support `Bar* Bar::create()`, use `Foo* Bar::create()` instead, and similarly for `clone()`.

33 Chapter 0.1.5 on page 25

While using these virtual constructors, you *must* manually deallocate the object created by calling **delete** `baz;`. This hassle could be avoided if a smart pointer (e.g. `std::auto_ptr<Foo>`) is used in the return type instead of the plain old `Foo*`.

Remember that whether or not `Foo` uses dynamically allocated memory, you *must* define the destructor **virtual** `~Foo ()` and make it `virtual` to take care of deallocation of objects using pointers to an ancestral type.

virtual Destructor

It is of special importance to remember to define a virtual destructor even if empty in any base class, since failing to do so will create problems with the default compiler generated destructor that will not be virtual.

A virtual destructor is not overridden when redefined in a derived class, the definitions to each destructor are cumulative and they start from the last derivate class toward the first base class.

Pure virtual Destructor

Every abstract class should contain the declaration of a pure virtual destructor.

Pure virtual destructors are a special case of pure virtual functions (meant to be overridden in a derived class). They must always be defined and that definition should always be empty.

```
class Interface {
public:
  virtual ~Interface() = 0; //declaration of a pure virtual destructor
};

Interface::~Interface(){} //pure virtual destructor definition (should always be
 empty)
```

EXPANDTEMPLATES[34] EXPANDTEMPLATES[35]

Law of three

The "law of three" is not really a law, but rather a guideline: if a class needs an explicitly declared copy constructor, copy assignment operator, or destructor, then it usually needs all three.

There are exceptions to this rule (or, to look at it another way, refinements). For example, sometimes a destructor is explicitly declared just in order to make it `virtual`; in that case there's not necessarily a need to declare or implement the copy constructor and copy assignment operator.

Most classes should not declare any of the "big three" operations; classes that manage resources generally need all three.

34 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
35 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

0.1.6 Subsumption property

Subsumption is a property that all objects that reside in a class hierarchy must fulfill: an object of the base class can be substituted by an object that derives from it (directly or indirectly). All mammals are animals (they derive from them), and all cats are mammals. Therefore, because of the subsumption property we can "treat" any mammal as an animal and any cat as a mammal. This implies abstraction, because when we are "treating" a mammal as an animal, the only we should know about it is that it lives, it grows, etc, but nothing related to mammals.

This property is applied in C++, whenever we are using pointers or references to objects that reside in a class hierarchy. In other words, a pointer of class animal can point to an object of class animal, mammal or cat.
Let's continue with our example:

```
 //needs to be corrected
  enum AnimalType {
       Herbivore,
       Carnivore,
       Omnivore,
  };

  class Animal {
        public:
               AnimalType Type;
               bool bIsAlive;
               int iNumberOfChildren;
  };

  class Mammal : public Animal{
        public:
               int iNumberOfTeats;
  };

  class Cat : public Mammal{
        public:
              bool bLikesFish;  // probably true
  };

  int main() {
      Animal* pA1 = new Animal;
      Animal* pA2 = new Mammal;
      Animal* pA3 = new Cat;
      Mammal* pM  = new Cat;

      pA2->bIsAlive = True;     // Correct
      pA2->Type = Herbivore;    // Correct
      pM->iNumberOfTeats = 2;   // Correct

      pA2->iNumberOfTeats = 6;  // Incorrect
      pA3->bLikesFish = True;   // Incorrect

      Cat* pC = (Cat*)pA3;      // Downcast, correct (but very poor practice, see
 later)
      pC->bLikesFish = False;   // Correct (although it is a very awkward cat)
   }
```

In the last lines of the example there is cast of a pointer to *Animal*, to a pointer to *Cat*. This is called "Downcast". Downcasts are useful and should be used, but first we must ensure that the object we are casting is really of the type we are casting to it. Downcasting a base class to an unrelated class is an error. To resolve this issue, the casting operators `dynamic_cast`[36], or `static_cast`[37]<> should be used. These correctly cast an object from one class to another, and will throw an exception if the class types are not related. eg. If you try:

```
Cat* pC = new Cat;

motorbike* pM = dynamic_cast<motorbike*>(pC);
```

Then, the app will throw an exception, as a cat is not a motorbike. Static_cast is very similar, only it will perform the type checking at compile time. If you have an object where you are not sure of its type then you should use `dynamic_cast`[38], and be prepared to handle errors when casting. If you are downcasting objects where you know the types, then you should use `static_cast`[39]. Do not use old-style C casts as these will simply give you an access violation if the types cast are unrelated.

0.1.7 Local classes

A **local class** is any class that is defined inside a specific statement block, in a LOCAL SCOPE[40], for instance inside a function. This is done like defining any other class, but *local classes* can not however access non-static local variables or be used to define STATIC DATA MEMBERS[41]. These type of classes are useful especially in template functions, as we will see later.

```
void MyFunction()
{
   class LocalClass
   {
   // ... members definitions ...
   };

   // ... any code that needs the class ...

}
```

0.1.8 User defined automatic type conversion

We already covered AUTOMATIC TYPE CONVERSIONS[42] (implicit conversion) and mentioned that some can be user-defined.

36 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FPROGRAMMING%20LANGUAGES%2FC%2B%2B%2FCODE%2FKEYWORDS%2FDYNAMIC_CAST
37 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FPROGRAMMING%20LANGUAGES%2FC%2B%2B%2FCODE%2FKEYWORDS%2FSTATIC_CAST
38 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FPROGRAMMING%20LANGUAGES%2FC%2B%2B%2FCODE%2FKEYWORDS%2FDYNAMIC_CAST
39 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FPROGRAMMING%20LANGUAGES%2FC%2B%2B%2FCODE%2FKEYWORDS%2FSTATIC_CAST
40 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FPROGRAMMING%20LANGUAGES%2FC%2B%2B%2FCODE%2FSTATEMENTS%2FSCOPE
41 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FPROGRAMMING%20LANGUAGES%2FC%2B%2B%2FCODE%2FKEYWORDS%2FSTATIC%2FDATA%20MEMBER
42 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FPROGRAMMING%20LANGUAGES%2FC%2B%2B%2FCODE%2FSTATEMENTS%2FVARIABLES%2FTYPE%20CASTING%23AUTOMATIC%20TYPE%20CONVERSION

A user-defined conversion from a class to another class can be done by providing a constructor in the target class that takes the source class as an argument, `Target(const Source& a_Class)` or by providing the target class with a conversion operator, as `operator Source()`.

0.1.9 Ensuring objects of a class are never copied

This is required e.g. to prevent memory-related problems that would result in case the default copy-constructor or the default assignment operator is unintentionally applied to a class `C` which uses dynamically allocated memory, where a copy-constructor and an assignment operator are probably an overkill as they won't be used frequently.

Some style guidelines suggest making all classes non-copyable by default, and only enabling copying if it makes sense. Other (bad) guidelines say that you should always explicitly write the copy constructor and copy assignment operators; that's actually a bad idea, as it adds to the maintenance effort, adds to the work to read a class, is more likely to introduce errors than using the implicitly declared ones, and doesn't make sense for most object types. A sensible guideline is to *think* about whether copying makes sense for a type; if it does, then first prefer to arrange that the compiler-generated copy operations will do the right thing (e.g., by holding all resources via resource management classes rather than via raw pointers or handles), and if that's not reasonable then obey the LAW OF THREE[43]. If copying doesn't make sense, you can disallow it in either of two idiomatic ways as shown below.

Just declare the copy-constructor and assignment operator, and make them `private`. Do not define them. As they are not `protected` or `public`, they are inaccessible outside the class. Using them within the class would give a linker error since they are not defined.

```
class C
{
  ...

  private:
    // Not defined anywhere
    C (const C&);
    C& operator= (const C&);
};
```

Remember that if the class uses dynamically allocated memory for data members, you *must* define the memory release procedures in destructor `~C ()` to release the allocated memory.

A class which only declares these two functions can be used as a private base class, so that all classes which privately inherits such a class will disallow copying.

Note:
A part of the BOOST[44] library, the utility class `boost:noncopyable` performs a similar function, easier to use but with added costs due to the required derivation.

43 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FCLASSES%2FMEMBER%20FUNCTIONS%23LAW%20OF%20THREE
44 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FLIBRARIES%2FBOOST

0.1.10 Container class

A class that is used to hold objects in memory or external storage is often called a *container class*. A container class acts as a generic holder and has a predefined behavior and a well-known interface. It is also a supporting class whose purpose is to hide the topology used for maintaining the list of objects in memory. When it contains a group of mixed objects, the container is called a heterogeneous container; when the container is holding a group of objects that are all the same, the container is called a homogeneous container.

0.1.11 Interface class

To do:
Complete

0.1.12 Singleton class

A Singleton[45] class is a class that can only be instantiated once (similar to the use of static variables or functions). It is one of the possible implementations of a creational pattern[46], which is fully covered in the Design Patterns Section[47] of the book.

ExpandTemplates[48]

45 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2FCode%2FDesign_Patterns%23Singleton
46 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2FCode%2FDesign_Patterns%23Creational_Patterns
47 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2FCode%2FDesign_Patterns
48 http://en.wikibooks.org/wiki/Category%3AC%2B%2B%20Programming

1 Authors

Edits	User
2	ADRIGNOLA[1]
4	DWARRIOR[2]
8	DARKLAMA[3]
1	DERBETH[4]
2	GRONAU[5]
1	GURUPATHI[6]
2	HAGINDAZ[7]
1	HERBYTHYME[8]
1	IXTLI[9]
1	JAMES BROWN[10]
1	JGUK[11]
1	JOHNOWENS[12]
14	LEANDROGOE[13]
5	MVHOKIES[14]
2	MERRHEIM[15]
1	MJCHAEL[16]
3	OMAIR.MAJID[17]
156	PANIC2K4[18]
3	PHOSGRAM[19]
2	REMI0O[20]
1	RFROHARDT[21]

1 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:ADRIGNOLA
2 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:DWARRIOR
3 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:DARKLAMA
4 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:DERBETH
5 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:GRONAU
6 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:GURUPATHI
7 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:HAGINDAZ
8 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:HERBYTHYME
9 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:IXTLI
10 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:JAMES_BROWN
11 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:JGUK
12 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:JOHNOWENS
13 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:LEANDROGOE
14 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:MVHOKIES
15 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:MERRHEIM
16 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:MJCHAEL
17 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:OMAIR.MAJID
18 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:PANIC2K4
19 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:PHOSGRAM
20 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:REMI0O
21 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:RFROHARDT

2 RONYCLAU[22]
1 SAE1962[23]
3 SIGMA 7[24]

22 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:RONYCLAU
23 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:SAE1962
24 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:SIGMA_7

List of Figures

1	TKGD2007[25]	GPL
2	TKGD2007[26]	GPL
3	TKGD2007[27]	GPL
4	TKGD2007[28]	GPL
5	TKGD2007[29]	GPL

25 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3ATKGD2007
26 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3ATKGD2007
27 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3ATKGD2007
28 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3ATKGD2007
29 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3ATKGD2007