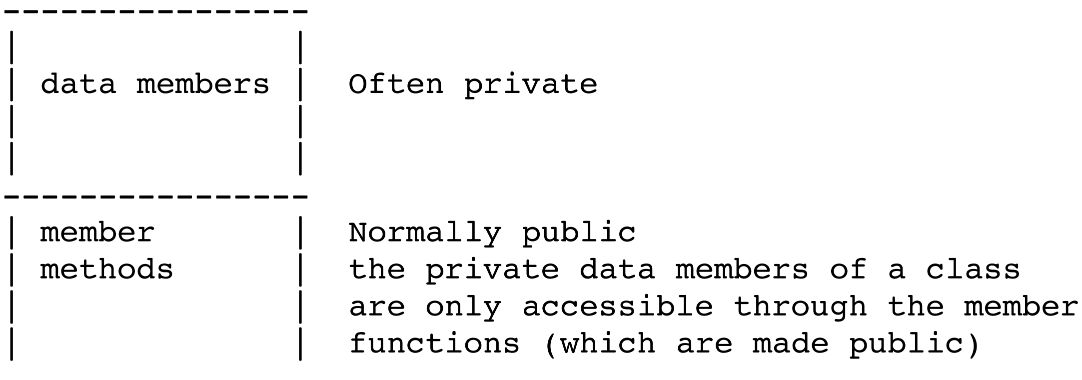
MECH215
Lecture Notes 9

Classes

Classes are the foundation of object-oriented programming in the C++ language. A C++ class is a collection of DATA MEMBERS (which may be of different types such as int, char, float, int*, etc.) and MEMBER FUNCTIONS (which are given the special name of METHODS). The class data type allows for the concept of ENCAPSULATION - a grouping of data members and the functions which operate on this data into a unified "object".

The class construct allows for information hiding - a class can restrict access to either its data members or its methods. Information within a class can be hidden by declaring the information to be 'private'. Conversely, information may be visible to users of the class by declaring it to be 'public'.

Here is a conceptual view of a class:



Circle Class Example

Here is a very simple example of a circle class in C++. A circle has associated with it a radius (stored as a float). We can define two methods called Area and Circumference which will return the area of a circle and its circumference. We will also define a method called SetRadius(float r) which will set the radius to a certain value which is passed as a parameter.

```
// Author: Ted Obuchowicz and John Peach
// Circle.h
// define a class called Circle, class is a C++ keyword

#ifndef Circle_h
#define Circle_h

class Circle {
    const static float PI = 3.1415926;
    float radius;       // private by default

    public:             // make these public
        float Area();
        float Circumference();
        void  SetRadius(float r);

};

#endif
```

```
// Author: Ted Obuchowicz and John Peach
// Circle.cpp

#include "Circle.h"

// Define the methods of the class.  Use scope resolution
// operator (::) to inform the compiler that the method belongs
// to the Circle

float Circle::Area() {
    return ( PI * radius * radius);
}

float Circle::Circumference() {
    return ( 2. * PI * radius);
}

void Circle::SetRadius(float r){
    radius = r;
}
```

```
// Author: Ted Obuchowicz and John Peach
// CircleDemo.cpp
// example program illustrating a simple Circle Class

#include <iostream>
#include "Circle.h"

using namespace std;

int main() {

    // create two Circle "objects"
    Circle circle1, circle2;

    circle1.SetRadius(2.00); // set the Radius to 2.00
    cout << "circle1 has area " << circle1.Area() << endl;
    cout << "circle1 has circumference " << circle1.Circumference()
<< endl;

    circle2.SetRadius(5.00);
    cout << "circle2 has area " <<  circle2.Area() << endl;
    cout << "circle2 has circumference " << circle2.Circumference()
<< endl;

    return 0;
}

The program output is:

circle1 has area 12.5664
circle1 has circumference 12.5664
circle2 has area 78.5398
circle2 has circumference 31.4159
```

Some comments on the program:

1) Note how we give the full name of the methods when we define them using the form:

return_type class_name::method_name

Note there is no space between the class_name::method_name. This is because there may be similarly named methods belonging to different classes.

2) We could have also defined the methods within the class declaration. This style of defining a method is known as INLINE DEFINITION. Here is the same program using the inline definition form for the method definitions. This is commonly done in the Java programming language but rarely in C++.

```
// Author: Ted Obuchowicz and John Peach
// CircleInline.h
// define a class called Circle with inline definitions

#ifndef CircleInline_h
#define CircleInline_h

class Circle {
    const static float PI = 3.1415926;
    float radius;       // private by default

    public:             // make these public
        float Circle::Area() {
            return ( PI * radius * radius);
        }

        float Circle::Circumference() {
            return ( 2. * PI * radius);
        }

        void Circle::SetRadius(float r){
            radius = r;
        }

};

#endif
```

Of course, the output produced by this version is exactly the same as the output produced by the earlier version:

```
circle1 has area 12.5664
circle1 has circumference 12.5664
circle2 has area 78.5398
circle2 has circumference 31.4159
```

3) IT IS AN ERROR TO ATTEMPT TO DIRECTLY ACCESS THE PRIVATE DATA OF A CLASS. Suppose in a main program we try to read the value of circle1's radius. For example, with the same class definition as above (either the inline one or the first one), if we write a main program such as:

```
// Author: Ted Obuchowicz and John Peach
// CirclePrivate.cpp
// example program illustrating a simple Circle Class

#include <iostream>
#include "Circle.h"

using namespace std;

int main() {

    // create a Circle
    Circle circle1;

    circle1.SetRadius(2.00); // set the Radius to 2.00
    cout << "circle1 has a radius " << circle1.radius << endl;

    return 0;
}
```

When compiled this program, the following error is reported:

```
Circle.h: In function 'int main()':
Circle.h:7:11: error: 'float Circle::radius' is private
CirclePrivate.cpp:16:48: error: within this context
```

Using Header Files for Class Definitions

Just as we used header files to include function prototypes, we use of header files to include class definitions. We also make use of the #ifndef compiler directive to prevent multiple redefinitions of identifiers in cases where a header file is included more than one time.

We can put the definition of the class Circle in a file called Circle.h (any file name may be used... it is good to follow
the class_name.h convention).

Another file containing the implementation of the methods found in the class. This file is called Circle.cpp.

The main program is written in a third file. Let us call it CircleDemo.cpp.

We can compile the above files using the command:

```
g++ -o CircleDemo Circle.cpp CircleDemo.cpp
```

Note, the .h files are not part of the command line options. This is because the file is "included" in the .cpp files.

Linking Your Program With A Pre-Compiled Class Implementation

The above example showed how we separated the class DEFINITION from its IMPLEMENTATION into two separate files. It is even possible (and often desirable) to further isolate the user of the class from its implementation by providing only the precompiled version of the class implementation (that is providing only the ".o" file [object file]). All the user of a class needs to know is the interface of the class: the class name, the name of its methods, etc. All this information is provided in the class definition. The actual details of how the methods operate are left to the implementer to decide.

We can precompile our Circle.cpp class into a binary object file called Circle.o using the following option to the g++ compiler:

-c Compile or assemble the source files, but do not link. The compiler output is an object file corresponding to each source file.

```
#>g++ -c Circle.cpp
```

This will create a binary object file called Circle.o Binary files are not human readable.

Next, we can link our Circle.o with our main program by doing:

```
#>g++ -o CircleDemo Circle.o CircleDemo.cpp
```

This command instructs the g++ compiler to compile the source code in file CircleDemo.cpp and LINK it together with the precompiled binary file Circle.o and create a single executable file called CircleDemo.

When you compile and link a program which makes use of library routines such as cin, cout, etc. the linker uses precompiled object files for these routines. Some of these routines are found in the directory /usr/lib. For example, there is a file called /usr/lib/libCrun.so.1 which contains various C runtime library utilities. The .so file name extension is the so called "shared object" library extension name. There is a very nice explanation of libraries and shared libraries in the textbook "Deep C Secrets: Expert C Programming". This is a wonderfully written book.

Constructors

The above circle class contains a method called SetRadius which must be manually invoked by the user to set the radius of a certain circle object to a specific value. This is burdensome and error-prone, a user of the class may forget to do so.

C++ has a way of automatically initializing the data members of a class. The method makes use of CONSTRUCTORS. Constructors are special purpose methods (which have the same name as the class), constructors are automatically called by the compiler when an object of the class is created.

Constructors differ from ordinary functions and methods in that NO RETURN TYPE (NOT EVEN A RETURN TYPE OF VOID) is specified in the definition of a constructor.

Default / No-Argument Constructors

If a constructor takes no arguments, it is known as a default or a no-argument constructor. If it is possible to define an object with default parameters then you should always include one.

Overloaded Constructors

A class definition may contain more than one definition of a constructor, we may define different versions of the constructor function by OVERLOADING the constructor (that is multiple definitions which vary in their argument list).

Circle Class With a Default and Overloaded Constructor:

We will modify our circle class example to include a default constructor (which will set the radius of a circle object to 1.00) and an overloaded constructor (which will set the radius to some user specified value). We will also include a new method called GetRadius, this method will return the radius of the circle object:

```
// Author: Ted Obuchowicz and John Peach
// CircleConstructor.h
// example program illustrating use of
// a simple Circle Class using constructors

#ifndef CircleConstructor_h
#define CircleConstructor_h

const float PI = 3.1415926;

// define a class called Circle, class is a C++ keyword

class Circle {
  float radius;     // everything in a class is private by default

  public:           // we want the following methods to be public, so we
                    // make them public by using the public keyword
followed
                    // by a colon :
  Circle();         // default constructor
  Circle(float r);  // overloaded constructor
  float Area();
  float Circumference();
  float GetRadius();
};

#endif
```

We now define the methods of the class, we use the scope resolution operator to inform the compiler that the method belongs to the class called Circle

```
// Author: Ted Obuchowicz and John Peach
// CircleConstructor.cpp
// example program illustrating use of
// a simple Circle Class using constructors

#include <iostream>
#include "CircleConstructor.h"

float Circle::Area() {
  return PI * radius * radius;
}

float Circle::Circumference() {
  return 2. * PI * radius;
}

float  Circle::GetRadius() {
  return radius;
}

// define the default constructor
Circle::Circle() {
  std::cout << "Default constructor called... " << std::endl;
  std::cout << "Setting circle's radius to 1.00 " << std::endl;
  radius = 1.0;
}

// define the overloaded constructor

Circle::Circle(float r) {
  std::cout << "Overloaded constructor invoked... " << std::endl;
  radius = r;
}
```

Let us use the class definition and the definitions of the three methods in a main program

```
// Author: Ted Obuchowicz and John Peach
// CircleConstructorDemo.cpp
// example program illustrating use of
// a simple Circle Class using constructors

#include <iostream>
#include "CircleConstructor.h"
```

```
int main() {

  // create two Circle "objects"

  Circle defaultCircle;           // using the default constructor
  Circle sizedCircle(456.89);     // Use the overloaded constructor

  std::cout << "Circle defaultCircle has radius "
            << defaultCircle.GetRadius() << std::endl;
  std::cout << "Circle sizedCircle has radius "
            <<  sizedCircle.GetRadius() << std::endl;

  return 0;
}
```

The output is:

```
#> g++ -o CircleConstructorDemo CircleConstructorDemo.cpp CircleConstructor.cpp
Default constructor called...
Setting circle's radius to 1.00
Overloaded constructor invoked...
Circle defaultCircle has radius 1
Circle sizedCircle has radius 456.89
```

Operator Overloading

C++ allows to overload built-in operators such as +, -, *, etc. Let us overload the + operator such that it will "add" two circle objects together. For the purposes of our example, we will define the addition of two circles to give a third circle whose radius is the sum of the two circles being added together.

```
// Author: Ted Obuchowicz and John Peach
// CircleAdd.h
// example program illustrating use of
// a simple Circle Class to overload an operator

#ifndef CircleAdd_h
#define CircleAdd_h

const float PI = 3.1415926;

class Circle {
  float radius;     // everything in a class is private by default

  public:           // we want the following methods to be public, so we
                    // make them public using the public keyword followed
                    // by a colon :
  Circle();         // default constructor
  Circle(float r);  // overloaded constructor
  float Area();
  float Circumference();
```

```
  float GetRadius();
  Circle operator+(Circle); // overload the operator +
};

#endif
```

We now define the methods of the class, we use the scope resolution operator to inform the compiler that the method belongs to the class called Circle

```
// Author: Ted Obuchowicz and John Peach
// CircleAdd.cpp
// example program illustrating use of
// a simple Circle Class using constructors

#include "CircleAdd.h"

float Circle::Area() {
  return PI * radius * radius;
}

float Circle::Circumference() {
  return 2. * PI * radius;
}

float  Circle::GetRadius() {
  return radius;
}

Circle::Circle() {
  radius = 1.0;
}

Circle::Circle(float r) {
  radius = r;
}

Circle Circle::operator+(Circle otherCircle) {

  // Create the circle that we are going to return
  // and set its radius
  Circle temp(radius + otherCircle.radius);
  return temp;
}
```

```
// Author: Ted Obuchowicz and John Peach
// CircleAddDemo.cpp
// example program illustrating use of
// a simple Circle Class using constructors

#include <iostream>
#include "CircleAdd.h"

int main() {

  // create two Circle "objects"

  Circle Circle1(50);
  Circle Circle2(120);
  Circle Circle3;

  std::cout << "Circle Circle1 has radius " << Circle1.GetRadius()
            << std::endl;
  std::cout << "Circle Circle2 has radius " <<  Circle2.GetRadius()
            << std::endl;

  // add the two circles together
  Circle3 = Circle1 + Circle2;
  std::cout << "Circle Circle3 has radius " <<  Circle3.GetRadius()
            << std::endl;

  return 0;
}
```

The program output is:

```
#> g++ -o CircleAddDemo CircleAddDemo.cpp CircleAdd.cpp
#> ./CircleAddDemo
Circle Circle1 has radius 50
Circle Circle2 has radius 120
Circle Circle3 has radius 170
```

NOTE: The line

```
Circle3 = Circle1 + Circle2;
```

can also be written as:

```
Circle3 = Circle1.operator+(Circle2);
```

Written in this manner, it is clearer that we are actually invoking the method called operator+ belonging to the circle Circle1 and passing to this method the circle Circle2 as a parameter. The return value of the operator+ method is then assigned to the circle Circle3.

If we write it as Circle3 = Circle1 + Circle2, it is more apparent that we are "adding" two circles together and assigning the result of the "addition" to another circle.

Differences Between A C++ Struct And A C++ Class

A C++ struct is very similar to a C++ class. In fact, the only difference between C++ structs and classes is that in A STRUCT EVERYTHING IS CONSIDERED PUBLIC (unless explicitly denoted as private) BY DEFAULT AND IN A CLASS EVERYTHING IS CONSIDERED PRIVATE (unless explicitly denoted as public by use of the public keyword). Traditionally, C++ programmers use the class construct when we want to encapsulate the data members togethers with the methods which operate on these members. A struct is traditionally used only when there are data members (all public) and no methods. As always, one can abandon tradition and be a rebel. So, let us be a bit rebellious and use a struct which has some methods in addition to data members:

```
// Author: Ted Obuchowicz and John Peach
// Fraction.cpp
// Example of using a struct the same way as using a class

#include <iostream>

struct Fraction {
  private: // in a struct everything is public unless made private
    int numerator;
    int denominator;

 public:
    Fraction() {  // inline definition of default constructor
      numerator = 1;
      denominator = 1;
    };

    Fraction(int top, int bottom) {
      numerator = top;
      denominator = bottom;
    };

    Fraction add(Fraction x) {
      Fraction temp;
      temp.numerator = numerator * x.denominator +
                       x.numerator * denominator;
      temp.denominator = denominator * x.denominator;
```

```
      return temp;
    };

    void printFraction() {
      std::cout << numerator << " / " << denominator << std::endl;
    }
};

int main() {

  // Create 2 fractions that are to be added plus one to store it in
  Fraction F1(2,3);
  Fraction F2(3,4);
  Fraction sum;

  // Print the fractions, add then together and report
  F1.printFraction();
  F2.printFraction();
  sum = F1.add(F2);
  sum.printFraction();

  return 0;
}
```

The output is:

```
2 / 3
3 / 4
17 / 12
```

This example illustrates the INLINE method of defining the methods of the struct (or a class as well). Note if we use the inline methods there is no need to use the scope resolution operator together with the class name to give the "full name" as was done in the previous circle class examples. Note that we use a method called add (instead of overloading the + operator which we could have done if we so desired... this is left as an exercise to the interested reader) which performs the addition of two fraction objects. No reduction of the fraction to its "lowest common denominator" is performed. Again, the interested reader is encouraged to do this as an exercise.

A Complex Number Class

Here is a more representative example of a C++ class which provides for adding two complex numbers together (by overloading the + operator). Recall that a complex number consists of a real part and an imaginary part:

complex number = real part + i (imaginary part)

where i is $\sqrt{-1}$.

Complex numbers are used frequently in electrical engineering, most notably in the study of electromagnetic theory.

Some comments on the program follow:

```
// Author: Ted Obuchowicz and John Peach
// Complex.h

#ifndef Complex_h
#define Complex_h

#include <iostream>

class Complex {
  double real;
  double imaginary;

  public:
    Complex() { real = 0.0; imaginary = 0.0; }
    Complex(double,double);
    void write() {
      std::cout << real << " + " << imaginary << "i" << std::endl;
    }

    Complex operator+(const Complex) const ;
};

#endif
```

```
// Author: Ted Obuchowicz and John Peach
// Complex.cpp

#include "Complex.h"

Complex::Complex(double real, double imaginary) {
  this->real = real;
  (*this).imaginary = imaginary;
}

Complex  Complex::operator+(const Complex c) const {
  Complex temp;    // create the return value

  // Set the real and imaginary components
  temp.real = real + c.real;
```

```
  temp.imaginary = imaginary + c.imaginary;

  return temp;
}
```

```
// Author: Ted Obuchowicz and John Peach
// ComplexDemo.cpp

#include "Complex.h"

int main() {
  Complex c1;
  Complex c2(7.7, 4.4);
  Complex c3(1.1, 2.2);
  c1 = c2 + c3;
  c1.write();
}
```

The output is:

```
8.8 + 6.6i
```

Comments:

The const at the end of the declaration of the operator+

`Complex  Complex::operator+(const Complex c)` <u>`const`</u>

The underlined const simply guarantees that the object which is calling (ie. object c2 in our sample program) the method operator+ will NOT be changed by the method.

The const in the parameter list :

`Complex  Complex::operator+(`<u>`const`</u>`  Complex c) const`

The underlined const guarantees that the operator+ method will not change the object which is passed to it as a parameter (ie. the object c3 in the sample program).

If the method operator+ attempted to do something stupid like change the values of either the calling object or the object received as a parameter, it would be flagged as an error by the compiler.

In the overloaded constructor:

```
Complex::Complex(double real, double imaginary) {
  this->real = real;
  (*this).imaginary = imaginary;
}
```

The formal parameters are called real and imaginary. They are local to the constructor. However, the class also has instance variables called real and imaginary and they exist everywhere in the class, including the constructor. To avoid confusion between which variable we are using we can use the "this" keyword. "this" is automatically part of any object and is a pointer to the object itself. In this example it is a pointer of type Complex*. The -> operator is used to access instance variables or methods. So, the "real" instance variable (the one that is defined in the class) can be accessed with this->real and the local variable can be accessed simply by typing "real". The use of "this" is not required and is generally only used when there is the potential for confusion. If we exam the no-argument constructor:

```
Complex() { real = 0.0; imaginary = 0.0; }
```

We will see that real and imaginary are using used without the “this” reference. This is because there are no local variables that conflict with the name. We could have written the constructor this way:

```
Complex() { this.real = 0.0; this.imaginary = 0.0; }
```

But there is no need and the preferred method is to drop "this" when it is not needed.

Destructors

Constructors are used to perform INITIALIZATION of data members. This initialization may involve some DYNAMIC MEMORY allocation (for example assign some pointer an address returned by the new operator). Recall that when an object which dynamically allocates some memory goes out of scope, the memory used for that object's data members are returned to the operating system (and given back to the heap space, HOWEVER ANY DYNAMICALLY ALLOCATED MEMORY IS NOT RETURNED. Thus, a memory leak is said to occur, and we have the possibility of exhausting the heap space causing the program to crash or start to run very slowly.

PLEASE DO NOT ACTUALLY RUN THIS PROGRAM ON THE ENCS SERVER AS IT WILL USE UP ALL THE MEMORY. IT WILL CAUSE ALL KINDS OF PROBLEMS FOR OTHER USERS AND THE SYS ADMINS. IT WOULD BE A VIOLATION OF THE TERMS OF USE OF YOUR ACCOUNT AND YOU MANY HAVE YOUR COMPUTER ACCESS REVOKED. If you want to run it on your personal computer, go for it.

This is illustrated in the program below:

```
// Author: Ted Obuchowicz and John Peach
// Big.cpp

#include <iostream>
#include <cstdlib>

using namespace std;

const long GIGABYTE = 1073741824;
```

```
class Big {
  private: // while instance variables are private by default get in
the habit of
           // always using the private keyword
  char* string; // a data member which is a pointer to a character..

  public:
    Big(); // default constructor does DYNAMIC memory allocation

};

Big::Big() {
  string = new char [GIGABYTE];  // get a gigabyte worth of char
cells
  if (!string) {
    cout << "OUT OF MEMORY!!!!" << endl;
    exit(1);
  }
}

void garbage(void) {
  Big oops;

  // do some local processing of object oops
}

int main() {

  /* WARNING:
        DO NOT ACTUALLY RUN THIS PROGRAM ON THE ENCS SERVER AS IT
        WILL USE UP ALL THE MEMORY.
        IT WILL CAUSE ALL KINDS OF PROBLEMS FOR OTHER USERS AND THE
        SYS ADMINS.
        IT IS A VIOLATION OF THE TERMS OF USE OF YOUR ACCOUNT AND YOU
        MANY HAVE YOUR COMPUTER ACCESS REVOKED

        You have been warned.

        If you want to run it on your personal computer, go for it.
  */

  unsigned int i = 0;

  while(true) {
    cout << "I = " << ++i << endl;  // we get a segmentation fault
```

```
    garbage();
  }
}
```

We have a class which has a single character pointer has a data member. There is also a default constructor defined which assigns the pointer the starting address of a dynamically allocated array of 1 073 741 824 bytes (a gigabyte). There is a function called garbage which simply creates an object of class Big (recall that the default constructor will be called whenever an object of class Big is declared).

In the main program, we have a loop which calls the function garbage an infinite number of times (well, until we run out of memory). Each time through the loop, the default constructor will be called (allocating 1 073 741 824 bytes from the heap), eventually we will exhaust all of the available memory and the program will crash.
On my workstation, this occurred during loop iteration 131071:

My output is:

```
I = 131067
I = 131068
I = 131069
I = 131070
I = 131071
terminate called after throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc
Aborted (core dumped)
```

Solutions:

There are two solutions. One is to have the user of the class, explicitly give back any dynamically allocate memory with the delete operator when that memory is no longer needed:

```
// Author: Ted Obuchowicz and John Peach
// BigClear.cpp

#include <iostream>
#include <cstdlib>

using namespace std;

const long GIGABYTE = 1073741824;

class Big {
  private: // instance variables are private by default but get in
           // the habit of always using the private keyword
  char* string; // a data member which is a pointer to a character.
```

```
  public:
    Big(); // default constructor will do DYNAMIC memory allocation
    void clear(); // method  which will deallocate dynamic memory
};

void  Big::clear() {
  delete [] string;
  string = 0;
}

Big::Big() {
  string = new char [GIGABYTE];  // get a gigabyte worth of chars
  if (!string) {
    cout << "OUT OF MEMORY!!!!" << endl;
    exit(1);
  }
}

void garbage(void) {
 Big oops;

 // using this method we must remember to invoke the method
 // clear before returning from from the function garbage

 oops.clear();

 // this is not elegant as it relies upon the developer implementing
 // function garbage to remember to delete any memory which was
 // dynamically allocated by the constructor function Big()

 // would it not be simpler if there were a way to have this done
 // automatically somehow..?

}

int main() {

  unsigned int i = 0;

  // Let us set a limit of the loop so that it goes not run forever
  while(i < 1000000) {
    cout << "I = " << ++i << endl;  // we get a segmentation fault
    garbage();
  }
}
```

Now, there is no problem. It will run until I = 1000000, which is where we told it to stop. However, this method is cumbersome. It relies upon the user to call the method clear() explicitly. There is a better way which uses DESTRUCTORS.

DESTRUCTOR functions are special functions inside of class objects, they NEITHER TAKE ANY ARGUMENTS, NOR DO THEY RETURN A VALUE; DESTRUCTORS ARE HIGHLY SPECIALIZED FUNCTIONS WHICH EXIST PRIMARLY TO RELEASE ANY MEMORY WHICH WAS DYNAMICALLY ALLOCATED BY A CONSTRUCTOR!!!!!! They are also used to clean up anything, such as closing files, etc. when the object is on longer being used.

A DESTRUCTOR's name is similar to the constructor in the respect that each is the same as the class name. However, a destructor name is preceded with the tilde (~) symbol.

Here is a program which makes use of the destructor ~Big()

```
// Author: Ted Obuchowicz and John Peach
// BigDestructor.cpp

#include <iostream>
#include <cstdlib>

using namespace std;

const long GIGABYTE = 1073741824;

class Big {
  private: // instance variables are private by default get in the
           // habit of always using the private keyword
  char* string; // a data member which is a pointer to a character.

  public:
    Big();  // do some DYNAMIC memory allocation
    ~Big(); // destructor which will deallocate dynamic memory
};

Big::~Big() {
  delete [] string;
  string = 0;
}

Big::Big() {
  string = new char [GIGABYTE];  // get a gigabyte worth of chars
  if (!string) {
    cout << "OUT OF MEMORY!!!!" << endl;
    exit(1);
```

```
  }
}

void garbage(void) {
 Big oops;

 // do some local processing of object oops

}

int main() {

  unsigned int i = 0;

  // Let us set a limit of the loop so that it goes not run forever
  while(i < 1000000) {
    cout << "I = " << ++i << endl;  // we get a segmentation fault
    garbage();
  }
}
```

This program completes its for loop with no memory problems:

```
I = 999994
I = 999995
I = 999996
I = 999997
I = 999998
I = 999999
I = 1000000
```

It should be noted here that the variable oops in function garbage is created when oops is called and then distroyed when oops finishes. Therefore, there is only ever one copy of oops in memory. This is the case in all of the examples. However, when an object of type Big is created it asks the OS to dynamically allocate memory for string. However, this dynamically allocated memory is technically part of the memory used by an object of type Big and therefore, when the function garbage terminates and removes oops it does not automatically clean up the dynamically allocated memory used by an object of type Big. Therefore, we need to have the destructor do this for us. Note, we never call the destrutor. It is called when the object is being deleted.

More On Constructors

Constructors are automatically invoked by the compiler when a class object is declared. As such, they cannot be invoked in the "regular" manner as a method function is invoked. The following program will generate a compile-time error:

```
// Author: Ted Obuchowicz and John Peach
// ConstructorError.cpp

#include <iostream>

using namespace std;

class C {
  private:
    int x ;
  public:
    C();
    void print(void);
} ;

// define the default constructor

C::C() {
  cout << "default constructor called" << endl;
  x = 1;
};

void C::print(void) {
  cout << x << endl;
}

int main() {

  C x;
  x.print();

  // see if we can invoke a constructor explicitly...
  x.C();

  // nope it gives a compile time error

  return 0;
}
```

In this program, we attempt to invoke explicitly the constructor C() using the objectName.methodName() style. This results in a compile time error.
ConstructorError.cpp: In function ‘int main()’:

```
ConstructorError.cpp:34:5: error: invalid use of ‘C::C’
```

There is a way to explicitly invoke the constructor function, but is has to be called at the time of the

object's declaration:

```
// Author: Ted Obuchowicz and John Peach
// ConstructorCall.cpp

#include <iostream>

using namespace std;

class C{
  private:
    int x ;
  public:
    C();
    C(int num);
    void print(void);
} ;

// define the default constructor

C::C() {
  cout << "default constructor called" << endl;
  x = 1;
};

C::C(int num) {
  cout << "overloaded constructor called" << endl;
  x = num;
};

void C::print(void) {
  cout << x << endl;
};

int main() {

    C x = C(55);   // make a call to the overloaded constructor
    x.print();

    C y = C();     // make a call to the default constructor
    y.print();

    return 0;
}
```

The program output is:

```
overloaded constructor called
55
default constructor called
1
```

Member Initialization in Constructors

C++ allows an alternate form for a constructor to perform member initialization by a mechanism known a "base/member initialization". The following program illustrates its use:

```
// Author: Ted Obuchowicz and John Peach
// ConstructorInitialization.cpp

#include <iostream>

using namespace std;

class X {
  private:
    int i;
    float f;
    char c;

  public:
    X(int first=1, float second=2.0, char third='a') : i(first) ,
f(second) , c(third) { }
    void print() { cout << i << " " << f << " " << c << endl;}
};

int main() {

  X var1;                 // no-argument constructor
  X var2(6, 2.34, 'z'); // Overloaded constructor
  X var3(7);              // Is there a valid method?

  var1.print();           // Prints the default values
  var2.print();           // Prints the passed parameters
  var3.print();           // What will this print?

  return 0;
}
```

The output is:

```
1 2 a
6 2.34 z
7 2 a
```

Observe that there is only one constructor method and that we have called it with three different sets of actual arguments. This can be done because default values have been set. Any missing parameters will use the default values. If the parameters are present then the actual values, not the default, will be used.

Also observe that the member/base initialization is done outside of the function definition.

Assignment of One Class Object to Another of The Same Type

An object of a class may be assigned to another object of the same class. The compiler provides a default assignment operator which simply copies the memory used by one object into the memory used by another object. Effectively, it performs a member by member assignment.

```
// Author: Ted Obuchowicz and John Peach
// ObjectAssignment.cpp

#include <iostream>

using namespace std;

class X {
  private:
    int i;
    float f;
    char c;

  public:
  X() {
    i = 1 ; f = 2.0 ; c = 'a';
    cout << "default called" << endl;
  }
  X(int first, float second, char third) {
    i = first; f = second; c = third;
    cout << "overloaded called" << endl;
  }
  void print() { cout << i << " " << f << " " << c << endl; }

};
```

```
int main() {

  X var1;
  X var2(10,100.0,'z');
  var1.print();
  var2.print();

  var2 = var1 ;  // the compiler provides a default assignment
                 // operator which simply performs a member by member
                 // assignment

  var2.print();

  return 0;

}
```

The output is:

```
default called
overloaded called
1 2 a
10 100 z
1 2 a
```

We can always provide our own version of the overloaded assignment (=) operator instead of relying upon the one which is provided default by the compiler:

```
// Author: Ted Obuchowicz and John Peach
// ObjectAssignmentOverload.cpp

#include <iostream>

using namespace std;

class X {
  private:
    int i;
    float f;
    char c;

  public:
  X() {
    i = 1 ; f = 2.0 ; c = 'a';
    cout << "default called" << endl;
  }
```

```
  X(int first, float second, char third) {
    i = first; f = second; c = third;
    cout << "overloaded called" << endl;
  }
  void print() { cout << i << " " << f << " " << c << endl; }
  void operator=(X rhs);
};

void X::operator=(X rhs)
{
  cout << "using the user-defined assignment operator" << endl;

  // now simply copy the values of the data members of rhs
  i = rhs.i;
  f = rhs.f;
  c = rhs.c;
};

int main() {

  X var1;
  X var2(10,100.0,'z');

  var1.print();
  var2.print();

  var2 = var1 ;  // use the programmer supplied version of the
                 // assignment operator.  This is the same as
                 // var2.operator=(v1);

  var2.print(); // print out the values of the data members

  return 0;
}
```

The output is:

```
default called
overloaded called
1 2 a
10 100 z
using the user-defined assignment operator
1 2 a
```

Why would we want to provide our own version of the assignment operator when the compiler always provides its own default one? There are certain cases in which a class contains a pointer variable in

which the compiler provided assignment operator is "not good enough". In order to understand why we have to learn a little about garbage and dangling pointers.

Garbage and Dangling Pointers

The following program produces what is known as a "dangling pointer":

```
// Author: Ted Obuchowicz and John Peach
// DanglingPointer.cpp

#include <iostream>

using namespace std;

int main() {

  int* ptr ; // declare a pointer to an int

  ptr = new int(5); // request space and initialize to 5
                    // and assign address of allocated space to ptr

  cout << "The value of ptr is " << ptr
       << " and the data stored in this location is "
       << *ptr << endl;

  delete ptr ; // give back the allocated space to the heap

  // ptr is now called a dangling pointer.. it does not point to any
  // valid memory location... it should not be dereferenced until it
  // has been assigned some valid address... note the value of
  // ptr is not affected by the delete operator... it still
  // holds the same address... except that this address is "no longer
  // in existence"

  cout << "The value of ptr after the delete is " << ptr << endl;

  return 0;
}
```

Garbage is the result of when we dynamically allocate memory from the heap, assign its address to some pointer variable, and then assign some other address to the pointer variable BEFORE we have deleted the allocated memory (returned it back to the heap). The allocated memory can now NEVER be returned back to the heap (at least until the program completes execution) since we now longer have its address available. Such memory is referred to as GARBAGE. The following program illustrates how a program may generate garbage.

```
// Author: Ted Obuchowicz and John Peach
// Garbage.cpp

#include <iostream>

using namespace std;

int main() {

  int* a = new int(10);

  cout << "An integer with value 10  has been allocated from the heap
at address: " << a << endl;

  a = new int(2) ;   // allocated an integer with initial value.  The
                    // old value is lost and cannot be reclaimed
                     // since we no longer have its address ... It is
                     // said to be GARBAGE.

  cout << "The original pointer holding the integer value with 10 is
now assigned a new address: " << a << endl;

  delete a  ; // give back the second integer to the heap

  return 0;
}
```

The output is:

```
An integer with value 10  has been allocated from the heap at
address: 0x1b1d010
The original pointer holding the integer value with 10 is now
assigned a new address: 0x1b1d030
```

If a class contains data members which have been made to point to dynamically allocated memory and a default assignment operator is used to assign one object to another, garbage can result as in the following example:

```
// Author: Ted Obuchowicz and John Peach
// GarbageClass.cpp

#include <iostream>

using namespace std;

class X {
```

```
  private:
    int* ptr;

  public:
    X() { ptr = new int ;}
    void showAddress() { cout << "Value of ptr is: " << ptr << endl;}

};

int main() {

  X var1, var2;
  var1.showAddress();
  var2.showAddress();

  var2 = var1 ;

  var1.showAddress();
  var2.showAddress();

  return 0;
}
```

The output is:

```
Value of ptr is: 0x6e4010
Value of ptr is: 0x6e4030
Value of ptr is: 0x6e4010
Value of ptr is: 0x6e4010
```

After the statement var2 = var1; we no longer have a pointer to the memory which was allocated from 0x211f8 because var2.ptr has been overwritten with value of var1.ptr (which points to some other place in the heap). We have created some garbage at address 0x6e4030 and this garbage cannot be de-allocated with the delete operator since we no longer have its address.

Even if the class contains a destructor function, the use of the compiler provided assignment operator will result in a dangling pointer and garbage. The details of the "picture" of main memory in the case of a user-provided destructor functions which simply performs ~X() { delete ptr } is left to the interested reader... I highly encourage you to go through this constructive exercise.

We can provide our own version of the assignment operator which circumvents this problematic behaviour:

```
// Author: Ted Obuchowicz and John Peach
// NoGarbageClass.cpp

#include <iostream>

using namespace std;

class X {
  private:
    int* ptr;

  public:
    X(int value) {
      ptr = new int(value); // allocated memory from the heap
    }
    void showAddress() {
      cout << "Value of ptr is: " << ptr
           << " and the value stored is: " << *ptr << endl;
     }
     void operator=(X arg);
};

void X::operator=(X arg) {

  if ( ptr ) {
    cout << "Delete the space at address " << ptr << endl;
    delete ptr;
  };
  ptr = new int;       // allocate space from the heap
  *ptr = *(arg.ptr);  // copy the value from one to another
};

int main() {

  X var1(5);
  X var2(6);
  var1.showAddress();
  var2.showAddress();

  var2 = var1 ; // no more garbage

  var1.showAddress();
  var2.showAddress();

  return 0;
}
```

The program output is:

```
Value of ptr is: 0x919010 and the value stored is: 5
Value of ptr is: 0x919030 and the value stored is: 6
Delete the space at address 0x919030
Value of ptr is: 0x919010 and the value stored is: 5
Value of ptr is: 0x919030 and the value stored is: 5
```

Something interesting happened in that when the memory for ptr is var2 was released to the heap. We then requested a new chunk of memory and the OS returned the same address. This does not always happen.