Chapter 9: Introduction to Objects

- Declaring and Defining a Class
- Data encapsulation; public and private class members; getter/setter methods
- Pointers to classes
- Initializing data
- Constructors and default constuctors
- The “this” pointer
- Initialization lists
- Destructors
- Shallow and deep copies, copy constructors, and copy by assignment
- Intro to move constructors
- Classes that don’t allow copying, singleton classes
- Classes only creatable on the heap
- sizeof() a class
- Friend Functions and Friend Classes
- Const member functions

Introduction to Objects

An *object* is a user-defined datatype like an integer or a string. Unlike those simple datatypes, though, an object can have much richer functionality. It typically collects some data ("member data") and some functionality ("methods"). For example, we might create a class to handle a matrix, or a tensor, or a student's record in a class, etc.

Declaring and Defining a Class

Before we can use variables of a given class, we first have to specify the class. Analogous to functions, we can *declare* the class by specifying any member data it contains and providing a list of its available functions (along with what arguments those functions take and their return types). This is often done in a header file (whose name is typically the name of the class, with a ".h" suffix). We can then define the functions in the class in another file if we like (often with the name of the class with a ".cc" suffix). Alternatively, some or all of the function definitions can also go in the header file (often this is done for short functions or inline functions).

Example: The Student Class

- Suppose we want to keep track of students taking a course. Each student will have a name, a midterm grade, a final grade, and a course project grade. These grades will be used to compute a final course grade.

```
//Listing of student1.cc:
#include <iostream>
#include <string>
using namespace std;

class Student
{
  private:
    string Name;
    double MidtermGrade;
    double FinalGrade;
    double ProjectGrade;

  public:
    void SetName(string theName)
    {
      Name = theName;
    }
    void SetMidtermGrade(double grade)
    {
      MidtermGrade = grade;
    }
    void SetFinalGrade(double grade)
    {
      FinalGrade = grade;
    }
    void SetProjectGrade(double grade)
    {
      ProjectGrade = grade;
    }
    double ComputeCourseGrade(void)
    {
      double courseGrade = (MidtermGrade + FinalGrade + ProjectGrade) / 3.0;
      return(courseGrade);
    }
    string GetName(void)
    {
      return(Name);
    }
}; // done defining and declaring Student class
```

Dissecting the Class

- Because this is a simple class, we forgo writing a declaration in a header file, and we just declare and define the class all at once in a .cc source file (here, student1.cc).
- The class is declared using the syntax “class classname { … };” Inside the declaration, we place member data and function declarations
- For this example, the member data for this class is found within the “private” section, and the function declarations are found within the “public” section. However, member data and functions can generally be either public or private, as desired. See below for what these keywords mean.

Public and Private

- Member data and/or functions declared "public" are accessible by code that resides outside the class (e.g., in main())
- Member data and/or functions declared "private" may only be used by code contained within the class (e.g., class member functions)
- If "private" stops you from accessing data/functions outside the class, why would you ever want to use it?

Data Encapsulation

One of the key tenents of object-oriented programming is that of “data encapsulation.” This means that (at least some) member data is hidden within a class and is not accessible from outside that class (at least not *directly* accessible). This is considered a good thing because in a large program, another programmer coming in and directly manipulating data in your class might have unexpected side-effects. By requiring other programmers to go through an interface you (the class programmer) provide, you lessen the chance of such side-effects.

Getter/Setter Methods

- Of course, programmers who use your class will want to be able to interact with it and get or set data within it. Hence, the class programmer provides “getter” and “setter” methods for data that a user of the class might need to interact with.
- In our example, we provide a setter method for every piece of member data (conveniently, all the setter functions begin with “Set” so it’s clear what they do). However, in this case we only provide getter functions for the student’s name [GetName()] and the overall course grade [ComputeCourseGrade()]. We could certainly provide getter functions for the other data, too, if we wanted.

Using the Class

- Ok, here's some code in main() that allows us to use the class

```
int main()
{
  // Construct a student object and set some properties
  Student Student1;
  Student1.SetName("John Smith");
  Student1.SetMidtermGrade(80.0);
  cout << "Student " << Student1.GetName() << " has course grade ";
  cout << Student1.ComputeCourseGrade() << "\n";
}
```

Program output:
Student John Smith has course grade 26.6667

Accessing Member Data and Functions

- Notice that once we create a new object of type Student using the syntax “Student student1”, we can access its associated functions (methods) using syntax like “student1.SetName(“John Smith”);” The dot operator comes between the name of the object and the name of the data/method we want to access.
- We could try setting the name directly using something like “student1.Name = “John Smith”;” That would normally work, but not in this case because we declared Name to be Private, meaning that direct access without going through Public functions is impossible. See the next page for a modified version of the program that would allow direct access to the Name variable in the class.

Direct Access to Public Member Data

- Here is a list of modifications to our program that would allow direct accessing of the Name field (see listing student1a.cc). However, keep in mind this kind of direct access is *discouraged* because it breaks data encapsulation!

```
class Student
{
  private:
    double MidtermGrade;
    double FinalGrade;
    double ProjectGrade;

  public:
    string Name;
     …
}; // done defining and declaring Student class

int main()
{
  Student Student1;
  Student1.Name = "John Smith"; // direct access now
  Student1.SetMidtermGrade(80.0);
  cout << "Student " << Student1.Name << " has course grade "; // direct now
  cout << Student1.ComputeCourseGrade() << "\n";
}
```

Access to Classes Using Pointers

- Of course, sometimes we might want to create our new object dynamically using the “new” operator (perhaps we want an array of type Student, for example)
- When we have a pointer to a class, we access data and/or methods using the “->” operator instead of the “.” operator. For example (see listing student1b.cc):

```
int main()
{
  // Construct a student object and set some properties
  Student* Student1 = new Student();  // note parentheses
  Student1->SetName("John Smith");
  Student1->SetMidtermGrade(80.0);
  cout << "Student " << Student1->GetName() << " has course grade ";
  cout << Student1->ComputeCourseGrade() << "\n";
}
```

Uninitialized Data

- Note that we only bothered to set the Midterm exam grade. Perhaps it's early in the semester, or perhaps the student never completed the other two assignments. The overall course grade reflects zeroes for these two assignments and thus yields a overall course grade of 26.6 … unless it doesn't!
- Note that we never initialized the grades for the other assignments to zero. So, on some machines these other grades might have random, nonzero values, leading to trouble computing grades unless all of the grades are set using the setter methods!

Initializing Data

- You might think this would be easy to fix… we could just go into the "private" section defining the member data and change it like so:

```
private:
   string Name;
   double MidtermGrade = 0.0;
   double FinalGrade = 0.0;
   double ProjectGrade = 0.0;
```

- Unfortunately this won't work! We get a warning or error like this:

```
student1.cc:9: error: ISO C++ forbids initialization of member MidtermGrade
student1.cc:9: error: making MidtermGrade static
```

Static member data

When we provide an initialization value, the compiler thinks we want to make this piece of data “static”. For a class, static data is data that is the same for all objects created from the same class (all “instantiations” of the class). That‘s not at all what we want here. There must find another way to initialize the member data when a new instantiation of a class is created. We do this using a “constructor”.

Constructors

- To ensure the member data of a new instantiation of a class object is set properly, we can call a “constructor”. This is a function that’s called automatically every time a new object is made from the class, and it has the same name as the class itself. It does not have a return type. If, as we have been doing so far, we put the definition and declaration in the same place, use this syntax:

```
class Student
{
  public:
    Student()
    {
      // code can go here
    }
```

Constructors

- To separate the definition from the declaration, use this syntax:

```
class Student
{
  public:
    Student(); // constructor declaration
};

// constructor definition
Student::Student()
{
  // constructor code goes here
}
```

- The *scope resolution operator* (::) in the definition shows that the function Student() [constructors are functions with the same name as their class] belongs to class Student. In fact we could use this syntax for any of the other methods of class Student to define them separately from their declaration, e.g., Student::SetName().

Next Iteration of the Student Class

On the following page is our next version of the student class (student2.cc) that uses a constructor to zero out the grades when a new Student object is constructed. We've also decided that the calling program probably only needs to print out the grades, so we've moved the course grade computation and the printing together into a new function, PrintCourseGrade()

```
class Student
{
  private:
    string Name;
    double MidtermGrade;
    double FinalGrade;
    double ProjectGrade;

  public:
    Student()
    {
      MidtermGrade = FinalGrade = ProjectGrade = 0.0; // initialize vars to 0
    }

// setter functions same as before
…

    void PrintCourseGrade(void)
    {
      cout << "Student " << Name << " has course grade ";
      cout << (MidtermGrade + FinalGrade + ProjectGrade) / 3.0 << "\n";
    }
}; // done defining and declaring Student class

int main()
{
  Student Student1;
  Student1.SetName("John Smith");
  Student1.SetMidtermGrade(80.0);
  Student1.PrintCourseGrade();
}
```

More Elaborate Constructors

Our constructor is pretty basic: it just sets the grades to zero when the object is created. But the constructor function, like other functions, can be overloaded. This gives us the option to create an object with certain information specified at the time of creation. For example, we could make a constructor that takes a string argument to automatically set the student's name, like this:

```
Student student1("John Smith");
```

The next page shows how we could implement this.

Constructor with Arguments Example

```
// student3.cc
class Student
{
  // private data as before
  …

  public:
    Student() // default constructor
    {
      MidtermGrade = FinalGrade = ProjectGrade = 0.0;
    }
    Student(string theName) // constructor to set Name
    {
      MidtermGrade = FinalGrade = ProjectGrade = 0.0;
      Name = theName;
    }
    …
}; // done defining and declaring Student class

int main()
{
  Student Student1("John Smith");
  Student1.SetMidtermGrade(80.0);
  Student1.PrintCourseGrade();
}
```

Classes Without a Default Constructor

- In our original listing (student1.cc), we didn't have a constructor. That's fine, if we have absolutely no constructors in our code, the compiler will make a basic default constructor one for us (although it doesn't initialize variables, so it's pretty useless in that regard).
- However, *if we do specify any constructors*, then *the compiler will not make a default constructor*. That means that if we make a constructor that takes arguments, *and* we don't make a default constructor, then objects can only be created with the constructor that takes arguments. Syntax like this won't work:

  Student Student1;

  Indeed, in our example, it does no good to track students who don't at least have a name associated with their record; we'll eliminate our default constructor in our next version of the Student class.

Variable Names in Setter Methods

- We haven't mentioned it until now, but notice that the arguments to all our setter methods have different names than the class member data we're trying to set. Why bother to make the variable names different? After all, normally we can name function arguments anything we want! But using a different name is important…otherwise, we would get nonsense-looking code like this:

```
void SetName(string Name)
  {
    Name = Name;
  }
```

- And indeed, this doesn't work because it's ambiguous. Which "Name" do we mean?

The “this” pointer

- As seen on the last page, we can run into trouble if we use member data variable names for other uses in a class, e.g., as the names of arguments to a setter function. The simplest way to avoid this problem is to use different names for the parameters.
- Another way to avoid this problem is to use the “this” pointer to distinguish between a local variable name (like a function parameter) and a member data variable name. For example, we could write SetName() like this:

```
void SetName(string Name)
  {
    this->Name = Name;
  }
```

- Yet another way to avoid problems with using the same name for parameters and member data is via initialization lists (see below)

Constructors with Default Values

- Just like regular C++ functions can have default values, so can constructors. Let's now suppose we want to keep track of whether a student in the class is a regular student or an auditor, and we require this information as well as their name when we create a new Student object. However, the majority of the students will not be auditors, so we'll make the default value for the auditor argument to be false. The relevant code is on the next page.
- Note: if the class provides a constructor in which all the arguments have default values, then this counts as a default constructor and it would again allow construction of objects without specification of arguments, like Student student1;

```
// student4.cc
class Student
{
  private:
    string Name;
    bool Auditor;
    …
  public:
    Student(string theName, bool isAuditor = false)
    {
      MidtermGrade = FinalGrade = ProjectGrade = 0.0;
      Name = theName;
      Auditor = isAuditor;
    }
    …
    void PrintCourseGrade(void)
    {
      if (Auditor) cout << "Auditor ";
      else cout << "Student ";
      cout << Name << " has course grade ";
      cout << (MidtermGrade + FinalGrade + ProjectGrade) / 3.0 << "\n";
    }
}; // done defining and declaring Student class

int main()
{
  Student Student1("John Smith"); // no 2nd argument given, assumes default
  Student1.SetMidtermGrade(80.0);
  Student1.PrintCourseGrade();
  Student Student2("Jane Doe", true);
  Student2.SetMidtermGrade(100.0);
  Student2.PrintCourseGrade();
}
```

Constructors with Initialization Lists

- This is an alternative way to initialize member data that saves a little typing by allowing us to replace explicit statements setting member data with an implicit initialization. It can also be a way to utilize a base class constructor with particular arguments (more on this later when we discuss inheritance).

Initialization List example

```
// former constructor:
Student(string theName, bool isAuditor = false)
   {
    MidtermGrade = FinalGrade = ProjectGrade = 0.0;
    Name = theName;
    Auditor = isAuditor;
   }

 // initialization list constructor (listing student4a.cc):
Student(string theName, bool isAuditor = false)
        :Name(theName), Auditor(isAuditor)
   {
    MidtermGrade = FinalGrade = ProjectGrade = 0.0;
   }
```

- Note: with an initialization list, we avoid having to name the arguments something different from the member data variable names. We could have used Name and Auditor for the argument names above, and that would also work

Destructors

- Just like a constructor creates/initializes an object, a destructor deletes an object. As with constructors, if one isn't supplied by the programmer, then the compiler supplies a basic one. However, the compiler-supplied destructor does an absolute minimum and is only sufficient for very basic classes that don't do any dynamic memory allocation.
- Let's rewrite our Student class to create dynamically allocated memory, and then use a destructor to free up that memory (with the delete [] operation) when we're done with the object
- The destructor is called when an object goes out of scope (or, if smart pointers are used, when no smart pointer remains that points to the object --- smart pointers are discussed later in the tutorial)

Adding Arrays to the Student Class

- Suppose instead of having member data like MidtermGrade, FinalGrade, ProjectGrade, we just make an array of doubles called Grades. Now, to set a particular grade, we'll call a new function, SetGrade(int whichGrade, double grade) that will set some particular element (whichGrade) of the Grades array to the provided value (grade), i.e.,

```
Grades[whichGrade] = grade;
```

```
// student5.cc

class Student
{
  private:
    string Name;
    bool Auditor;
    double *Grades;

  public:
    // Constructor
    Student(string theName, bool isAuditor = false)
          :Name(theName), Auditor(isAuditor)
    {
      Grades = new double[3];
      for (int i=0; i<3; i++) Grades[i] = 0.0;
    }
    // Destructor
    ~Student()
    {
      delete [] Grades; // delete the dynamically allocated array
    }
    void SetGrade(int whichGrade, double grade)
    {
      Grades[whichGrade] = grade;
    }
    void PrintCourseGrade(void)
    {
      if (Auditor) cout << "Auditor ";
      else cout << "Student ";
      cout << Name << " has course grade ";
      cout << (Grades[0] + Grades[1] + Grades[2]) / 3.0 << "\n";
    }
}; // done defining and declaring Student class

int main()
{
  Student Student1("John Smith"); // no 2nd argument given, assumes default
  Student1.SetGrade(0, 80.0);
  Student1.PrintCourseGrade();
  Student Student2("Jane Doe", true);
  Student2.SetGrade(0, 100.0);
  Student2.PrintCourseGrade();
}
```

Comments about the Destructor

- As we can see from our example, the destructor is called ~Student(). It has no return type (just like a constructor), and it takes no arguments.
- Without the destructor, our allocated array Grades[] would never be de-allocated, and this would eat up more and more memory the more Student objects that were created; this would constitute a “memory leak”