C++ Programming Tutorial Part II: Object-Oriented Programming

C. David Sherrill

Georgia Institute of Technology

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”

Copy Constructors

- A copy constructor is a constructor that creates a new object as a copy of another object
- This is can be a convenient way for the programmer to create multiple instantiations of identical objects (or similar objects --- we could call member functions to modify the copies as desired)
- However, copy constructors are more useful and more necessary than this; recall C++ functions use “pass by value,” in which the *values* of function arguments are passed, not the actual variables themselves. Similarly, if we pass an object as an argument to a function, then we pass a *copy* of the object, not the object itself (unless we use a pointer or reference to it). This means the compiler has to have a way to make a copy of an object. If we haven’t provided a copy constructor, the compiler will generate one automatically for us.

The Problem with Default Copy Constructors

- As you might have guessed by now, the compiler-generated copy constructor isn't all that great. In particular, only a simple "shallow copy" is performed; this means that if a pointer is one of our member data, we only copy the pointer, not the data that is being pointed to
- At first this may seem ok, since the copy object has a pointer to the array, and so does the original: both can access the data
- The problem is that each object might think that it "owns" the array: in particular, if the first object goes out of scope and we call the destructor, it will delete the array --- but now the second object has a pointer that points to invalid memory!
- The solution is as follows: *Any class that points to dynamically allocated memory should provide a copy constructor that performs a "deep copy"* (i.e., dynamically allocated memory is copied, not just the pointers to this memory)

```
// student6.cc [will not work!]

class Student
{
…
  void PrintGrades(void) {
     if (Auditor) cout << "Auditor ";
     else cout << "Student ";
     cout << Name << " grades: ";
     cout << Grades[0] << "," << Grades[1] << "," << Grades[2] << endl;
    }
    void PrintCourseGrade(void)
    {
     if (Auditor) cout << "Auditor ";
     else cout << "Student ";
     cout << Name << " has course grade ";
     cout << (Grades[0] + Grades[1] + Grades[2]) / 3.0 << endl;
    }
}; // done defining and declaring Student class

void PrintStudentGrades(Student Input)
{
  Input.PrintGrades();
  Input.PrintCourseGrade();
}

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

The Problem Manifests Itself

- Listing student6.cc will compile, but when executed it gives an error like this:

```
Student John Smith grades: 80,0,0
Student John Smith has course grade 26.6667
Auditor Jane Doe grades: 100,0,0
Auditor Jane Doe has course grade 33.3333
*** glibc detected *** ./student6: double free
or corruption (fasttop): 0x00000000009ab090
***
```

Analysis of Problem in student6.cc

- We can pass the Student object down to our PrintStudentGrades() function just fine
- However, when we pass Student1 and Student2 into this function in main(), we don't pass the objects themselves, we pass copies
- And since we didn't provide a copy constructor, the compiler makes one for us; the one it makes is poor and uses only shallow copies (pointers to Grades are copied, but not the arrays themselves)
- This doesn't prevent PrintStudentGrades() from printing the correct data
- But when PrintStudentGrades() is done, the local copy of the object, "Input", goes out of scope and we call the destructor
- But our destructor deletes the Grades array! This is a problem because both Input and the object passed into PrintStudentGrades() point to this *same* array! After we execute PrintStudentGrades(Student1), Student1's Grades pointer now points to invalid memory!
- And *this* is a problem because when main() is done, Student1 goes out of scope and we call the destructor… but the Grades array has *already* been deleted! (Same problems happen for Student2)

The Solution: Deep Copy

- We can avoid all these issues if we just copy the dynamically allocated memory, not just the pointers to it
- Listing student6a.cc provides an example of a deep copy constructor
- The syntax for a copy constructor is as follows:

```
Student(const Student& Source)
{
   // copy constructor code goes here
}
```

Fixed with Copy Constructor

```
// student6a.cc

class Student
{
  public:
  …
    // Copy constructor
    Student(const Student& Source)
    {
      cout << "In copy constructor!" << endl;
      Name = Source.Name;
      Auditor = Source.Auditor;
      Grades = new double[3];
      for (int i=0; i<3; i++) Grades[i] = Source.Grades[i];
    }
  …
}; // done defining and declaring Student class

…
```

Output:

In copy constructor!

Auditor grades: 80,0,0

Auditor has course grade 26.6667

In copy constructor!

Auditor grades: 100,0,0

Auditor has course grade 33.3333

Importance of const ref in Copy Constructors

```
Student(const Student& Source)
{
   // copy constructor code goes here
}
```

- Notice that our copy constructor took a const ref as an argument
- const means we aren't modifying the original object
- It's critical that we use a reference to an object as the argument, and not an object itself; if we didn't use a reference, then the copy constructor would be pass-by-value, and the object passed into it would be copied via a shallow copy!

Copy by Assignment

- So far we have been considering copy constructors that might be called when an object is passed as an argument to a function
- We can also create one object from another via an assignment (=) operator; operators can be overloaded to work with objects
- For exactly the same reasons that the copy constructor needs to make deep copies of arrays, so should copies via the assignment operator
- More on operator overloading later, but for now, analogous to the syntax of our copy constructor, the copy assignment operator should be defined as below.
- Copy by assignment has a couple of subtle features (e.g., we don't want to copy an object if we're assigning it back to itself), so we'll postpone further discussion of these for now.

```
Student& Student::operator= (const Student& Source)
{
   // copy assignment operator code here
}
```

Summary of Precautions for Classes containing Pointers

- If a class contains a raw pointer, you have to exercise special caution when copying this class via a copy constructor or an assignment operator
- If you don't perform deep copies of the data being pointed to, bad things can happen; for example, if two objects contain a member that is a pointer, then both objects might try to free that data when the object destructors are called: the first free will succeed, and the second one will fail
- Avoid raw pointers in classes, or write custom copy constructors and assignment operators that perform deep copies; these functions should take const references to the object being copied
- Replacing raw pointer member data with smart pointers is an alternative solution (to be discussed later); also, use C++ strings instead of C-style char* arrays for member data

Intro to C++11 Move Constructors

- Using deep copies if a class contains raw pointers is good, but it can cause unexpected performance penalties
- This happens if a very temporary copy of the object has to be created because of how the object is used
- Sometimes the compiler sees what's happening and optimizes the problem away; if not, it's nice to improve performance with a "move constructor"
- A move constructor is a constructor that can be invoked when *temporary* instances of an object are passed as parameters; in cases like this, we can simply move the data from the temporary object to the permanent one
- This is a new feature of C++11
- More on this later

Classes that Don’t Allow Copying

- For some classes, a copy operation might not make sense
- We can deny copying if we declare a private copy constructor and a private copy assignment operator; they don’t have to actually be defined because they can’t be used (since they’re private) --- declaring them just prevents the compiler from supplying default functions

```
Class X
{
private:
   X(const X&); // private copy constructor
   X& operator= (const X&); // private copy assignment operator
   …
}
```

Singleton Class

- Taking the ideas of the previous page a little further, we might have a class that we want only one instance of (in the past page we prohibited copying, but didn't prohibit creation of multiple instances)
- We can prohibit creation of multiple instances by also making the constructor private
- But by itself, this would make it impossible to create any instances of the class
- So, we make a public function that is able to create an instance, and we use "static" keywords to (1) make the function shared among all instances of the class, and (2) make a local variable in this function that retains its value between function calls (see next page)
- By the way, member data declared as "static" will be shared among all instances of the class

Singleton Class Example

```
Class X
{
private:
   X() {}; // private constructor (means we can't construct from outside)
   X(const X&); // private copy constructor
   const X& operator= (const X&); // private copy assignment operator
   …
public:
   static X& GetInstance() { // because static, can invoke w/out an object!
      static X OnlyInstance;
      return OnlyInstance;
   }
   … // other functions here
} ;
int main()
{
   X& OnlyX = X::GetInstance();
   // perform operations on OnlyX
}
```

Class only Creatable on the Heap

- If we have a class with very large datastructures, we want to make sure it is created using dynamic memory allocation (i.e., on the heap, like Student* A = new Student()) and not on the stack (like Student A)
- We can accomplish this my making the destructor private; the compiler will realize that objects on the stack need to be destroyed when they go out of scope, and that this can't happen if the destructor is private
- Of course, this also means one can't delete objects on the heap, either --- at least not in the usual way. We need the static keyword and a helper function, analogous to how we implemented singleton classes

Heap-Only Class Example

```
Class X
{
private:
   ~X(); // private destructor
public:
   static void DestroyInstance(X* instance)
   {
      // this static member can get at the private destructor
       delete instance;
   }
};

int main() {
   X* pX = new X();
   X::DestroyInstance(pX); // do this instead of delete pX
}
```

The this Pointer and static Functions

- We saw before in this chapter that the “this” pointer is implicitly passed to all member functions of a class and is useful to clarify whether an operation is on the current class object or on one passed as a parameter
- “this” is **not** implicitly passed to member functions declared as static, since static functions are shared among all members of a class
- Thus, in static functions, we do not have access to the “this” pointer

sizeof() a Class

- Just like we can use sizeof() to get the number of bytes taken up by a plain old data type, we can also use it on a class
- When used on a class, sizeof() returns the number of bytes for the basic member data, *not including* the size of any dynamically allocated memory pointed to by the class (and also not including any space for member functions)

Friend Functions and Classes

- We've seen that member functions and member data listed as "private" are not accessible outside the class
- We can make special exceptions using the "friend" keyword: a friend function can access private class data, and so can a friend class
- We declare friend functions or classes like this:

```
Class X
{
private:
   friend void FriendlyFunction(…);  // this function is a friend
   friend class FriendlyClass;          // this entire class is a friend
   …
};
```

Friend Function Example

```
// student7.cc
class Student
{
  private:
    string Name;
    bool Auditor;
    double *Grades;

  public:
    ....
    // friend declaration gives function outside class access to private bits
    friend void PrintStudentGrades(const Student& Input);
}; // done defining and declaring Student class

void PrintStudentGrades(const Student& Input)
{
  if (Input.Auditor) cout << "Auditor ";
  else cout << "Student ";
  cout << Input.Name << " grades: ";
  cout << Input.Grades[0] << "," << Input.Grades[1] << "," << Input.Grades[2];
  cout << ". Course grade: ";
  cout << (Input.Grades[0] + Input.Grades[1] + Input.Grades[2]) / 3.0 << endl;
}

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

Friend Class Example

```
// friend8.cc
class Student
{
  private:
    string Name;
    bool Auditor;
    double *Grades;

  public:
    ...
    // friend declaration gives function outside class access to private bits
    friend class PrintHelper;
}; // done defining and declaring Student class

class PrintHelper
{
  public:
    void PrintStudentGrades(const Student& Input)
    {
      if (Input.Auditor) cout << "Auditor ";
      else cout << "Student ";
      cout << Input.Name << " grades: ";
      cout << Input.Grades[0] << "," << Input.Grades[1];
      cout << "," << Input.Grades[2];
      cout << ". Course grade: ";
      cout << (Input.Grades[0] + Input.Grades[1] + Input.Grades[2]) / 3.0;
      cout << endl;
    }
};

int main()
{
  Student Student1("John Smith");
  Student1.SetGrade(0, 80.0);
  PrintHelper X;
  X.PrintStudentGrades(Student1);
}
```

const Member Functions

- In Part 1, we saw various uses of the const keyword: we can have constants (e.g., const int), constant pointers to variables (e.g., int* const p), constant pointers to contants (e.g., const int* const p), or references to constants (int& const p or equivalently const int& p, which simply means even though we have a ref we promise not to modify p)
- When dealing with classes, we can also have constant functions. These are functions that are guaranteed to not modify the object. They are specified like this:
  int MyFunction([args]) const { … }
  The const needs to be added to the function declaration *and* the definition
- Several of our previous and future examples could have been modified to specify that the member function was constant

Chapter 10: Inheritance

- Defining inheritance
- Base classes and derived classes
- Protected members
- Public, Protected, and Private inheritance
- Derived classes hiding base class methods
- Invoking base class methods in derived classes
- Slicing of derived classes
- Multiple inheritance

Inheritance

- In this chapter, we discuss *inheritance*, which is a powerful feature of object-oriented programming
- Inheritance allows us to specify more generic behavior for some objects, and more specific behavior for other objects (for example, squares have all the properties of rectangles, but they also have additional special properties as well)

Polygon Class Example

```
// polygon.cc
#include <iostream>
#include <string>
using namespace std;

class Polygon
{
  public:
    int NumSides;
    void Draw(void) {
      cout << "Drawing polygon with " << NumSides << " sides." << endl;
    }
    Polygon(int sides) {
      NumSides = sides;
    }
};

int main()
{
  Polygon Triangle(3);
  Polygon Square(4);
  Triangle.Draw();
  Square.Draw();
}
```

Polygon Derived Classes

- You can imagine that it might make sense to create Triangle, Square, etc., classes that behave as typical polygons but also know about the special properties of triangles, squares, etc.
- We can do this through inheritance. We will create Triangle and Square classes that inherit from the Polygon class.
- In this example, Polygon is the *base class*, and Triangle and Square are the *derived classes*
- Alternatively, we can call Polygon the *superclass*, and we can call Triangle and Square the *subclasses*

Inheritance Syntax

- We declare and define the Triangle and Square classes just like we do for any other class; the only difference inheritance makes is that we add some extra bits to the class declaration line, like this:

```
class Triangle: public Polygon
{
        …
};
```

This specifies that class Triangle inherits from class Polygon (the “public” is an “access specifier” which could also be “protected” or “private”, but usually it’s public … we’ll explain this below)

```
// polygon2.cc

class Polygon
{
  public:
    int NumSides;
    void Draw(void) {
      cout << "Drawing polygon with " << NumSides << " sides." << endl;
    }
    Polygon(int sides) {
      NumSides = sides;
    }
};

class Square: public Polygon
{
  public:
    Square() {                          // problem here … see next page
      NumSides = 4;
    }
};

class Triangle: public Polygon
{
  public:
    Triangle() {                        // problem here … see next page
      NumSides = 3;
    }
};

int main()
{
  Triangle MyTriangle;
  Square MySquare;
  Polygon MyPentagon(5);
  MyTriangle.Draw();
  MySquare.Draw();
  MyPentagon.Draw();
}
```

Base Class Initialization

- The listing on the previous page won't work; we get an error message like the following:

```
polygon2.cc: In constructor Square::Square():
polygon2.cc:21: error: no matching function for call to  Polygon::Polygon()
polygon2.cc:13: note: candidates are: Polygon::Polygon(int)
polygon2.cc:7: note:                 Polygon::Polygon(const Polygon&)
polygon2.cc: In constructor Triangle::Triangle():
polygon2.cc:29: error: no matching function for call to Polygon::Polygon()
polygon2.cc:13: note: candidates are: Polygon::Polygon(int)
polygon2.cc:7: note:                 Polygon::Polygon(const Polygon&)
```

- The problem is that derived classes always call the base class constructor first, and we don't have a default constructor available to match the default constructor for the derived classes

Base Class Initialization

- We have a default constructor for a Triangle, but that default constructor automatically tries to call the default constructor for Polygon (the base class), and there isn't one
- We need to explicitly call an allowed base class constructor; we can do that if we just pass along the number of sides, like this:

```
class Square: public Polygon
{
  public:
    Square(): Polygon(4) { // Call base class constructor with required args
      cout << "In Square constructor" << endl;
    }
};
```

```
// polygon2a.cc
…
class Polygon
{
  public:
    int NumSides;
    void Draw(void) {
      cout << "Drawing polygon with " << NumSides << " sides." << endl;
    }
    Polygon(int sides) {
      NumSides = sides;
      cout << "In Polygon constructor with " << NumSides << " sides" << endl;
    }
};

class Square: public Polygon
{
  public:
    Square(): Polygon(4) { // Call base class constructor with required args
      cout << "In Square constructor" << endl;
    }
};

class Triangle: public Polygon
{
  public:
    Triangle(): Polygon(3) { // Call base class constructor with required args
      cout << "In Triangle constructor" << endl;
    }
};

int main()
{
  Triangle MyTriangle;
  Square MySquare;
  Polygon MyPentagon(5);
  MyTriangle.Draw();
  MySquare.Draw();
  MyPentagon.Draw();
}
```

Program output:

```
In Polygon constructor with 3 sides
In Triangle constructor
In Polygon constructor with 4 sides
In Square constructor
In Polygon constructor with 5 sides
Drawing polygon with 3 sides.
Drawing polygon with 4 sides.
Drawing polygon with 5 sides.
```

Constructor ordering

- Notice that when a Triangle or Square is constructed, the base class constructor is called first, then the constructor of the derived class
- Destructors are called in the opposite order (derived class first, then base class)

Protecting the Data

- In our current version of the polygons example, there's nothing to stop us from doing something like this:

```
int main()
{
  Triangle MyTriangle;
  Polygon MyPentagon(5);
  MyTriangle.NumSides = 4;
  MyTriangle.Draw();
  MyPentagon.Draw();
}
```

```
In Polygon constructor with 3 sides
In Triangle constructor
In Polygon constructor with 5 sides
Drawing polygon with 4 sides.
Drawing polygon with 5 sides.
```

- This seems odd to have a triangle with a variable number of sides; how can we prevent users from doing this?

Protecting the Class Data

- We saw in the last chapter that member data of a class can be public (it can be accessed outside the class) or private (it cannot be accessed outside the class)
- This gives us a good solution --- simply make the number of sides in the polygon (NumSides) a private member datum

```
// polygon3a.cc

class Polygon
{
  private:
    int NumSides; // outsiders shouldn't be able to change this
  public:
    void Draw(void) {
      cout << "Drawing polygon with " << NumSides << " sides." << endl;
    }
    Polygon(int sides) {
      NumSides = sides;
      cout << "In Polygon constructor with " << NumSides << " sides" << endl;
    }
};

class Square: public Polygon
{
  public:
    Square(): Polygon(4) { // Call base class constructor with required args
      cout << "In Square constructor" << endl;
    }
};

class Triangle: public Polygon
{
  public:
    Triangle(): Polygon(3) { // Call base class constructor with required args
      cout << "In Triangle constructor" << endl;
    }
};

int main()
{
  Triangle MyTriangle;
  Polygon MyPentagon(5);
  //MyTriangle.NumSides = 4; // Now we can't do this...good!
  MyTriangle.Draw();
  MyPentagon.Draw();
}
```

Problems Inheriting Private Members

- The previous example solves our problem of wanting to deny outsiders the ability to change the number of sides in one of our polygons
- However, making NumSides a private member of Polygon has consequences we might not have expected
- In particular, *derived classes do not have access to private members of a base class*
- Let's illustrate this with an example

```
// polygon3b.cc
#include <iostream>
#include <string>
using namespace std;

class Polygon
{
  private:
    int NumSides; // outsiders shouldn't be able to change this
  public:
    void Draw(void) {
      cout << "Drawing polygon with " << NumSides << " sides." << endl;
    }
    Polygon(int sides) {
      NumSides = sides;
      cout << "In Polygon constructor with " << NumSides << " sides" << endl;
    }
};

class Triangle: public Polygon
{
  public:
    Triangle(): Polygon(3) { // Call base class constructor with required args
      cout << "In Triangle constructor" << endl;
    }
    void Report(void) {
      cout << "I'm a triangle, I have " << NumSides << " sides." << endl;
    }
};

int main()
{
  Triangle MyTriangle;
  MyTriangle.Report();
}
```

Compiling this program produces errors:

polygon3b.cc: In member function void Triangle::Report():
polygon3b.cc:9: error: int Polygon::NumSides is private
polygon3b.cc:27: error: within this context

Protected Member Data

- How can we fix this? We want to protect NumSides like a private member, but we still want our derived classes to be able to access it
- This is precisely what “protected” class members do: they block access from outside the object, but they still allow it for derived classes
- So, we just need to specify that NumSides is protected.

```
class Polygon
{
  protected:
    int NumSides;
…
```

```
// polygon3c.cc

class Polygon
{
  protected:
    int NumSides; // outsiders shouldn't be able to change this
                  // but perhaps derived classes should be able to access it
  public:
    void Draw(void) {
      cout << "Drawing polygon with " << NumSides << " sides." << endl;
    }
    Polygon(int sides) {
      NumSides = sides;
      cout << "In Polygon constructor with " << NumSides << " sides" << endl;
    }
};

class Triangle: public Polygon
{
  public:
    Triangle(): Polygon(3) { // Call base class constructor with required args
      cout << "In Triangle constructor" << endl;
    }
    void Report(void) {
      cout << "I'm a triangle, I have " << NumSides << " sides." << endl;
    }
};

int main()
{
  Triangle MyTriangle;
  //MyTriangle.NumSides = 4; // this won't work
  MyTriangle.Report();
}
```

Program output:

In Polygon constructor with 3 sides
In Triangle constructor
I'm a triangle, I have 3 sides.

Derived Classes of Derived Classes

- Triangle is a derived class from base class Polygon
- What if we derived a new class from Triangle? Say, RightTriangle? Would it also have access to NumSides if it was specified in Polygon as a protected member?
- Yes! See the next example.

```
// Listing polygon3d.cc
[class Polygon as before, with NumSides declared as protected]

class Triangle: public Polygon
{
  public:
    Triangle(): Polygon(3) { // Call base class constructor with required args
      cout << "In Triangle constructor" << endl;
    }
    void Report(void) {
      cout << "I'm a triangle, I have " << NumSides << " sides." << endl;
    }
};

class RightTriangle: public Triangle
{
  public:
    RightTriangle(): Triangle() {
      cout << "In RightTriangle constructor" << endl;
    }
    void Report(void) {
      cout << "I'm a right triangle, I have " << NumSides << " sides." << endl;
    }
};

int main()
{
  RightTriangle MyRightTriangle;
  MyRightTriangle.Report();
}
```

This works! Program output:

```
In Polygon constructor with 3 sides
In Triangle constructor
In RightTriangle constructor
I'm a right triangle, I have 3 sides.
```

Summary of Accessibility of Members of Base Classes from Derived Classes

- *Public* members (data or functions) of a base class *are* available to a derived class *and* through objects of that derived class (e.g., ns = MyPolygon.NumSides)
- *Private* members of a base class *are not* available to a derived class *or* through objects of that derived class
- *Protected* members of a base class *are* available to a derived class but *are not* available through objects of that derived class

Private Inheritance

- So far, we have assumed “public” inheritance, e.g., class Derived: public Base // public inheritance
- There also exists a “private” inheritance, e.g., class Derived: private Base // private inheritance
- If we use “private inheritance,” then *users* of a derived class have *no* direct access to members of the parent class --- data or methods --- even if those members were declared public in the base class
- Thus, the “access specifier” in an inheritance declaration can further limit the access to the base class from an object of a derived class

```
// polygon4.cc
…

class Polygon
{
  public:
    int NumSides;
    void Draw(void) {
      cout << "Drawing polygon with " << NumSides << " sides." << endl;
    }
    Polygon(int sides) {
      NumSides = sides;
      cout << "In Polygon constructor with " << NumSides << " sides" << endl;
    }
};

class Triangle: private Polygon
{
  public:
    Triangle(): Polygon(3) { // Call base class constructor with required args
      cout << "In Triangle constructor" << endl;
    }
};

int main()
{
  Triangle MyTriangle;
  Polygon MyPentagon(5);
  // MyTriangle.NumSides = 4; // won't work if we uncomment
  // MyTriangle.Draw(); // this won't work either!
  // MyPentagon.Draw(); // or this!
}
```

- Succeeds in keeping NumSides hidden from users
- But also makes the public member functions of Polygon inaccessible!

Private Inheritance Problems

- As seen in the previous example, private inheritance can keep member data of a parent class safe from outsiders who have an object of a derived class
- But it also makes any public member functions of the base class inaccessible except through member functions of the derived class!
- This isn't usually the behavior we want; it only makes sense when the derived class is considered to have an instance of the base class within it ("has a" relationship), not when the derived class is a specific example of the base class ("is a" relationship)
- For example, a triangle "is a" polygon; it should be able to do anything a generic polygon can do, so it should be able to access the public members of polygon. Public inheritance is appropriate.
- On the other hand, a college course "has a" gradebook, so private inheritance of a gradebook class by a college course parent class might be appropriate

Private Inheritance Problems

- A further problem with private inheritance is that it effectively allows the derived class only a single instance of the parent class (e.g., the college course “has a” gradebook). What if we needed the derived class to have multiple instances of a parent class (e.g., maybe we decide to keep two gradebooks, one of the original scores and one with curved scores)?
- Often it’s better to simply make the derived class include one or more actual objects of the base class instead of using private inheritance, e.g.,

```
class CollegeCourse
{
  GradeBook MyGradeBook;
   …
};
```

Protected Inheritance

- Protected inheritance is much like private inheritance
- Like private inheritance, it assumes the derived class “has a” instance of the base class
- Like private inheritance, functions of the derived class can access all public and protected members of the base class
- Like private inheritance, *users* of an object of the derived class cannot access any of the members of the base class (e.g., Triangle.NumSides)
- *Unlike* private inheritance, protected inheritance allows classes that *derive from the derived class* to access public members of the original base class
- Protected inheritance suffers from all the same problems we just enumerated for private inheritance; making the derived class contain objects of the base class is often a better solution than protected inheritance

Overriding Base Class Methods

- Sometimes we want a derived class to override a generic member function of the base class to do something more specialized in the derived class
- We can do this by simply specifying the same function (with the same arguments) for the derived class
- Member functions will be called from the derived class first (if available), and only if absent from the derived class will the ones from the base class be used

```
// polygon5.cc
#include <iostream>
#include <string>
using namespace std;

class Polygon
{
  public:
    int NumSides;
    void Draw(void) {
      cout << "Drawing polygon with " << NumSides << " sides." << endl;
    }
    Polygon(int sides) {
      NumSides = sides;
      cout << "In Polygon constructor with " << NumSides << " sides" << endl;
    }
};

class Triangle: public Polygon
{
  public:
    Triangle(): Polygon(3) { // Call base class constructor with required args
      cout << "In Triangle constructor" << endl;
    }
    void Draw(void) { // this routine overrides the base class Draw() routine
      cout << "Using special triangle drawing routine" << endl;
    }
};

int main()
{
  Triangle MyTriangle;
  Polygon MyPentagon(5);
  MyTriangle.Draw(); // uses specialized Triangle::Draw() routine
  MyPentagon.Draw(); // uses generic Polygon::Draw() routine
}
```

Program output:

```
In Polygon constructor with 3 sides
In Triangle constructor
In Polygon constructor with 5 sides
Using special triangle drawing routine
Drawing polygon with 5 sides.
```

Unintended Hiding of Base Class Methods

- By a quirk of C++, if you override one form of a base class method, you're considered to have overridden all forms of that method
- That is, if a base class has a function with multiple different signatures, like this:

  ```
  void DoSomething(void) {…}
  void DoSomething(string message) {…}
  void DoSomething(int size) {…}
  ```

  and if we override the function in a derived class like this:

  ```
  void DoSomething(void) { // new code here …}
  ```

  then the other two signatures from the base class (the ones taking a string or an integer as an argument) become "hidden" and no longer directly accessible from the derived class (even if they are declared public in the base class)

```
// polygon6.cc
…

class Polygon
{
  public:
    int NumSides;
    void Draw(void) {
      cout << "Drawing polygon with " << NumSides << " sides." << endl;
    }
    void Draw(string message) {
      cout << "Drawing polygon with " << NumSides << " sides" << endl;
      cout << "and writing " << message << " inside" << endl;
    }
    Polygon(int sides) {
      NumSides = sides;
    }
};

class Triangle: public Polygon
{
  public:
    Triangle(): Polygon(3) { // Call base class constructor with required args
    }
    void Draw(void) { // This routine overrides the base class Draw() routine
      cout << "Using special triangle drawing routine" << endl;
    }
};

int main()
{
  Triangle MyTriangle;
  Polygon MyPentagon(5);
  MyPentagon.Draw("Hello"); // uses generic Polygon::Draw() routine
  // following line won't work because we overrode Draw(void)
  // and this hides all other forms of Draw() in base class from derived class
  MyTriangle.Draw("Hi");
}
```

Resolving Hidden Base Class Methods

- How can we get around this problem?
- One solution is to override void Draw(string message) also
- But suppose we're actually happy with the generic routine from the base class when there's a message to print; overriding just needlessly duplicates code
- We can specify explicitly that we want to use the base class Draw(string) method like this:

```
myTriangle.Pentagon::Draw("Hi");
```

```
// polygon6a.cc
…
class Polygon
{
  public:
    int NumSides;
    void Draw(void) {
      cout << "Drawing polygon with " << NumSides << " sides." << endl;
    }
    void Draw(string message) {
      cout << "Drawing polygon with " << NumSides << " sides" << endl;
      cout << "and writing " << message << " inside" << endl;
    }
    Polygon(int sides) {
      NumSides = sides;
    }
};

class Triangle: public Polygon
{
  public:
    Triangle(): Polygon(3) { // Call base class constructor with required args
    }
    void Draw(void) { // This routine overrides the base class Draw() routine
      cout << "Using special triangle drawing routine" << endl;
    }
};

int main()
{
  Triangle MyTriangle;
  Polygon MyPentagon(5);
  MyPentagon.Draw("Hello"); // uses generic Polygon::Draw() routine
  MyTriangle.Polygon::Draw("Hi"); // now this will work
}
```

Program output:

Drawing polygon with 5 sides
and writing Hello inside
Drawing polygon with 3 sides
and writing Hi inside

Alternative Solution to Hidden Base Class Methods

- The previous solution works fine, but it seems awkward for the user of the class to have to remember they have to call
  MyTriangle.Polygon::Draw("Hi");
  instead of
  MyTriangle.Draw(“Hi”);
- Better to handle this in the derived class definition. There’s a way to do this *without* having to cut and paste the base class method into the derived class.

```
// polygon6b.cc

class Polygon
{
  public:
    int NumSides;
    void Draw(void) {
      cout << "Drawing polygon with " << NumSides << " sides." << endl;
    }
    void Draw(string message) {
      cout << "Drawing polygon with " << NumSides << " sides" << endl;
      cout << "and writing " << message << " inside" << endl;
    }
    Polygon(int sides) {
      NumSides = sides;
    }
};

class Triangle: public Polygon
{
  public:
    Triangle(): Polygon(3) { // Call base class constructor with required args
    }
    void Draw(void) { // This routine overrides the base class Draw() routine
      cout << "Using special triangle drawing routine" << endl;
    }
    void Draw(string message) { // happy with base class method, just call it!
      Polygon::Draw(message);
    }
};

int main()
{
  Triangle MyTriangle;
  Polygon MyPentagon(5);
  MyPentagon.Draw("Hello"); // uses generic Polygon::Draw() routine
  MyTriangle.Draw("Hi");        // simpler syntax here now
}
```

Program output:

```
Drawing polygon with 5 sides
and writing Hello inside
Drawing polygon with 3 sides
and writing Hi inside
```

Slicing Problems

- What happens if we have a function expecting an object of one type, and we pass it an object of a derived class type? That is,

```
void DoSomething(Base p);
…
Derived myDerived;
DoSomething(myDerived);
```

- Similarly, what if we tried this?

```
Derived myDerived;
Base myBase = myDerived;
```

- This is part of a larger discussion, but for now let's note that in code like this, the object of type Derived has to be copied (explicitly or implicitly by a copy constructor or an assignment operator) into an object of type Base
- When this happens, the compiler only copies the part of the Derived object that corresponds to class Base. This is called slicing and is not usually what the programmer wants to happen.
- The simplest way to avoid slicing is to avoid passing by value; pass by pointers or references to the base class (this also avoids issues with copy constructors and assignment operators we've previously discussed)

Multiple Inheritance

- C++ (unlike Java) allows multiple inheritance; that is, a derived class can have more than one parent class
- The syntax for multiple inheritance is as follows:

```
class Derived: access-specifier Base1, access-specifier Base 2 [, etc.]
{
        // declare class members here
};
```

```
// tank.cc
#include <iostream>
#include <string>
using namespace std;

class Vehicle
{
  public:
    void Move(void) {
      cout << "Vehicle is moving!" << endl;
    }
};

class Weapon
{
  public:
    void Fire(void) {
      cout << "Weapon is firing!" << endl;
    }
};

class Tank: public Vehicle, public Weapon
{
};

int main()
{
  Tank MyTank;
  MyTank.Move();
  MyTank.Fire();
}
```

Program output:
Vehicle is moving!
Weapon is firing!

Chapter 11: Polymorphism

- Introducing Polymorphism
- Virtual functions
- Pure virtual functions and abstract base classes
- The “diamond problem” and virtual inheritance

Introducing Polymorphism

- Polymorphism means “many forms”. In object-oriented programming, it has to do with the fact that a derived class can behave as its own type or as the type of its base class(es)
- This is a good thing --- we can treat a collection of objects with a common base class in a generic way according to the rules of the base class
- But this also introduces a problem --- there can be ambiguities about when we want an object to behave generically according to its base class, and when we want it to behave specifically according to its derived class

```
// polygon7.cc

class Polygon
{
  public:
    int NumSides;
    void Draw(void) {
      cout << "Drawing polygon with " << NumSides << " sides." << endl;
    }
    Polygon(int sides) {
      NumSides = sides;
    }
};

class Triangle: public Polygon
{
  public:
    Triangle(): Polygon(3) {
    }
    void Draw(void) { // this won't get used in this example
      cout << "Using special triangle drawing routine" << endl;
    }
};

void DrawPolygon(Polygon& poly) {
  poly.Draw();
}

int main()
{
  Polygon MyPentagon(5);
  Triangle MyTriangle;
  DrawPolygon(MyPentagon);
  // Since we pass MyTriangle as a generic Polygon to DrawPolygon,
  // we wind up drawing with the generic Polygon::Draw() not the
  // specialized Triangle::Draw() like you might expect!
  DrawPolygon(MyTriangle);
}
```

Program output:

Drawing polygon with 5 sides.

Drawing polygon with 3 sides.

Ensuring Use of the Specialized Functions

- In the previous example, we used a helper function, DrawPolygon(), to draw our polygons … but because it accepts generic Polygons as inputs, it calls the generic Polygon::Draw() function
- This is a shame because we have a specialized drawing routine for triangles that we probably want to use!
- We can explain to the compiler that we want to use the more specific functions from the derived class when they override functions from the base class
- We do this by declaring the base class method as a "virtual" function … this means it will only be used if no more specific function from a derived class is available (even when we refer to the object as an object of the base class type)

Virtual Functions

- To declare a virtual function, we just add "virtual" to the beginning:

```
class Base {
    virtual ReturnType MyFunction (params);
};
class Derived {
    ReturnType MyFunction (params);
}
```

```
// polygon8.cc

class Polygon
{
  public:
    int NumSides;
    virtual void Draw(void) {
      cout << "Drawing polygon with " << NumSides << " sides." << endl;
    }
    Polygon(int sides) {
      NumSides = sides;
    }
};

class Triangle: public Polygon
{
  public:
    Triangle(): Polygon(3) {
    }
    void Draw(void) { // now this will get used
      cout << "Using special triangle drawing routine" << endl;
    }
};

void DrawPolygon(Polygon& poly) {
  poly.Draw();
}

int main()
{
  Polygon MyPentagon(5);
  Triangle MyTriangle;
  DrawPolygon(MyPentagon);
  // Now thanks to virtual functions, the correct (more specialized)
  // version of the Draw() routine will be used, i.e., Triangle::Draw()
  DrawPolygon(MyTriangle);
}
```

Program output:
Drawing polygon with 5 sides.
Using special triangle drawing routine

The Need for Virtual Destructors

- A more dangerous situation when we need to call the right function for a derived class is when we need to call a destructor
- If we are treating an object generically and call the generic (parent class) destructor, we might not free up all the memory we're supposed to --- a derived class should always be calling its own destructor
- The next example shows an example of a derived class (Triangle) destructor not being called

```
// polygon9.cc

class Polygon
{
  public:
    int NumSides;
    Polygon(int sides) {
      cout << "Creating a Polygon" << endl;
      NumSides = sides;
    }
    ~Polygon() {
      cout << "Destroyed Polygon" << endl;
    }
};

class Triangle: public Polygon
{
  public:
    Triangle(): Polygon(3) {
      cout << "Creating a Triangle" << endl;
    }
    ~Triangle() {
      cout << "Destroyed Triangle" << endl;
    }
};

void DeletePolygon(Polygon* poly) {
  delete poly;
}

int main()
{
  cout << "About to create a Triangle on the heap" << endl;
  Triangle* pTriangle = new Triangle;
  cout << "About to destroy the Triangle on the heap" << endl;
  DeletePolygon(pTriangle);

  cout << "About to create a Triangle on the stack" << endl;
  Triangle MyTriangle;
  cout << "Triangle on stack about to go out of scope" << endl;
  return 0;
}
```

Program output:

```
About to create a Triangle on the heap
Creating a Polygon
Creating a Triangle
About to destroy the Triangle on the heap
Destroyed Polygon
About to create a Triangle on the stack
Creating a Polygon
Creating a Triangle
Triangle on stack about to go out of scope
Destroyed Triangle
Destroyed Polygon
```

Missing “Destroyed Triangle”!

- Recall that when constructing a derived class, the base class constructor is supposed to be called first, then the derived class constructor
- Destructors should be called in the reverse order of constructors

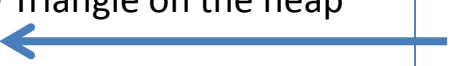
Virtual Destructors

- We know how to solve this problem now… just make the destructor of the base class virtual, like this:
  virtual ~Polygon() { … }
- See the next listing for a corrected version of our example

```
// polygon9a.cc

class Polygon
{
  public:
    int NumSides;
    Polygon(int sides) {
      cout << "Creating a Polygon" << endl;
      NumSides = sides;
    }
    virtual ~Polygon() {
      cout << "Destroyed Polygon" << endl;
    }
};

class Triangle: public Polygon
{
  public:
    Triangle(): Polygon(3) {
      cout << "Creating a Triangle" << endl;
    }
    ~Triangle() {
      cout << "Destroyed Triangle" << endl;
    }
};

void DeletePolygon(Polygon* poly) {
  delete poly;
}

int main()
{
  cout << "About to create a Triangle on the heap" << endl;
  Triangle* pTriangle = new Triangle;
  cout << "About to destroy the Triangle on the heap" << endl;
  DeletePolygon(pTriangle);

  cout << "About to create a Triangle on the stack" << endl;
  Triangle MyTriangle;
  cout << "Triangle on stack about to go out of scope" << endl;
  return 0;
}
```

Program output:

```
About to create a Triangle on the heap
Creating a Polygon
Creating a Triangle
About to destroy the Triangle on the heap
Destroyed Triangle
Destroyed Polygon
About to create a Triangle on the stack
Creating a Polygon
Creating a Triangle
Triangle on stack about to go out of scope
Destroyed Triangle
Destroyed Polygon
```



Got it now!

Always Supply Virtual Destructors

- To avoid potential memory leaks, etc., you should always supply your base classes virtual destructors

The Virtual Function Table

- Classes use a “virtual function table” to keep track of what function they’re supposed to be using in different contexts
- For each class, this is just a list of function pointers, pointing to the functions that the class is supposed to be using (base class functions or derived class functions)
- This only applies to functions that were declared virtual in the base class
- This adds a little extra memory overhead (not much) to classes that have virtual functions and to their derived classes

Run Time Type Identification (RTTI)

- Occasionally the programmer also wants to know whether an object of a generic class is really of that class, or whether it actually belongs to a derived class
- This can be done using dynamic_cast to see if a pointer of type Base* is really of type Derived*
- More on this later
- A dynamic cast is sometimes considered a poor programming practice and a sign that the class hierarchy wasn't designed optimally

Pure Virtual Functions and Abstract Base Classes

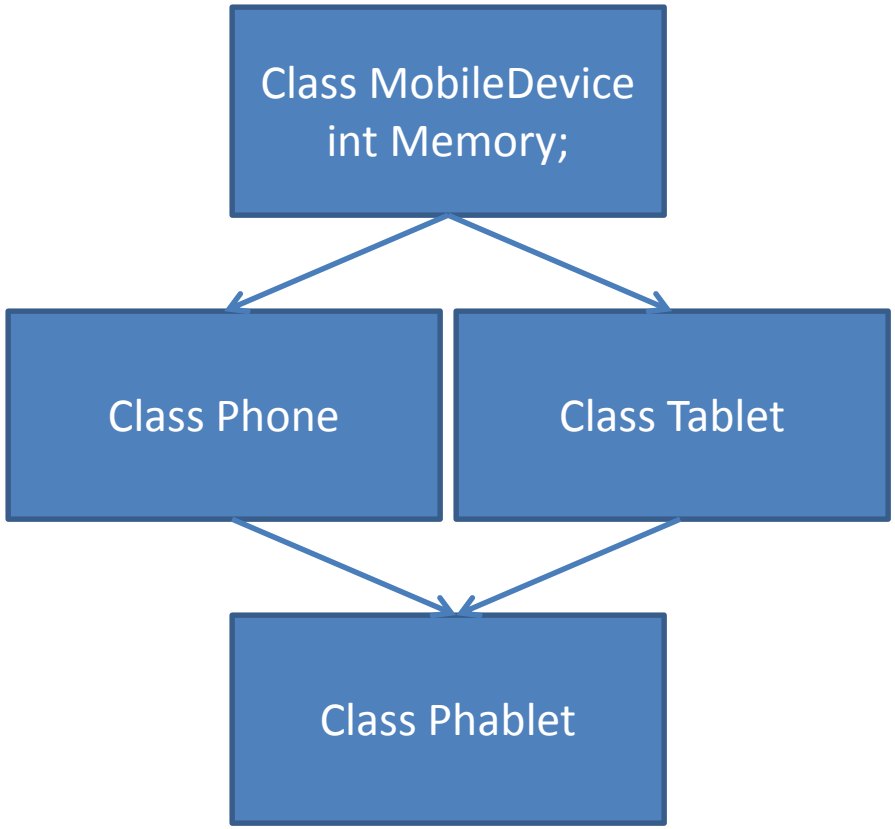
- Sometimes we might want to ensure that every derived class has its own special version of some function (i.e., we will never want to call a generic version of the function from the base class)
- We can ensure this happens by declaring the base class function to be "pure virtual", like this:

```
class Base
{
  public:
    virtual void SomeFunction() = 0; // pure virtual
};
```

- A pure virtual function can't be used … it's just a signal that any derived class must override this function
- A pure virtual function means that the base class can never be instantiated (because it would have no working definition of this pure virtual function) --- this makes the base class an *Abstract Base Class* (one that can't be instantiated)

The “Diamond Problem”

- Suppose we have an inheritance hierarchy like the one on the right
- This can cause a problem in C++ because the final class (Phablet) inherits two instances of MobileDevice (one from Tablet, one from Phone) --- the compiler doesn’t know which instance to access if we want to get at an inherited variable like Memory!



```
// mobile.cc

class MobileDevice
{
  public:
    int Memory;
    MobileDevice() {
      cout << "MobileDevice constructor" << endl;
    }
};

class Phone: public MobileDevice
{
  public:
    Phone() {
      cout << "Phone constructor" << endl;
    }
};

class Tablet: public MobileDevice
{
  public:
    Tablet() {
      cout << "Tablet constructor" << endl;
    }
};

class Phablet: public Phone, public Tablet
{
  public:
    Phablet() {
      cout << "Phablet constructor" << endl;
    }
};

int main()
{
  cout << "About to create a Phablet on the stack" << endl;
  Phablet MyPhablet;
  // MyPhablet.Memory = 32; // this won't work --- ambiguous reference
}
```

Program output:

About to create a Phablet on the stack
MobileDevice constructor
Phone constructor
MobileDevice constructor
Tablet constructor
Phablet constructor

- Notice MobileDevice constructor is called *twice* (once for each parent of Phablet)
- That means if we try to do something like set MyPhablet.Memory = 32 it will fail because it's not sure which Memory we mean, the one from Phone or the one from Tablet!

Solving the Diamond Problem

- We really want just one instance of MobileDevice associated with Phablet, not two
- We can accomplish this using “virtual inheritance”
- Unfortunately virtual inheritance has nothing to do with virtual functions
- Virtual inheritance just means we want a common instance of a base class if we have a diamond problem, not multiple instances of the base class
- To use virtual inheritance, in the “middle-level” classes of the diamond, specify their inheritance like this:

```
class Derived1: public virtual Base { … };
class Derived2: public virtual Base { … };
```

```
// mobile2.cc

class MobileDevice
{
  public:
    int Memory;
    MobileDevice() {
      cout << "MobileDevice constructor" << endl;
    }
};

class Phone: public virtual MobileDevice
{
  public:
    Phone() {
      cout << "Phone constructor" << endl;
    }
};

class Tablet: public virtual MobileDevice
{
  public:
    Tablet() {
      cout << "Tablet constructor" << endl;
    }
};

class Phablet: public Phone, public Tablet
{
  public:
    Phablet() {
      cout << "Phablet constructor" << endl;
    }
};

int main()
{
  cout << "About to create a Phablet on the stack" << endl;
  Phablet MyPhablet;
  MyPhablet.Memory = 32; // now this works!
}
```

Program output:

```
About to create a Phablet on the stack
MobileDevice constructor
Phone constructor
Tablet constructor
Phablet constructor
```

Just one now!

Virtual Copy Constructors

- One might imagine it would be useful to define virtual copy constructors, so that derived classes could implement the copy constructor appropriate to them (with deep copies, etc., specialized to the derived class)
- Unfortunately constructors are considered a special type of function that work only for a specified type; hence, virtual copy constructors are not allowed in C++
- If virtual copy constructors are needed, one can use a work-around of defining a regular virtual function that does much the same work as a copy constructor

Chapter 12: Operator Overloading

In Chapter 4, we considered operators; now we consider how they can be overloaded to work on classes

- How to overload operators

Increment and Decrement Operators

- Suppose we have a Time class that keeps track of the hour (0 to 23) and the minute (0 to 59)
- It would be really convenient to be able to update the time by one minute using a simple ++ operator, like this:

```
Time myTime(0, 0); // initialize to midnight
myTime++; // add one minute!
```

Overloadable Operators

- There are two types of operators: *unary* operators that act on a single object, and *binary* operators that act on two objects
- Overloadable unary operators: ++, --, *, ->, !, &, -, +, ~, conversion operators
- Overloadable binary operators: , (comma), !=, %, %=, &, &&, &=, *, *=, +, +=, -, -=, ->*, /, /=, <, <<, <<=, <=, =, ==, >, >=, >>, >>=, ^, ^=, |, |=, ||, [] (subscript operator)

Syntax to Declare Operator Overloading Functions

- Unary operators
  - Within a class:
    return_type operator operator_type ();
  - Outside a class:
    return_type operator operator_type (param_type);
- Binary operators
  - Within a class:

    return_type operator_type (param1);
  - Outside a class:

    return_type operator_type (param1, param2);

Increment/Decrement Example

- For example, to declare a function that overloads the prefix ++ operator, we could do this:
  return_type operator ++ ();
  This is the operator we use in a statement like ++i; recall this increments and then provides the incremented value
- This is different than i++, which increments i but provides the value *before* incrementing; we call this a postfix ++ operator, and we could declare a function to overload it like this:
  return_type operator ++ (int);
- Decrement operators work similarly

Time Example

- The next example keeps track of the time (hours and minutes) and overloads the ++ and - - operators (both prefix and postfix)
- We also check that the minutes always stay between 0 and 59, and that the hours always stay between 0 and 23
- When we use the prefix operators, we can just increment/decrement and then return the modified object --- to avoid unnecessary object copies, we'll just use a reference return type
- When we use the postfix operators, we need access to the value of the time before we did the increment/decrement (because i++ gives the value of i before the increment); here we have to copy the time before the increment/decrement and then return this (unmodified) copy

```
// timeclass.cc

class Time
{
 private:
   int Hours;   // 0 to 23
   int Minutes; // 0 to 59
 public:
   Time() {
     Hours = Minutes = 0;
   }
   Time(int hrs, int mins)
     :Hours(hrs),Minutes(mins) {
   }
   void printTime() {
     cout << Hours << " hours and " << Minutes << " minutes" << endl;
   }
   // do any rollovers necessary on hours or minutes
   void TimeCheck() {
     if (Minutes > 59) {
       Minutes = 0; ++Hours;
     }
     if (Minutes < 0) {
       Minutes = 59; --Hours;
     }
     if (Hours > 23) Hours = 0;
     if (Hours < 0) Hours = 23;
   }
   // prefix ++ operator, i.e., ++myTime
   Time& operator ++ () {
     ++Minutes; TimeCheck();
     return *this; // don't want to make a new object
   }
   // prefix -- operator, i.e., --myTime
   Time& operator -- () {
     --Minutes; TimeCheck();
     return *this; // don't want to make a new object
   }
   // postfix ++ operator, i.e., myTime++
   // here we need to return the state before increment
   Time operator ++ (int) {
     Time TmpTime(Hours, Minutes); // make a backup
     ++Minutes; TimeCheck();
     return TmpTime;
   }
   // postfix -- operator, i.e., myTime--
   // here we need to return the state before decrement
   Time operator -- (int) {
     Time TmpTime(Hours, Minutes); // make a backup
     --Minutes; TimeCheck();
     return TmpTime;
   }
};
```

```
int main()
{
  Time T1(0, 0);
  --T1;
  T1.printTime();
  T1++;
  T1.printTime();
  Time T2 = T1++; // T2 will equal T1 *before* increment
  T1.printTime();
  T2.printTime();
}
```

Program output:

```
23 hours and 59 minutes
0 hours and 0 minutes
0 hours and 1 minutes
0 hours and 0 minutes
```

Conversion Operators

- We can also use operator overloading to specify how to convert the object to various other types
- For example, it might be nice to be able to print a Time object like this:

```
Time T1(0,0);
cout << “Time is “ << T1 << endl;
```

- For this to work, we need to be able to convert our Time class into a C++-style string or a C-style string (const char*); let’s consider the latter in the next example

```
// timeclass2.cc
#include <iostream>
#include <sstream>    // for stringstream below
#include <string>
using namespace std;

class Time
{
  private:
    int Hours;   // 0 to 23
    int Minutes; // 0 to 59
    string StringForm; // time converted to a string

  public:
    Time() {
      Hours = Minutes = 0;
    }
  …
    operator const char*() {
      ostringstream out;        // stringstream lets us build strings
      out << Hours << ":" << Minutes;
      StringForm = out.str();    // hold this as member data so it and the
                                 // char* to it won't disappear at end
                                 // of this function
      return StringForm.c_str(); // get C-style const char* from string
    }
};

int main()
{
  Time T1(0, 0);
  --T1;
  cout << "Time is " << T1 << endl;
}
```

Program output:
Time is 23:59

Binary Addition/Subtraction

- Binary addition is declared within a class like this: X operator + (argument);
- The current object (of type X) performs an addition with itself and argument (argument could be another object of the same type, or a different type) and it returns a new object of type X
- Subtraction is analogous but with “-” instead of “+” as the operator

```
// timeclass3.cc

class Time
{
  private:
    int Hours;   // 0 to 23
    int Minutes; // 0 to 59

  public:
    Time() {
      Hours = Minutes = 0;
    }
    Time(int hrs, int mins)
      :Hours(hrs),Minutes(mins) {
      TimeCheck();
    }
    void printTime() {
      cout << Hours << " hours and " << Minutes << " minutes" << endl;
    }
    // do any rollovers necessary on hours or minutes
    void TimeCheck() {
      if (Minutes > 59) {
        Hours += Minutes / 60;
        Minutes = Minutes % 60;
      }
      if (Minutes < 0) {
        Hours -= (-Minutes / 60) + 1;
        Minutes = 60 - (-Minutes % 60);
      }
      if (Hours > 23) Hours = Hours % 23;
      if (Hours < 0) Hours = 24 - (-Hours % 24);
    }
// - operator
    Time operator - (const Time &otherTime) {
      int newMinutes = Minutes - otherTime.Minutes;
      int newHours = Hours - otherTime.Hours;
      Time NewTime(newHours, newMinutes);
      return NewTime;
    }
};
```

```
int main()
{
  Time T1(0, 0);
  cout << "T1(0, 0): ";
  T1.printTime();
  Time T2(2, 5);
  cout << "T2(2, 5): ";
  T2.printTime();
  Time T3 = T1 - T2;
  cout << "T3 = T1-T2 = ";
  T3.printTime();
}
```

Program output:

T1(0, 0): 0 hours and 0 minutes

T2(2, 5): 2 hours and 5 minutes

T3 = T1-T2 = 21 hours and 55 minutes

Addition/Subtraction Assignment Operators, += and -=

- These work similarly to + and – operators except that the result is held in one of the operands (instead of making a new copy to hold the result)
- For example, X += Y, where both X and Y can be objects
- Alternatively, Y could actually be of another type (so long as the += or -= operator is defined to handle its addition/subtraction from X)
- To declare this within a class, the syntax is
  void operator += (argument)
  where argument can be of the same type as the class or a different type. There's no return type because we're changing the current object.

```
// timeclass4.cc

class Time
{
  private:
    int Hours;   // 0 to 23
    int Minutes; // 0 to 59

  public:
    Time() {
      Hours = Minutes = 0;
    }
    Time(int hrs, int mins)
      :Hours(hrs),Minutes(mins) {
      TimeCheck();
    }
    void printTime() {
      cout << Hours << " hours and " << Minutes << " minutes" << endl;
    }
    // do any rollovers necessary on hours or minutes
    void TimeCheck() {
      if (Minutes > 59) {
        Hours += Minutes / 60;
        Minutes = Minutes % 60;
      }
      if (Minutes < 0) {
        Hours -= (-Minutes / 60) + 1;
        Minutes = 60 - (-Minutes % 60);
      }
      if (Hours > 23) Hours = Hours % 23;
      if (Hours < 0) Hours = 24 - (-Hours % 24);
    }
}
    // - operator
    void operator -= (const Time &otherTime) {
      Minutes -= otherTime.Minutes;
      Hours -= otherTime.Hours;
      TimeCheck();
    }
};
```

```
int main()
{
  Time T1(0, 0);
  cout << "T1(0, 0): ";
  T1.printTime();
  Time T2(2, 5);
  cout << "T2(2, 5): ";
  T2.printTime();
  T1 -= T2;
  cout << "T1 -= T2 = ";
  T1.printTime();
}
```

Program output:

```
T1(0, 0): 0 hours and 0 minutes
T2(2, 5): 2 hours and 5 minutes
T1 -= T2 = 21 hours and 55 minutes
```

Equality (==) and Inequality (!=) Operators

- It can also be useful to overload the == and != operators so we can see if two objects are equal or not
- If we don't overload the equality operator, then it will return whether or not the objects are exactly equal at a binary level; this is usually too stringent a test and can fail even when the objects basically are equal
- Syntax to overload == is like this:
  bool operator == (const X& otherX) { [comparison code here] }
- Since the != operator is the logical negation of the == operator, we can just write the == operator and then set != to be its logical opposite, like this:

```
bool operator != (const X& otherX) {
        return !(this->operator==(otherX));
}
```

```
// timeclass5.cc

class Time
{
  private:
    int Hours;   // 0 to 23
    int Minutes; // 0 to 59

  public:
    Time() {
      Hours = Minutes = 0;
    }
   …
    bool operator == (const Time &otherTime) {
      return((Minutes == otherTime.Minutes) && (Hours == otherTime.Hours));
    }
    bool operator != (const Time &otherTime) {
      return !(this->operator==(otherTime));
    }
};
```

```
int main()
{
  Time T1(0, 0);
  cout << "T1(0, 0): ";
  T1.printTime();
  Time T2(2, 5);
  cout << "T2(2, 5): ";
  T2.printTime();
  if (T1==T2) cout << "T1 == T2" << endl;
  else cout << "T1 != T2" << endl;
}
```

Program output:

```
T1(0, 0): 0 hours and 0 minutes
T2(2, 5): 2 hours and 5 minutes
T1 != T2
```

Inequality Operators

- We can also overload the inequality operators <, >, <=, and >=
- In the last example, we saw that != was just the negation of ==, so we only had to do work on one of these
- For inequality operators, we can similarly define some of them and then define the others in terms of the ones already defined: for example, if we've defined <=, then > is just the logical negation of <=

```
// timeclass6.cc

class Time
{
…
   bool operator == (const Time &otherTime) {
     return((Minutes == otherTime.Minutes) && (Hours == otherTime.Hours));
   }
   bool operator != (const Time &otherTime) {
     return !(this->operator==(otherTime));
   }
   bool operator < (const Time &otherTime) {
     if (Hours < otherTime.Hours) return(true);
     else if (Hours > otherTime.Hours) return(false);
     // by now we know Hour == otherTime.Hours
     else if (Minutes < otherTime.Minutes) return(true);
     else return(false);
   }
   bool operator <= (const Time &otherTime) {
     if (this->operator==(otherTime)) return(true);
     else return(this->operator<(otherTime));
   }
   bool operator > (const Time &otherTime) {
     return !(this->operator<=(otherTime));
   }
   bool operator >= (const Time &otherTime) {
     if (this->operator== (otherTime)) return(true);
     else return(this->operator>(otherTime));
   }
};
```

```
int main()
{
  Time T1(0, 0);
  cout << "T1(0, 0): ";
  T1.printTime();
  Time T2(2, 5);
  cout << "T2(2, 5): ";
  T2.printTime();
  cout << "T1 == T2 : " << (T1==T2) << endl;
  cout << "T1 < T2  : " << (T1<T2) << endl;
  cout << "T1 > T2  : " << (T1>T2) << endl;
  cout << "T1 <= T2 : " << (T1<=T2) << endl;
  cout << "T1 >= T2 : " << (T1>=T2) << endl;
}
```

Program output:

```
T1(0, 0): 0 hours and 0 minutes
T2(2, 5): 2 hours and 5 minutes
T1 == T2 : 0
T1 < T2  : 1
T1 > T2  : 0
T1 <= T2 : 1
T1 >= T2 : 0
```

Copy Assignment Operator (=)

- If we are creating a new object through a copy of an existing object, like this:
  ClassX X;
  ClassX Y = X;
  that calls the *copy constructor*
- However, if we already have two objects and we then want to copy one into the other, like this:
  ClassX X, Y;
  Y = X;
  that calls the *copy assignment operator*
- This was briefly mentioned in our discussion of copy constructors in Chapter 9
- If our classes contain dynamically allocated memory, we need to ensure we perform deep copies
- Copy assignment operators have the following general syntax:

```
ClassX& operator = (const ClassX& src) {
        if (this != &src) { // don't copy into self
                // do the copying
        }
        return *this;
}
```

```
// grades.cc

class ClassGrades
{
  private:
    string Name;
    int* Grades;

  public:
    ClassGrades(string inName) {
      cout << "In constructor" << endl;
      Name = inName;
      Grades = new int[3]; // assume 3 assignments
      for (int i=0; i<3; i++) { Grades[i] = 0; }
    }
    ~ClassGrades() {
      cout << "In destructor" << endl;
      delete [] Grades;
    }
ClassGrades& operator= (const ClassGrades& src) {
      cout << "In Copy Assignment Operator" << endl;
      if (this != &src) {
        Name = src.Name;
        if (Grades != NULL) delete[] Grades;
        Grades = new int[3];
        for (int i=0; i<3; i++) { Grades[i] = src.Grades[i]; }
      }
    }
    void SetGrade(int id, int grade) {
      Grades[id] = grade;
    }
    void PrintGrades() {
      cout << "Student " << Name << " has grades: ";
      for (int i=0; i<3; i++) {
        cout << Grades[i] << " ";
      }
      cout << endl;
    }
};
```

```
int main()
{
  ClassGrades Student1("John Smith");
  Student1.SetGrade(0, 99);
  cout << "Student1: "; Student1.PrintGrades();
  ClassGrades Student2("Jane Doe");
  Student2.SetGrade(0, 100);
  cout << "Student2: "; Student2.PrintGrades();
  Student2 = Student1;
  cout << "Student2: "; Student2.PrintGrades();
}
```

Program output:

```
In constructor
Student1: Student John Smith has grades: 99 0 0
In constructor
Student2: Student Jane Doe has grades: 100 0 0
In Copy Assignment Operator
Student2: Student John Smith has grades: 99 0 0
In destructor
In destructor
```

Subscript Operator []

- We can overload the [] operator; this is useful to access elements of an array inside a class
- Typical syntax
  /* const*/ return_type& operator [] (int Index) /*const*/
- In the line above, the first (optional) const would mean we can't modify the value that's being returned; the second (optional) const means that the overloaded operator can't modify the class attributes at all. Using const at either or both places helps preserve data encapsulation (although might be inconvenient or inappropriate if we did need to allow direct access to the data)

```
// grades2.cc

class ClassGrades
{
  private:
    string Name;
    int* Grades;

  public:
   ...
    /* replaced by [] operator below
    void SetGrade(int id, int grade) {
      Grades[id] = grade;
    }
    */
    int& operator [] (int Index) {
      if (Index >=0 && Index < 3) {
        return Grades[Index];
      }
    }
  ...
};
```

```
int main()
{
  ClassGrades Student1("John Smith");
  Student1[0] = 99;
  cout << "Student1: "; Student1.PrintGrades();
  ClassGrades Student2("Jane Doe");
  Student2[0] = 100;
  cout << "Student2: "; Student2.PrintGrades();
  Student2 = Student1;
  cout << "Student2: "; Student2.PrintGrades();
}
```

Program output:

```
In constructor
Student1: Student John Smith has grades: 99 0 0
In constructor
Student2: Student Jane Doe has grades: 100 0 0
In Copy Assignment Operator
Student2: Student John Smith has grades: 99 0 0
In destructor
In destructor
```

Overloading (): The Function Operator

- We can overload the () operator to make an object behave like a function (a “functor”)
- This is most often used to provide callback functions (a function that will get called by another function or object) without having to use function pointers
- More on functors later; the next page gives a simple example

```
// printclass.cc
#include <iostream>
#include <string>
using namespace std;

class PrintClass
{
  public:
    void operator () (string Input) const
    {
      cout << Input << endl;
    }
};

int main()
{
  PrintClass myPrintObject;
  myPrintObject("Hello, World!");
}
```

C++11: Move Constructors

- We've already discussed the need to have deep copies when we copy an object that contains dynamically allocated memory
- However, this copying can degrade performance if it's called for objects that only exist temporarily
- For example, an overloaded + or – operator typically constructs a new object of the current class and returns it --- but the object goes out of scope when the +/- function is done, so we wind up invoking a copy constructor (and deep-copying dynamically allocated memory)
- It would be better if we could "move" the temporary object about to go out of scope into the object that is now being created from it, to avoid an otherwise unnecessary deep copy

Move Constructor Example

Suppose we have a record of class grades for student John Smith, and we want to see how far above or below the Class Average John scored on each assignment; we can make another record for the Class Averages and then just subtract the two objects using an overloaded subtraction operator.

```
int main()
{
  ClassGrades Student1("John Smith");
  Student1[0] = 99; Student1[1] = 85; Student1[2] = 86;
  Student1.PrintGrades();
  ClassGrades Avg("Class Average");
  Avg[0] = 70; Avg[1] = 72; Avg[2] = 68;
  Avg.PrintGrades();
  // Compute how far above/below Class Avg were each of Student1's scores
  ClassGrades StudentvsAvg("Label-will-be-overwritten");
  StudentvsAvg = Student1 - Avg;
  StudentvsAvg.PrintGrades();
}
```

Move Constructor Example

- We can define the subtraction operator this way:

```
ClassGrades operator- (const ClassGrades& ToSubtract) {
    cout << "Subtracting record " << ToSubtract.Name;
    cout << " from record " << Name << endl;
    string NewName;
    NewName = Name + " - " + ToSubtract.Name;
    ClassGrades NewClassGrades(NewName);
    for (int i=0; i<3; i++) {
      NewClassGrades[i] = Grades[i] - ToSubtract.Grades[i];
    }
    return NewClassGrades;
  }
```

- When we return NewClassGrades it goes out of scope, so in principle the compiler should call a copy constructor to copy it into a new object in the calling code (here, Student1 – Avg)
- Some compilers seem to be smart enough to optimize away this extra copy automatically; if this doesn't happen, we can use a Move Constructor

Move Constructor Example

```
ClassGrades(ClassGrades&& src) { // move constructor (C++11 only)
    cout << "In Move Constructor moving from " << src.Name << endl;
    Name = src.Name;
    Grades = src.Grades; // take ownership of arrays
    src.Grades = NULL;
  }
```

- The && in the argument distinguishes this constructor as a move constructor
- We “move” the pointer to the Grades array from being owned by src to being owned by the current object (“this”)
- We then invalidate the src.Grades pointer in the source object, so that if it goes out of scope, the memory won’t go away with it (this means we need to change the destructor to free Grades[] only if the pointer isn’t NULL)

```
~ClassGrades() {
    cout << "In destructor for record " << Name << endl;
    if (Grades != NULL) delete [] Grades;
  }
```

Move Assignment Operators

- In our example code we use a copy assignment operator to copy the result of subtraction into an existing object:

```
ClassGrades StudentvsAvg("Label-will-be-overwritten");
StudentvsAvg = Student1 - Avg;
```

  - This can also lead to an unwanted deep copy, because Student1-Avg is going out of scope as soon as it is copied into StudentsvsAvg
  - By the way, Student1-Avg it is called an “r-value” because it has a value, but that value can only sit on the right-hand side of an assignment operation (i.e., it would never make sense to say Student1-Avg = X); move operators are useful in avoiding copies of r-value expressions
  - A move assignment operator lets us move (dynamically allocated member data from) the temporary rvalue (Student1-Avg) into the existing object StudentsvsAvg

Move Assignment Operator Example

```
// move assignment operator (C++11 only)
  ClassGrades& operator= (ClassGrades&& src) {
    cout << "In Move Assignment Operator" << endl;
    cout << "Moving record " << src.Name << " into record " << Name << endl;
    Name = src.Name;
    if (this != &src && src.Grades != NULL) {
      if (Grades != NULL) delete[] Grades;
      Grades = src.Grades; // take ownership of arrays
      src.Grades = NULL;
    }
    return *this;
  }
```

Complete Example (w/out C++ Move Operations)

```
// grades3.cc

class ClassGrades
{
  private:
    string Name;
    int* Grades;

  public:
    ClassGrades(string inName) {
      cout << "In constructor for record " << inName << endl;
      Name = inName;
      Grades = new int[3]; // assume 3 assignments
      for (int i=0; i<3; i++) { Grades[i] = 0; }
    }
    ~ClassGrades() {
      cout << "In destructor for record " << Name << endl;
      if (Grades != NULL) delete [] Grades;
    }
    ClassGrades(const ClassGrades& src) {
      cout << "In Copy Constructor copying from " << src.Name << endl;
      Name = src.Name;
      Grades = new int[3];
      for (int i=0; i<3; i++) { Grades[i] = src.Grades[i]; }
    }
    ClassGrades& operator= (const ClassGrades& src) {
      cout << "In Copy Assignment Operator" << endl;
      cout << "Copying record " << src.Name << " into record " << Name << endl;
      Name = src.Name;
      if (this != &src && src.Grades != NULL) {
        if (Grades != NULL) delete[] Grades;
        Grades = new int[3];
        for (int i=0; i<3; i++) { Grades[i] = src.Grades[i]; }
      }
      return *this;
    }
    int& operator [] (int Index) {
      if (Index >=0 && Index < 3) {
        return Grades[Index];
      }
    }
    ClassGrades operator- (const ClassGrades& ToSubtract) {
        cout << "Subtracting record " << ToSubtract.Name;
        cout << " from record " << Name << endl;
        string NewName;
        NewName = Name + " - " + ToSubtract.Name;
        ClassGrades NewClassGrades(NewName);
        for (int i=0; i<3; i++) {
          NewClassGrades[i] = Grades[i] - ToSubtract.Grades[i];
        }
        return NewClassGrades;
      }
      void PrintGrades() {
        cout << "Student " << Name << " has grades: ";
        for (int i=0; i<3; i++) {
          cout << Grades[i] << " ";
        }
        cout << endl;
      }
};

int main()
{
  ClassGrades Student1("John Smith");
  Student1[0] = 99; Student1[1] = 85; Student1[2] = 86;
  Student1.PrintGrades();
  ClassGrades Avg("Class Average");
  Avg[0] = 70; Avg[1] = 72; Avg[2] = 68;
  Avg.PrintGrades();
  // Compute how far above/below Class Avg were each of Student1's
scores
  ClassGrades StudentvsAvg("Label-will-be-overwritten");
  StudentvsAvg = Student1 - Avg;
  StudentvsAvg.PrintGrades();
}
```

Most Relevant Parts, and Output

```
ClassGrades operator- (const ClassGrades& ToSubtract) {
    cout << "Subtracting record " << ToSubtract.Name;
    cout << " from record " << Name << endl;
    string NewName;
    NewName = Name + " - " + ToSubtract.Name;
    ClassGrades NewClassGrades(NewName);
    for (int i=0; i<3; i++) {
      NewClassGrades[i] = Grades[i] - ToSubtract.Grades[i];
    }
    return NewClassGrades;
  }
```

```
int main()
{
  ClassGrades Student1("John Smith");
  Student1[0] = 99; Student1[1] = 85; Student1[2] = 86;
  Student1.PrintGrades();
  ClassGrades Avg("Class Average");
  Avg[0] = 70; Avg[1] = 72; Avg[2] = 68;
  Avg.PrintGrades();
  // Compute how far above/below Class Avg were each of Student1's
scores
  ClassGrades StudentvsAvg("Label-will-be-overwritten");
  StudentvsAvg = Student1 - Avg;
  StudentvsAvg.PrintGrades();
}
```

- Compiler seems to have optimized out the copy constructor when we return from the (-) operator; excellent!
- But we are still having to do a deep copy in the copy assignment operator; better to "move" the tempoarary rvalue (Student1-Avg) into StudentvsAvg !

Program output:

```
In constructor for record John Smith
Student John Smith has grades: 99 85 86
In constructor for record Class Average
Student Class Average has grades: 70 72 68
In constructor for record Label-will-be-overwritten
Subtracting record Class Average from record John Smith
In constructor for record John Smith - Class Average
In Copy Assignment Operator
Copying record John Smith - Class Average into record Label-will-be-overwritten
In destructor for record John Smith - Class Average
Student John Smith - Class Average has grades: 29 13 18
In destructor for record John Smith - Class Average
In destructor for record Class Average
In destructor for record John Smith
```

- If we add the move constructor and move assignment operators given above (example listing grades3a.cc), we now get this output

```
In constructor for record John Smith
Student John Smith has grades: 99 85 86
In constructor for record Class Average
Student Class Average has grades: 70 72 68
In constructor for record Label-will-be-overwritten
Subtracting record Class Average from record John Smith
In constructor for record John Smith - Class Average
In Move Assignment Operator
Moving record John Smith - Class Average into record Label-will-be-overwritten
In destructor for record John Smith - Class Average
Student John Smith - Class Average has grades: 29 13 18
In destructor for record John Smith - Class Average
In destructor for record Class Average
In destructor for record John Smith
```

- The move assignment operator has replaced the copy assignment operator (and saved us one deep copy)
- To get the C++ features of grades3a.cc working, we compiled it like this (using g++):

```
g++ -std=c++11 -o grades3a grades3a.cc
```

Chapter 13: Casts

A cast is an operation that converts one type to another, sometimes in a way that is not strictly safe; use with caution

- C-style casts
- static_cast
- dynamic_cast
- reinterpret_cast
- const_cast

The Reason for Casts

- In well written C++, casts should hardly ever be needed. However, in C, they were needed somewhat frequently, and hence they may also be needed in a C++ program if it calls legacy C code
- For example, C originally did not support Boolean data types; integers were usually used as stand-ins (0 for false and 1 for true)
- If we want to call an old C routine from C++, we might need to convert our data from Boolean types to integers
- Rather than making the programmer do the conversion manually, we can do it automatically using a cast
- A C-style cast can still be used in C++; the syntax to convert variable y from some type old_type to a variable x of type new_type is

```
new_type x = (new_type) y;
```

C-Style Casting Example

```
// c-cast.cc
#include <iostream>
using namespace std;

int main()
{
  int i;
  bool b_t = true, b_f = false;

  i = (int) b_t;
  cout << "true casts to int as " << i << endl;
  i = (int) b_f;
  cout << "false casts to int as " << i << endl;
}
```

Program output:
true casts to int as 1
false casts to int as 0

Implicit Casts

- It is also possible to forget the cast syntax, and just let the compiler do the conversion for you. For example,

```
old_type j = some_value;
new_type i = j;
```

- This avoids the need to add the cast specifier [e.g., new_type i = (old_type) j]
- However, since a cast is being performed, it is considered better form to specify the cast directly as a clue to other programmers that a conversion is happening (especially because casts should always be used with caution)
- Implicit casts or C-style casts can effectively convert things that should be convertible, like between integers and Booleans, double precision and integers (with roundoff of course), etc.
- They may not work as desired (or at all) for other examples, especially for user-defined data types (classes)

Implicit Cast Example (not recommended)

```
// implicit-cast.cc
#include <iostream>
using namespace std;

// let the compiler cast it for us automatically
int main()
{
  int i;
  bool b_t = true, b_f = false;

  i = b_t;
  cout << "true casts to int as " << i << endl;
  i = b_f;
  cout << "false casts to int as " << i << endl;
}
```

Program output:
true casts to int as 1
false casts to int as 0

Problems with C-Style Casts

- As we've just seen, it's usually considered better to use casting syntax rather than let the compiler handle it implicitly, as a way of warning other programmers that a conversion is happening
- On the other hand, C-style casts can be very dangerous because they can be used to write code that doesn't really make sense
- For example, the following code won't work in C++:

```
char* name = "David";
int* buf = name; // can't convert char* to int*
```

- However, we could force it to execute with a C-style cast:

```
int* buf = (int *)name;
```

- But we couldn't (or at least shouldn't!) use buf as an integer pointer to do any integer operations, because the contents aren't really meaningful as integers
- The reason C allows such nonsense casts is that in old C, libraries that dealt with arrays of arbitrary types had to have the pointers cast to some common pointer type (often char* or, later, void*)

C++ Ways to do Casts

- Given the dangerous nature of C-style casts, C++ introduces several new casting operations; which one to use depends on the situation
- Unfortunately, the C++ casts aren't particularly safe operations, either, and the syntax makes them a bit more cumbersome to use than the old C-style casts
- The C++ casts are of four types: static_cast, dynamic_cast, reinterpret_cast, and const_cast
- The syntax for all these has a common form:

  new_type x = cast_type <new_type>(y)
- This casts variable y of some old datatype to variable x of type new_type

static_cast

- static_cast will do explicit type conversion between basic data types where it makes sense (e.g., converting a double like 4.184 to an integer 4) and it can convert pointers between related data types
- The validity of the cast is checked at compile type (that's the "static" in static_cast)
- When converting pointers, the cast is valid if the compiler detects the data types are "related" (e.g., they are from the same class inheritance hierarchy)

Base and Derived Class Pointers

- Suppose we have a Base Class (say, Polygon) and a Derived Class (say, Square)
- If we make a new Square, we can have a pointer to it of type Square*
- We could just as easily treat this as a pointer of type Polygon*, since of course a Square is a Polygon
- Hence, we could create a Polygon* pointer to a new Square like this, with no problems:

```
Polygon* pPolygon = new Square();
```

- This does an implicit conversion of Square* to Polygon*; converting a pointer to a type upwards (towards parents) in an inheritance hierarchy is called “upcasting” and is perfectly valid

Base and Derived Class Pointers

- However, going the other way (converting a pointer of a parent type to a derived type) is called “downcasting” (because it goes down in the inheritance hierarchy) is not ok:

  ```
  Square* pSquare = pPolygon; // error
  ```

  after all, we don’t know the Polygon is really a square! The line above won’t compile.
- But what if pPolygon is really a pointer to a square, e.g., if we got it from an upcast:

  ```
  Polygon* pPolygon = new Square();
  ```
- Then we should be able to convert the Polygon* pointer back to a Square* because the object pointed to really is a square! Can do this with static_cast:

  ```
  Square* pSquare = static_cast<Square*>(pPolygon);
  ```

Dangers of static_cast

- So, static_cast gives us a way to explicitly convert basic data types or to convert pointers for related data types, like a Polygon* to a Square* when the Polygon* is really pointing to a Square
- The problem is that static_cast will also allow us to convert any Polygon* to Square*, even when Polygon* might have been upcast from some other shape (like Triangle*) or when Polygon* is simply pointing to a generic Polygon
- Of course this is bad because if we had a Polygon* pointing to a Triangle, and then cast it to a Square*, we would get undefined, nonsensical results if we then tried to access the Triangle like it was a Square

dynamic_cast

- The problems in using static_cast raised in the previous slide can be remedied using dynamic_cast, which means that the cast is attempted at runtime, when the system can know whether or not the cast succeeded
- Continuing the previous example, if we tried to dynamically cast a Polygon* that points to a Triangle down to a Square*, the cast would fail (as it should)
- If a cast fails, the pointer resulting from the cast is set to NULL; we can check for this before executing code that depends on the cast working

dynamic_cast Example

```
Polygon* pPolygon = new Triangle();
// try a dynamic downcast
Square* pSquare =
        dynamic_cast<Square*>(pPolygon);

if (pSquare) // See if cast succeeded (fails here)
        pSquare->SomeSquareMethod();
```

dynamic_cast Example

- The previous example is a bit silly because it's obvious to the programmer that pPolygon* indeed points to a Triangle, not a Square, so the dynamic cast is bound to fail
- However, you can easily imagine having a Polygon* passed to some subroutine that handles Polygons in a mostly generic way; maybe upcasting was done to take derived type pointers and convert them to Polygon* types so they could all be handled generically
- But if such upcasting was done outside the subroutine, the subroutine doesn't know what the original derived-type pointer types were…fortunately dynamic_cast does
- On the other hand, one might argue the subroutine is not well designed because it should indeed handle all the Polygon* types generically, and not need to know what the derived type is; virtual functions can still provide different functionality for different derived types

reinterpret_cast

- reinterpret_cast is basically the C++ version of the C-style cast: it says to “reinterpret” the old pointer type as the new pointer type (i.e., pretend they are the same), even when these types really have nothing to do with each other
- It has all the same dangers as C-style casts, and should therefore be avoided when possible
- It is useful for converting classes to some very basic data type for use in low-level libraries, such as writing out a class as a series of bytes to disk:

```
old_type* x;
unsigned char* buffer =
        reinterpret_cast<unsigned char*>(x);
```

const_cast

- const_cast is a way to ignore the const properties of an object
- This should not be used as a hack to get around an inconsistency in how const was specified in your code
- However, sometimes you interface with code from another source and you can’t make everything consistent; const_cast provides a workaround for mismatches in const specifications

const_cast Example

```
// const-cast.cc
// …
class MyClass {
  public:
    void Print() { // should be declared const but isn't
      cout << "Object prints!" << endl;
    }
};

// const below promises PrintData() won’t change X, but
// we didn’t declare MyClass::Print() as const, so X.Print() looks
// like trouble to the compiler
void PrintData(const MyClass& X) {
  // X.Print(); // compiler gives error
  MyClass& Y = const_cast <MyClass&>(X); // workaround
  Y.Print();
}

int main()
{
  MyClass X;
  PrintData(X);
}
```

Casting Summary

- Casting is ok in some situations, like when we need to truncate a double to an integer
- Casting is necessary in some cases when we want to interact with a legacy library (that can only deal with bytes of type unsigned char*, for example)
- Casting a pointer from a derived type to a parent type is called upcasting and is ok
- Casting a pointer from a parent type to a derived type is called downcasting and is only ok if the parent type pointer is *really* already of the derived type (e.g., we got the parent pointer from an upcast); dynamic_cast will check this for us and is safe to use as long as we catch failures by checking if the resulting pointer is NULL or not
- Usually it is better to use virtual functions than dynamic casts
- Using a const_cast is a bit of a hack but can be acceptable if you don't have control over all the code you're using and you need a workaround to solve a const mismatch

Chapter 14: Macros, Templates, and Smart Pointers

- The preprocessor and #define macros
- #ifdef, #ifndef, #endif, and defining symbols (like –D DEBUG) at compile time
- Using assert() in debugging
- Header guards
- Template functions
- Template classes
- Static member data in template classes
- shared_ptr
- unique_ptr

The C++ Preprocessor

- The preprocessor performs some basic text substitutions on a source file before handing it to the compiler
- It inserts the contents of header files included with #include
- It makes substitutions specified by #define
- #include and #define are called “preprocessor directives”

#define Macros

- #define allows one to specify shortcuts to more complicated expressions or to name constants. Examples:
  #define PI 3.1415926
  #define SQUARE(y) ((y)*(y))
- These types of uses made sense in the days of C, but C++ offers better ways to do these things now

#define constants

- In the days of C, it made much more sense to #define a constant like #define PI 3.14159 and then use the symbol PI, instead of literally hard-coding the number 3.14159 over and over in the code
- After all, it might be important to use the same number of digits consistently to avoid numerical issues, and we may decide later we need to add more digits; we could accomplish that by changing just one definition in the code
- However, with the C++ const keyword, it's better now to just define this as a const (which is type-safe):

  const double PI 3.14159;

#define macros

- For certain simple operations, it can be convenient to create a #define macro and use that instead of using a function call
- For example,
  #define SQUARE(y) ((y)*(y))
  can be used to replace something like SQUARE(42) with ((42)*(42))
- The advantage of the macro is that it will work equally well with float, double, and int types, and it doesn't require the overhead of an explicit function call
- The disadvantage is that it doesn't protect us if the user tries to pass it a string, etc.
- If you do use #define to make a macro, *you do need all those extra parentheses*: otherwise, unexpected results could occur if we pass an expression instead of a simple argument

#ifdef, #ifndef, and #endif

- The preprocessor can “turn on” or “turn off” (really, include or not include) a section of code based on directives
- To include a section of code if a symbol X has been defined, do this:

```
#ifdef X
[code to be included]
#endif
```

- To include a section of code if a symbol X has not been defined, do this:

```
#ifndef X
[code to be included]
#endif
```

Defining Symbols

- So, #ifdef X will include code up to the subsequent #endif directive, if symbol X is defined. How to we define this symbol?
- (1) We can simply define it using a preprocessor directive like this:
  #define X some-value
  (like in our previous SQUARE(x) macro example)
- (2) We don't even need to give the symbol a value; it's enough to just say it's defined, like this:
  #define X
- (3) We can also define the symbol at compile time; C++ compilers allow symbols to be defined using –D SYMBOL[=VALUE]

Example: Preprocessor Directives for Optional Debugging Code

- Frequently, we want to run a lot of extra checks when we are debugging a code, but not necessarily when we are running it normally (depending on what tests are done, they might slow down the code)
- We can do this with preprocessor directives! Just wrap the debug code in #ifdef DEBUG … #endif
- To turn on debugging, add #define DEBUG to the top of the file (before any instances of #ifdef DEBUG), or even better, just recompile with a compiler flag –D DEBUG (or –DDEBUG … the space is optional)
- To turn off debugging, remove any #define DEBUG directives and recompile without the –D DEBUG flag

-D DEBUG Example

```
// define_debug.cc
// for g++ compiled with:
// g++ -DDEBUG -o define_debug define_debug.cc
// -DDEBUG is equivalent to the source #define DEBUG
// If we recompile without -DDEBUG then the debug test
// below will not run
#include <iostream>
using namespace std;

int main() {
  #ifdef DEBUG
  cout << "I'm doing a debug test now." << endl;
  #endif
  cout << "Hello, world" << endl;
}
```

Debugging with assert()

- Alternatively, we can also add some debugging checks with the assert() function (sometimes implemented as preprocessor macros)
- assert(x) checks whether the expression x evaluates as true or not, e.g., assert(5>3); if the assertion fails, then the program prints an error message
- To use, #include <assert.h>
- Not recommended over #define DEBUG sections, which are more flexible (can do more than check the truth of some expression) and can be turned on or off (although some build systems might be smart enough turn off assert() for release builds)
- Another drawback: assert() can cause a core dump, and the programmer and/or user might not want the core dump files cluttering the directory

assert() example

```
// assert.cc
#include <iostream>
#include <assert.h>
using namespace std;

int main() {
  int x=5;

  assert (x>7);
  cout << "End of program" << endl;
}
```

Program output:

```
assert: assert.cc:9: int main(): Assertion `x>7' failed.
Abort (core dumped)
```

Preprocessor Header Guards

- Another legitimate use of the preprocessor is to create “header guards”
- Some C++ projects are complicated enough, with some header files including other header files, that it can be hard to track whether or not you already have included all the header files you need
- To avoid unnecessary multiple inclusion of header files (or even an infinite regression of header files including each other!), we can use a “header guard”
- The header guard will make use of preprocessor directives #ifndef and #endif

Header Guard Syntax

File myheader.h:

```
#ifndef MYHEADER_H
#define MYHEADER_H
[regular contents of header go here]
#endif // end of myheader.h
```

Now when myheader.h gets included, the contents are only added once; this sets MY_HEADER_H, and subsequent encounters of #include “myheader.h” will not add anything because MYHEADER_H is already defined

Introduction to Templates

- One advantage of a macro like #define SQUARE(y) ((y)*(y)) is that it can work with multiple data types, like ints, floats, and doubles
- However, a disadvantage is that it is not type-safe
- Templates provide a way to perform generic operations in a type-safe way; they also provide a way to create generic classes that utilize different types

Template Functions

- To create a function that deals generically with multiple datatypes, make a “template function” like this:

  template <typename T1[, typename T2, …]>
  return-type function-name ([function args])
  { … function definition … }

- If two of the types have to be identical, you can specify it like this:
  template <typename T1, typename T2 = T1>

```
// template_function.cc
#include <iostream>
using namespace std;

template <typename T>
T Square(const T& y)
{
  T ySquared = y * y; // * operator must exist for type T
  return (ySquared);
}

int main()
{
  int p = 2;
  int pSquared = Square(p);
  cout << p << " squared = " << pSquared << endl;

  double q = 0.5;
  double qSquared = Square(q);
  cout << q << " squared = " << qSquared << endl;
}
```

Program output:
2 squared = 4
0.5 squared = 0.25

Template Classes

- Just like we can use templates to create generic functions, we can also use templates to create generic classes that employ different types for member data and/or in their member functions
- The syntax for specifying a template class is similar to that for a template function:
  template <typename T1[, typename T2, …]>
  class MyClass
  { … class definition here, using T1, etc., to substitute for specific type names where desired…};
- To create a template class, you have to tell it what type(s) to use in creating it; specify these in <>:
  MyClass <int> IntMyClass;

```
// template_class.cc
#include <iostream>
using namespace std;

template <typename T>
class MyClass
{
  private:
    T Val;

  public:
    void SetVal(const T& x) { Val = x; }
    const T& GetVal() const { return(Val); }
    // function above is const function: does not change obj
    // returns const ref, meaning we can't change the value
    // even though GetVal() is giving us a ref
};

int main()
{
  MyClass <int> I; // specify what type to use with <>
  I.SetVal(42);
  std::cout << "I holds value " << I.GetVal() << endl;
  MyClass <double> D;
  D.SetVal(3.1415926);
  std::cout << "D holds value " << D.GetVal() << endl;
}
```

Program output:

I holds value 42

D holds value 3.14159

Templates with Default Types

- We can specify default datatypes for templates like this:

```
template <typename T1=int,
          typename T2=int>
class MyClass { … };
```

- A class could be instantiated like this:

```
MyClass <> X(1,2); // use default type int
MyClass <double, double> Y(1.2, 3.4);
```

Static Member Data in Template Classes

- Static member data in template classes have to be defined outside the template class, even if the data type of the static data is fixed and doesn't depend on the template
- We accomplish this as follows:
  template<typename T[,…]>
  StaticMemberType
  ClassName<T[,…]>::StaticMemberName;

```
// template_class_static.cc
#include <iostream>
using namespace std;

// oversimplified template class example where the class doesn't
// even really depend on typename, but the point is to show
// how template classes work with static member data
template <typename T>
class MyClass
{
  public:
    static int S;
};

// static data has to be declared outside the class
// here's how we do it, even though in this case the
// static data is always an int and doesn't depend on typename
template <typename T> int MyClass<T>::S;

int main()
{
  MyClass<int> IntClass;
  MyClass<int> IntClass2;
  IntClass.S = 1;

  MyClass<double> DoubleClass;
  DoubleClass.S = 99;

  cout << "IntClass static value = " << IntClass.S << endl;
  cout << "IntClass2 static value = " << IntClass2.S << endl;
  cout << "DoubleClass static value = " << DoubleClass.S << endl;
}
```

Program output:

```
IntClass static value = 1
IntClass2 static value = 1
DoubleClass static value = 3
```

Smart Pointers

A “smart pointer” is a pointer that automates destruction of the object it points to; consider using smart pointers if you want to dynamically allocate user-defined objects (classes) with “new”

When to Use Smart Pointers

- Smart pointers are a very useful replacement for regular pointers when you create a new object using “new”: they work like regular pointers, but when no pointer points to the object anymore, the object is *automatically* cleaned up with “delete”
- This is extremely helpful because it removes the need for the programmer to track when to delete the object (or who “owns” the object and is responsible for cleaning it up)
- They are not necessarily helpful if the programmer needs an object just within the scope of one function: there may be no need to create the object using new, or else it may be very simple to just call delete at the end of the function

std::shared_ptr

- A “shared pointer” is a smart pointer that tracks all references to the object; when the reference count gets to 0, the object is no longer needed and is automatically deleted. This type of pointer is called a “reference counted” pointer
- Originally standardized in the Boost library (www.boost.org), a popular place for helpful C++ libraries; eventually moved into the standard library as of C++11 (may need to compile with C++11 flags like this: g++ -std=c++11)
- Need to #include <memory> to use it

std::shared_ptr

- Create a shared_ptr smart pointer pointing to an object of type X using this syntax:

  std::shared_ptr<X> MyPtr(new X(ctor-args));

  where ctor-args are the arguments to the constructor
- We can then use MyPtr like a pointer (e.g., * and -> operations work because they've been overloaded)
- However, if we pass the pointer to a function, the function needs to be told to expect type std::shared_ptr<X>, not type X*

Compiling with C++11 features

- Some C++11 features, like smart pointers, are not fully implemented yet in all compilers
- To get the smart pointer examples in this chapter to work with gcc 4.8.2, I had to specify g++ -std=c++11

```
// shared_ptr.cc
#include <iostream>
#include <memory>
using namespace std;

class MyClass {
  public:
    MyClass() { cout << "Constructing MyClass" << endl; }
    ~MyClass() { cout << "Destructing MyClass" << endl; }
};

void misc_function(shared_ptr<MyClass> p);

int main()
{
  shared_ptr<MyClass> myp(new MyClass);
  // use_count() prints number of references the object pointed to;
  // mainly used for debugging / pedagogical purposes
  cout << "Just created shared pointer myp" << endl;
  cout << myp.use_count() << endl;
  misc_function(myp);
  cout << "Back in main" << endl;
  cout << myp.use_count() << endl;
}

void misc_function(shared_ptr<MyClass> p) {
  // Passed p by value, so now one more reference to
  // original obj while we're in this function
  cout << "In misc_function." << endl;
  cout << p.use_count() << endl;
}
```

Program output:

```
Constructing MyClass
Just created shared pointer myp
1
In misc_function.
2
Back in main
1
Destructing MyClass
```

Another shared_ptr Example

- This example revisits our “ClassGrades” class and this time dynamically allocates the objects and puts smart pointers (shared_ptr) pointing to them into a standard vector
- A vector of shared pointers to objects of type X is declared this way:
  vector<shared_ptr<X>> MyVec;

```
// shared_ptr_grades.cc
// for g++ compiled with:
// g++ -std=c++11 -o shared_ptr_grades shared_ptr_grades.cc
#include <iostream>
#include <string>
#include <vector>
#include <memory>  // for shared_ptr
using namespace std;

class ClassGrades
{
 private:
  string Name;
  int* Grades;

 public:
  ClassGrades(string inName) {
   cout << "In constructor for record " << inName << endl;
   Name = inName;
   Grades = new int[3]; // assume 3 assignments
   for (int i=0; i<3; i++) { Grades[i] = 0; }
  }
  ~ClassGrades() {
   cout << "In destructor for record " << Name << endl;
   if (Grades != NULL) delete [] Grades;
  }
  int& operator [] (int Index) {
   if (Index >=0 && Index < 3) {
    return Grades[Index];
   }
  }
  void PrintGrades() {
   cout << "Student " << Name << " has grades: ";
   for (int i=0; i<3; i++) {
    cout << Grades[i] << " ";
   }
   cout << endl;
  }
};
```

```
int main()
{
 // This time we will dynamically allocate students using new, and we
 // will handle the pointers using smart pointers so we don't have to
 // explicitly remember to delete the ClassGrades objects
 vector<shared_ptr<ClassGrades>> StudentList;

 // Just make 2 of them for simplicity
 int NumStudents = 2;
 for (int i=0; i<NumStudents; i++) {
  // make a shared pointer to a new student
  if (i==0) {
   shared_ptr<ClassGrades> NewStudent(new ClassGrades("John Smith"));
   (*NewStudent)[0] = 70; (*NewStudent)[1] = 72; (*NewStudent)[2] = 86;
   StudentList.push_back(NewStudent);
  }
  else if (i==1) {
   shared_ptr<ClassGrades> NewStudent(new ClassGrades("Sally Brown"));
   (*NewStudent)[0] = 82; (*NewStudent)[1] = 90; (*NewStudent)[2] = 80;
   StudentList.push_back(NewStudent);
  }
 }

 // iterate through the vector
 for (int i=0; i<StudentList.size(); i++) {
  cout << "Grades for student #" << i << ":" << endl;
  StudentList[i]->PrintGrades();
 }

 // we created the ClassGrades objects with new but they will delete
 // themselves when the smart pointers go out of scope
}
```

Program output:

```
In constructor for record John Smith
In constructor for record Sally Brown
Grades for student #0:
Student John Smith has grades: 70 72 86
Grades for student #1:
Student Sally Brown has grades: 82 90 80
In destructor for record John Smith
In destructor for record Sally Brown
```

std::unique_ptr

- Also introduced into C++11, and also defined in the header <memory>
- unique_ptr is a smart pointer that allows only one copy of the pointer to exist; one cannot copy a unique_ptr. However, one can move a unique_ptr to another unique_ptr (using std::move()) to “transfer ownership” of the object.
- Can still pass to a function if we use type const unique_ptr & (a reference, so we don’t have to make a copy)

unique_ptr Example

```
#include <memory>
using namespace std;

int main()
{
  unique_ptr<int> p(new int(5)); // make a smart ptr to an int holding val 5
//unique_ptr<int> q = p; // compile error can't make copy of unique_ptr
  unique_ptr<int> q = std::move(p); // Transfer ownership of p to q

  q.reset(); // deletes memory pointed to by q
  p.reset(); // does nothing, move operation invalidated p as pointer

}
```

Chapter 15: Exception Handling

- Introduction to exception handling
- try and catch
- The std::exception class
- Custom exceptions derived from std::exception

Introduction to Exceptions

- An “exception” is just a (hopefully) unusual situation that might cause problems for a program: an attempt to allocate more memory than we have, trying to read too many lines from a file, trying to divide by zero, etc.
- C++ provides a mechanism to try to catch these types of errors in a generic sort of way
- If we “handle” these sorts of exceptions, we make our code “exception safe”

Bad Allocation Example

- Suppose we want to create an array of 10 integers, but we have a typo and we accidentally request allocation of -10 integers:

```
int *p = new int[-10];
```

- This makes no sense, and the allocation will fail, as demonstrated in the following example program
- A more common example of failed memory allocation would be if we ask for too much memory

```
// bad_alloc.cc
#include <iostream>
using namespace std;

int main() {
  int* p = new int[-10];
  p[0] = 42;
  delete[] p;
}
```

Program output:

```
terminate called after throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc
Abort (core dumped)
```

Handling the Exception

- The previous program crashes messily and we get a core dump (creating a potentially large file with a name like core.17820; the core file might be useful to the programmer for debugging purposes, but the user is just going to see it as clutter
- We can avoid the core dump if we “handle” the exception using the try/catch keywords:
  try { [some code to execute] }
  catch (catch-type) { [do this upon exception]}

```
// bad_alloc2.cc
#include <iostream>
using namespace std;

int main() {
  try {
    int* p = new int[-10];
    p[0] = 42;
    delete[] p;
  }
  catch (...) {
    cout << "Some exception occurred" << endl;
  }
}
```

Program output:
Some exception occurred

Discussion of try/catch Example

- We put the code that might cause an exception into a try{} block
- If an exception occurs within this block, then immediately upon the exception, the program skips to the code specified by the catch{} block
- The core dump is avoided because the exception is caught
- Immediately after the catch keyword, the type of exception to catch is specified in parentheses; ellipses (…) say to catch all possible exceptions
- This try/catch handling of the exception saves us from the core dump, but the error message is actually less informative than what we got before! Need less generic handling of the exception.

More Specific Catch Statements

- We can specify multiple catch{} statements after a try{}, with code appropriate to handle each of the types of exception that might have occurred in the try{} block. For example,
  try { }
  catch (exception_type_1) { … }
  catch (exception_type_2) { … }
  catch (…) { … } // all remaining exception types

- This will first try to handle exception_type_1, and if that doesn't match, it will then try to handle exception_type_2, and if that doesn't match, it will finally treat the exception generically and handle all remaining types together; only the fist matching catch() block will be executed

```
// bad_alloc3.cc
#include <iostream>
using namespace std;

int main() {
  try {
    int* p = new int[-10];
    p[0] = 42;
    delete[] p;
  }
  catch (std::bad_alloc& ex) {
    cout << "Bad allocation occurred: " << ex.what() << endl;
  }
  catch (std::exception& ex) {
    cout << "Some standard exception occurred: " << ex.what() << endl;
  }
  catch (...) {
    cout << "Some exception occurred" << endl;
  }
}
```

Program output:
Bad allocation occurred: std::bad_alloc

Catch Statements in the Example

- In the previous listing, we first try to catch exceptions of the type std::bad_alloc (corresponding to a failed memory allocation, which is the case here)
- If that hadn’t worked, we then would have tried exceptions of the type std::exception (which would be virtually all of them)
- If that somehow still didn’t work, we would catch all remaining exceptions with (…)

Common Types of Exceptions

The std::exception class is a base class for multiple types of exceptions, including:

- bad_alloc: An attempt to create new memory using the new keyword has failed
- bad_cast: An attempt to do a dynamic cast has failed because objects are not of the appropriate type for a dynamic cast
- ios_base::failure: Problems in functions from the iostream library

Because bad_alloc is of type std::exception, in our previous listing, either of the first two catch() statements would have been a match; but the first match found is executed and the others are ignored

Creating an Exception with throw

- The “throw” keyword does the opposite of catch: instead of handling an exception, it creates an exception
- You can “throw” an exception of any type, so long as there is a “catch” that knows how to catch that type (usually derived from std::exception, but technically could also be a simple string or an integer)

Creating a Custom Exception Class

- You might want to handle your own types of exceptions that aren't defined already in C++
- To do this, create a custom exception class; you will probably want to derive from std::exception, because then all existing handlers for that type will also work for your exception
- Alternatively, you could also inherit from std::logic_error (error related to the program logic) or std::runtime_error (error detected during runtime), which both inherit from std::exception
- With this exception class available, you can then throw an object of this type

Custom Exception Class Example

```
// custom_exception.cc
#include <exception>
using namespace std;

class MyException: public std::exception
{
  string Message;

  public:
    MyException(const char *msg):Message(msg) {}
    virtual const char* what() const throw() { return Message.c_str(); }
    ~MyException() throw() {}
};
```

Custom Exception Class Explanation

- The std::exception class defines a virtual function what() that should return an error message of what kind of exception occurred
- This function is normally defined in the class like this:
  virtual const char* what() const throw() {…}
- The “const” after what() just says the function doesn’t change the class
- The “throw()” right before the definition just means that this function itself is not supposed to throw an exception (otherwise we would be throwing an exception while handling another exception, not a good situation); anything in parentheses would indicate some type that the function *could* throw
- We also need to promise the exception class destructor will not throw, either, and that’s done with this:
  ~MyException() throw() { } // don’t really need to define further

Custom Exception Class in Action

(using the custom exception class MyClass() defined previously)

```
double SafeSqrt(double x) {
  if (x < 0.0)
    throw MyException("MyException: Tried to take sqrt of negative number");
  return sqrt(x);
}

int main() {
  double x = -4.0, y;
  try {
    y = SafeSqrt(x);
    cout << "The square root of " << x << " is " << y << endl;
  }
  catch (std::exception& ex) { // any std::exception incl. ours!
    cout << ex.what() << endl;
  }
}
```

Program output:
MyException: Tried to take sqrt of negative number

Uncaught Exceptions

- What if we throw an exception that we don't catch?
- Then we're back in the same situation we were with our first exception example, bad_alloc.cc : we get an error message, and possibly a core dump, but the program terminates and we are unable to proceed
- If we handle (catch) the exception, we at least have the option of doing something and then letting the program proceed
- The next listing shows what happens if we throw an exception using MyException but don't catch it

```
// from listing custom_exception2.cc
// MyException still defined as previously

double SafeSqrt(double x) {
  if (x < 0.0)
    throw MyException("MyException: Tried to take sqrt of negative number");
  return sqrt(x);
}

// run this to see what happens if exception is not caught
int main() {
  double x = -4.0, y;
  y = SafeSqrt(x);
  cout << "The square root of " << x << " is " << y << endl;
}
```

Program output:

```
terminate called after throwing an instance of 'MyException'
  what():  MyException: Tried to take sqrt of negative number
Abort (core dumped)
```

Which catch will Catch?

- Suppose we have an exception thrown inside multiple try{}/catch{} statements
- The throw statement will send the thrown exception “up the chain” (from the current, innermost try{} statement, outwards to enveloping try{} statements) until an appropriate catch{} is encountered
- The thrown exception will also percolate up from inner function calls to outer function calls (eventually all the way to main()) in search of an appropriate catch, so long as it remains within some try{} block

```
// from custom_exception3.cc
double SafeSqrt(double x) {
  if (x < 0.0)
    throw MyException("MyException: Tried to take sqrt of negative number");
  return sqrt(x);
}

void ComputeRoots() {
  try {
    cout << "Square root of  4 is " << SafeSqrt(4.0) << endl;
    cout << "Square root of -4 is " << SafeSqrt(-4.0) << endl;
  }
  catch (MyException& ex) {
    cout << "Caught a sqrt exception in ComputeRoots()" << endl;
  }
}

int main() {
  try {
    ComputeRoots();
  }
  catch (std::exception& ex) { // any std::exception incl. ours!
    cout << "Caught an exception in main()" << endl;
    cout << ex.what() << endl;
  }
}
```

Exception caught in this "inner" function, not in main()

Program output:

```
Square root of  4 is 2
Caught a sqrt exception in ComputeRoots()
```

```
// from custom_exception4.cc
double SafeSqrt(double x) {
  if (x < 0.0)
    throw MyException("MyException: Tried to take sqrt of negative number");
  return sqrt(x);
}

void ComputeRoots() {
  cout << "Square root of  4 is " << SafeSqrt(4.0) << endl;
  cout << "Square root of -4 is " << SafeSqrt(-4.0) << endl;
}

int main() {
  try {
    ComputeRoots();
  }
  catch (std::exception& ex) { // any std::exception incl. ours!
    cout << "Caught an exception in main()" << endl;
    cout << ex.what() << endl;
  }
}
```

Exception handler removed from ComputeRoots(), so now has to be caught in main()

Program output:

```
Square root of  4 is 2
Caught a sqrt exception in main()
MyException: Tried to take sqrt of negative number
```

When to Handle Exceptions?

- You should try to handle system-generated exceptions like bad allocation errors; otherwise, you'll just get a messy crash of the program and an error message that may not be as user-friendly as desired
- Creating your own exception classes and handling them may be overkill for handling what might be relatively simple runtime error situations; frequently, simpler is better. Additionally, exception handling can cause performance issues if not used sparingly. If an error should be handled immediately in the code where it was encountered, you don't need custom exceptions.
- Other times, you might not know in a subroutine how you want to handle an error, and you want to "kick it up the chain" and let the calling program decide what to do with it; custom exceptions can be appropriate for this kind of situation, if the situation is more complicated than a simple passed/failed status from the subroutine

Exceptions within Exceptions

- We want to avoid the possibility of having more than one exception at once (otherwise the application terminates)
- Don’t use very complex code inside a catch{}; the more complex it is, the more likely it could throw an exception (while handling another exception)
- Don’t throw exceptions from destructors, because when an exception is thrown, objects on the stack are destructed in reverse order of their construction: if one of them throws an exception upon destruction, we could get an exception within an exception