Advanced C++ Programming

Frank J. Edwards
Edwards & Edwards Consulting, LLC
September 2011

Advanced C++ Programming was developed and written by Frank Edwards of **Edwards & Edwards Consulting, LLC**.

Edwards & Edwards Consulting, LLC
21103 Bircholm Court
Land O'Lakes, FL 34637-7464
Voice: +1 813 406 0604
Internet: *http://www.eeconsulting.net/*
support@eeconsulting.net (add **student** to the subject line)
You can also find us on LinkedIn and Google+

Instructors and students alike are welcome to visit *http://www.eeconsulting.net/labs/* to retrieve the files for the labs. Note that the hands-on exercises expect the students to either (a) create their own files or (b) use text files that already exist on the system. Most students will choose (b), of course. :)

This book was edited using **Vim v7.2** (a free **vi** clone) under Linux 2.6.18 (POSIX) and/or AIX™ 5L/6.

Formatting was done using **Groff v1.20** and it's manuscript formatting macros, and the output was processed by **Grops** for conversion to PostScript®.

GNU **Make v3.81** was used to manage the commands required to format chapters and to automate the process of building DOS-formatted lab diskettes, where required.

GNU **RCS v5.7** was used as the source librarian for all documentation and source code.

In all cases, the code that appears in this courseware was compiled using at least one of the following compilers: the GNU **G++ gcc v2.96** (or later) compiler under Linux and/or AIX, and possibly other compilers, and included directly into the document with C++ comments used to control formatting appearance.

AIX and AIXwindows are trademarks of International Business Machines, Inc.

PostScript is a trademark of Adobe Systems Incorporated which may be registered in some jurisdictions.

Any other trademarks or registered trademarks are the property of their respective owners.

I have presented here a bibliography for the beginning C++ programmer. I cannot, of course, take any responsibility for whether they can teach a particular student any certain topic, but I have found them to be quite enlightening...

The following three texts tend to overlap some aspects of each other considerably. If you are only going to buy one, maybe get *The C++ Programming Language* since it contains a copy of the reference manual [minus the annotations]. If you can only buy two, maybe get the Lippman and *The Annotated C++ Reference Manual*, because the annotations tend to help explain the design of the language a lot. But, buy all three if you can afford it! Each has something unique to offer. Note that the *ARM* is now a bit dated since ANSI C++ 3.0 was ratified in early 1997, but it is still insightful.

C++ Primer, 4th Edition, Lippman, Addison-Wesley 2005, ISBN-13 978-0201721485. Historically, the most common text to learn C++ from. Great coverage of multiple inheritance. Great coverage of templates. Very readable (although I've had non-C programmers tell me they are not impressed with the writing style).

It is a large book (with a tiny font!) and that makes it daunting. But it is definitely the book to get if you can only get one.

The C++ Programming Language, 3rd Edition, Stroustrup, Addison Wesley 1997, ISBN 0-201-53992-6. The newest text from the creator of the language. Also includes the reference manual. If you can get two or more books, drop down the second part of this list and pick up a book or two on style issues, then come back up here and pick up more references.

Written at a fairly high level of technical detail — Lippman probably makes for a little easier reading. Serious C++ programmers will want to read this — at least to get a better idea of where Stroustrup is coming from. Good coverage of templates and exceptions, and lots of **practical** advice on how to use C++ on real projects.

STL Tutorial and Reference Guide, Musser, David R. and Saini, Atul, Addison-Wesley 1996, ISBN 0-201-63398-1. This is a hardbound book which describes the Standard Template Library, an implementation of generic algorithms which makes heavy use of templated classes and functions. As the STL is now packaged with every compiler product, the STL is always there and ready to go. On the downside, using the STL without a proper understanding of its design and implementation can lead to massive "code bloat", and the text points out key areas where this can be a problem and how to avoid them.

The text consists of three main parts: a tutorial to using STL, example programs written using STL, and an STL reference guide.

These are useful to teach style issues instead of language basics:

C++ Programming Style, Tom Cargill, Addison-Wesley 1994, ISBN 0-201-56365-7. This book looks at articles published in various magazines and examines the efficiency, readability, and reusability of the code presented in those articles. In most cases, the code ends up being completely rewritten. The author does this in multiple steps, explaining each change as the code develops. Very good for understanding the "why" and "how" aspects of code evolution in C++, although its treatment of templates is very light, due to its age.

Because it's such an old book, I hesitate to recommend it. However, I learned a lot from this book during my early years with C++.

UML Distilled, Third Edition, Martin Fowler, Addison-Wesley 2005, ISBN 0-321-19368-7 A great book on design, this one discusses how to use the Unified Modeling Language (UML) to represent common programming concepts when creating an object-oriented design.

I've had some students tell me they love this text and others tell me they hate it. I consider it an excellent, er, "distillation", of how to use the UML technique, including the graphical elements, to accomplish the analysis and design phases of an application. When I first picked up this book, I had worked a bit with the Booch Method, but hadn't used UML -- this book taught me both at once.

Effective C++, Scott Meyers, Addison-Wesley 1992, ISBN 0-201-56364-9. Fifty gems of wisdom that every C++ programmer needs to know and follow. Covers probably 90% of the questions that are asked on **`comp.lang.c++`** He now has an entire series of books, starting with *More Effective C++* and continuing through *Effective STL* (ISBN 978-0201749625). I'm a big fan of this series.

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, and Vlissides, Addison-Wesley 1994, ISBN 978-0201633610. This is a classic and anyone who has written at least a single line of code should have read this book by now. If you haven't, you probably don't understand some of the foundations of object-oriented programming.

Chapter 01
C++ Fundamentals

Chapter 02
Operator Overloading

Chapter 03
Templates

Chapter 04
Stream I/O

Chapter 05
Inheritance

Chapter 06
Standard Template Library

Chapter 07
Miscellaneous

TAB HERE

Advanced C++ Programming

C++ Fundamentals

Objectives

- Why Pursue Object-Oriented Programming?
- A Review of C++ and Classes
  - **`const`**
  - **`static`**
  - **`mutable`**
  - **`explicit`** constructors.
  - Virtual functions
  - Name mangling
  - The **`this`** pointer
- Using **`typedef`** inside a class
- Nesting Class Declarations

Why Object-Oriented Programming?

- Object-oriented programming provides a methodology for making software applications easier to write and maintain.
- These benefits come largely from the core concepts of
  - modularity (through design of class hierarchies),
  - data hiding (through the use of class access specifiers), and
  - code reuse (through inheritance and class derivation).
- By encapsulating key components of an application into small functional units, those units can be combined to produce the desired result, with the direct benefit of using those components in different ways within the same application or other applications.
- However, because the functional breakdown of an application into its constituent parts can only be performed when the application features have been well-defined, it is best that some sort of functional or requirements specification be available at design time.
- Typically, the specifications may change over the lifetime of the application, but without a vision of the goal, the path taken to achieve it will never be a direct one.

Review of C++ and Classes

- Over the next few pages we'll be reviewing the use of **`static`** and **`const`**, and introducing two new keywords not discussed in the *Introduction to C++* course, **`mutable`** and **`explicit`**.

- The **`static`** keyword is used when only a single occurrence of a class member should be created.

  — The **`static`** keyword is used on data members to indicate that all objects of a given class share access to the same storage location throughout their life-time.

  — This is commonly used to keep a reference count of the number of objects created, for example, or to hide any other kind of persistent data which is specific to a particular class.

- The **`const`** keyword tells the compiler that changes to the named variable are not allowed.

  — The variable to which the **`const`** modifier is applied cannot be changed, and any attempt by the software to do so should be flagged as an error.

  — This applies to functions as well, so that functions which are not declared **`const`** cannot be invoked on const (read-only) objects.

- The **`mutable`** keyword is the opposite of **`const`** in that it tells the compiler that a member variable may change even if the object is declared const.

  — This is primarily for those occassions in which an embedded application is required to control external hardware.

Review of C++ and Classes (continued)

- Constructors marked **`explicit`** are those whose parameters will not be converted from one type to another in order to satisfy the argument matching requirements.

  — This means, for example, that a constructor declared **`explicit`** and taking a float as a parameter, cannot be invoked by using the assignment form of initialization: **`Shape s = 3.0;`** would be illegal.

  — This is most valuable when an implicit constructor call would be inappropriate.

```
1  class String {
2    public:
3      String(int len);
4      String(const char *s);
5  };
6  String s = 'a';          // invokes first constructor!
```

  — The problem in the above code can be corrected by placing the word **`explicit`** in front of the first constructor declaration, informing the compiler that only explicit calls of that constructor are legal.

```
1  class String {
2    public:
3      explicit String(int len);
4      String(const char *s);
5  };
6  String s = 'a';          // error
7  String t(10);            // 10 character string
```

- Note that using **`explicit`** also prevents the compiler from using constructors to convert one data type to another. This is discussed in detail in a later chapter.

Review of C++ and Classes (continued)

- Now we present a few pieces of code that demonstrate the use of these keywords.
- We start with the use of **`static`** and **`const`**. Examine them here and we'll discuss them in more detail in a few pages.

```
bash$ cat demo-keywords1.h
     1  #ifndef demo_keywords1_h_
     2  #define demo_keywords1_h_

     3  #include "date.h"                // My Date & Time classes

     4  class Shape {
     5      static int count;            // How many Shapes exist?
     6      const Time timestamp;        // When was this Shape made?
     7  public:
     8      Shape();
     9      ~Shape();
    10      static int GetCount();
    11      const Time &GetCreationTime() const;
    12  };

    13  #endif // demo_keywords1_h_
bash$ _
```

Review of C++ and Classes (continued)

- In the code below, notice that one of the header files is missing the typical **.h** extension...

```
bash$ cat demo-keywordsm.cpp
     1  #include <iostream>              // Note the missing ".h"!!
     2  using namespace std;

     3  #include <unistd.h>              // For sleep()
     4  #include "demo-keywords1.h"

     5  int main()
     6  {
     7      Shape s1;
     8      cout << "Shape #" << Shape::GetCount()
     9           << " was created at "
    10           << s1.GetCreationTime().Text()
    11           << endl;
    12      sleep(5);                    // POSIX functionality
    13      Shape s2;
    14      cout << "Shape #" << Shape::GetCount()
    15           << " was created at "
    16           << s2.GetCreationTime().Text()
    17           << endl;
    18  #ifdef WIN32
    19      char ch;    // Otherwise the window may immediately close
    20      cin.get(ch);
    21  #endif
    22      return 0;
    23  }
bash$ demo-keywordsm
Shape #1 was created at 19:23:11
Shape #2 was created at 19:23:16
bash$ _
```

Review of C++ and Classes (continued)

- And on this page, you'll see how the static data member has space allocated for it (putting an assignment statement inside the class won't work and will be silently ignored).

  — If you leave out this definition, the compiler will not complain but the linker will.

  — The rule of thumb is to put the definition in the same file as the implementation of the class.

```
bash$ cat demo-keywords1.cpp
     1  #include "demo-keywords1.h"

     2  int Shape::count = 0;   // Definition of static member

     3  Shape::Shape() : timestamp(Time::Now())
     4  {
     5      count++;
     6  }

     7  Shape::~Shape()
     8  {
     9      count--;
    10  }

    11  int Shape::GetCount()
    12  {
    13      return count;
    14  }

    15  const Time& Shape::GetCreationTime() const
    16  {
    17      return timestamp;
    18  }
bash$ _
```

Review of C++ and Classes (continued)

- On this page is an example which includes the use the **`string`** class, defined by ANSI C++ 3.0. The **`string`** class, and the **`std::`** prefix which precedes it, will be discussed more in the next few pages.

```
bash$ cat date.h
     1  #ifndef date_h_
     2  #define date_h_

     3  #include <string>       // Yes, the ".h" is missing!

     4  class Date {
     5      unsigned short year, month, day;
     6    public:
     7      Date();
     8      void Set(unsigned short year,
     9               unsigned short month,
    10               unsigned short day);
    11      std::string Text() const;   //    mm/dd/yyyy
    12      static const Date &Now();   // returns current date
    13  };

    14  class Time {
    15      unsigned short hour, minute, second;
    16    public:
    17      Time();
    18      void Set(unsigned short hour,
    19               unsigned short minute,
    20               unsigned short second);
    21      std::string Text() const;   //    hh:mm:ss
    22      static const Time &Now();   // returns current time
    23  };

    24  #endif // date_h_
bash$ _
```

Some New Material

- In the next example, there are a few things which you may not have seen before.
  - Nested classes/structures,
  - **`typedef`**s inside a class declaration,
  - Templates (using the **`<`** and **`>`** characters),
  - the **`string`** class instead of character pointers,
  - the **`std`** scope (something called *namespaces*).
- Over the next few pages, you may see reference to the "STL". This is referring to the *Standard Template Library*, a component of the language which comes with ANSI C++ 3.0-compliant compiler products. The STL will be covered in more detail later.

Some New Material (continued)

```
bash$ cat demo-review1.h
     1  #ifndef demo_review1_h_
     2  #define demo_review1_h_

     3  #include <string>       // STL std::string and std::wstring
     4  #include <list>         // STL linked-list class
     5  #include "date.h"       // My own Date & Time classes

     6  class Empl {
     7  public:
     8      struct StartStop {  // nested structure declaration
     9          Date  cdStart, cdStop;
    10          float fSalary;
    11          StartStop(float newsal);        // constructor
    12      };
    13      typedef std::list<StartStop*>       dateList;
    14      typedef dateList::iterator          dateListIter;
    15      typedef dateList::const_iterator    dateListIterC;

    16  private:
    17      Empl  *pceMgr;      // this employee's manager
    18      std::string csName;
    19      std::string csSSN;
    20      std::string csTitle;
    21      int    nAge;
    22      int    nAuth;       // autonomous spending level
    23      dateList lWorkDates;
    24  public:
    25      Empl();
    26      Empl(std::string name, std::string ssn, Empl *mgr = 0);
    27      ~Empl();

    28      void SetName(std::string name)      { csName  = name; }
    29      void SetSSN(std::string ssn)        { csSSN   = ssn; }
    30      void SetTitle(std::string title)    { csTitle = title; }
    31      void SetAge(int age)                { nAge    = age; }

    32      Empl *GetManager() const            { return pceMgr; }
    33      std::string GetName() const         { return csName; }
    34      std::string GetSSN() const          { return csSSN; }
    35      std::string GetTitle() const        { return csTitle; }
    36      int GetAge() const                  { return nAge; }

    37      // The salary functions are handled differently
```

Some New Material (continued)

```
    38      // because they must access the lWorkDates member
    39      // and set or get the information from the last entry.
    40      void SetSalary(float salary);
    41      float GetSalary() const;

    42      const dateList GetEmploymentDates() const;
    43  };
    44  #endif // demo_review1_h_
bash$ _
```

- Notice the use of the **`string`** class and the inclusion of the **<string>** header file. (Because this class uses strings instead of character pointers, there is no need for a destructor to deallocate memory for the character fields.) Throughout this course, we will be using the **`string`** class. In a later chapter, we'll look at the specifics of this class in more detail.

Some New Material (continued)

- The *interface* to the Empl class was defined in the functional specification. The *implementation* of the Empl class can therefore change without modifying the code which uses this class.

- Notice the **`typedef`**s inside the class declaration. This *scopes* the typedef to within the class, so that it can't be used stand-alone and doesn't pollute the global namespace. They are **`public`**, however, and can be used outside the class.

- Notice how the *dateListIterC* data type is used below; a return value of that type is provided by the *GetEmploymentDates* method. The application can then iterate through the dates in any way supported by the iterator class. The real beauty is that the class can change the type of iterator at any time (maybe our container changed from a linked list to a binary tree?) and the application source code just needs a recompile to be updated – no code changes in the application.

```
 1  #include "demo-review1.h"
 2  // ...
 3  int main()
 4  {
 5      Empl fred;
 6      Empl::dateListIterC dateStart, dateEnd;

 7      dateStart = fred.GetEmploymentDates().begin();
 8      dateEnd   = fred.GetEmploymentDates().end();
 9      while (dateStart != dateEnd) {
10          // `*dateStart' is what the iterator contains;
11          // in this case, a "StartStop *"
12          // dereference that pointer to get the salary
13          if ((*dateStart)->fSalary > 50000) {
14              // ...
15          }
16          dateStart++;
17      }
18      // ...
19  }
```

- In addition to the *Empl* class on the previous page, let us suppose that another, derived class, is required to represent a specialization or augmentation of the *Empl* class. This class is shown next.

Some New Material (continued)

```
bash$ cat demo-review2.h
     1  #ifndef demo_review2_h_
     2  #define demo_review2_h_

     3  #include "demo-review1.h"

     4  using namespace std;            // Now I don't need "std::"

     5  class Manager : public Empl {
     6      typedef list<string> projList;
     7    public:
     8      typedef projList::const_iterator projListIterC;

     9    private:
    10      string   csDept;
    11      projList clcsProjects;
    12    public:
    13      Manager();
    14      Manager(string name, string ssn, Empl *mgr = 0);

    15      void SetDept(string dept);
    16      string GetDept() const;

    17      int AddProject(string proj);
    18      int RemoveProject(string proj);
    19      projListIterC ProjectList() const;
    20  };

    21  #endif // demo_review2_h_
bash$ _
```

- Notice how the list and iterator types could change without requiring any changes on the part of the application, as long as all iterators support the same interface.

- This is, in fact, what the Standard Template Library tries to accomplish. All iterators are grouped into various categories and each iterator of a given category must support particular operations.

Some New Material (continued)

- It is worth noting that a good practice is to use **`typedef`**s to create aliases for all data types based on concepts of what the type represents.

  — For example, a variable that contains a salary might be of type *Money* instead of *float*. This has a number of advantages in the future when the desire to change the data type from *float* to a class becomes reality.

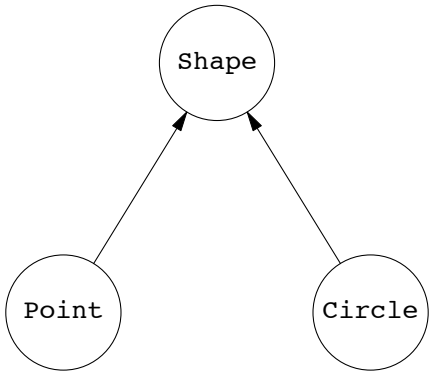
```
typedef float Money;

Money salary = 5.25;    // variable initialization
salary *= 1.10;         // give a 10% raise
```

could become the following in a transparent manner:

```
// typedef float Money;
class Money {
    // ...
};

Money salary = 5.25;    // constructor call
salary *= 1.10;         // calls operator*=(float);
```

Virtual Functions

- The next few pages contain an example of *polymorphism*; that is, the ability for a single item to assume many forms. In this case, the item is a Shape pointer, and the many forms would be Points and Circles.



- Polymorphism allows the application to create a list of pointers to the top class in the hierarchy and then store into the list a pointer to objects of any subtype. For example, an array of **`Shape *`** could contain pointers to shapes (of course), but could also contain pointers to **`Point`**s and **`Circle`**s.

- What virtual functions provide is a way for the runtime to figure out what type of object is actually being pointed to so that methods of the correct class are invoked. For example, both **`Shape`** and **`Circle`** contain **`draw()`** methods. But if all the compiler has is a **`Shape`** pointer it doesn't know what the real object type is... Virtual functions allow the runtime to figure out the actual object type and invoke the correct **`draw()`** method.

Virtual Functions (continued)

```
bash$ cat shapes1.h
     1  #ifndef shapes1_h_
     2  #define shapes1_h_

     3  class Shape {
     4      static int count;
     5    public:
     6      Shape();
     7      // If there are ANY virtual functions, there should
     8      // probably be a virtual destructor...  Why?
     9      virtual ~Shape();
    10      // pure virtual functions...
    11      virtual void draw() const = 0;
    12      virtual void rotate() const = 0;
    13  };

    14  #endif // shapes1_h_
bash$ _
```

```
bash$ cat shapes1.cpp
     1  #include <iostream>
     2  using namespace std;

     3  #include "shapes1.h"
     4  int Shape::count = 0;           // Definition of static

     5  Shape::Shape()
     6  {
     7      cout << "++Shape::count is " << ++count << endl;
     8  }
     9  Shape::~Shape()
    10  {
    11      cout << "Shape::count-- is " << count-- << endl;
    12  }
bash$ _
```

Virtual Functions (continued)

```
bash$ cat points1.h
     1  #ifndef points1_h_
     2  #define points1_h_

     3  #include <iostream>     // Needed for declaration of ostream
     4  using namespace std;

     5  #include "shapes1.h"

     6  class Point : public Shape {
     7      int x, y;
     8    public:
     9      Point(int nx, int ny);          // only constructor
    10      virtual void draw() const;      // always virtual
    11      virtual void rotate() const { } // do nothing
    12      int getX() const { return x; }  // inlined by optimizer
    13      int getY() const { return y; }  // inlined by optimizer
    14  };

    15  inline
    16  ostream &operator <<( ostream &os, const Point &p )
    17  {
    18      os << "[Point:" << p.getX() << ',' << p.getY() << "]";
    19      return os;
    20  }

    21  #endif // points1_h_
bash$ _
```

```
bash$ cat points1.cpp
     1  #include "points1.h"

     2  Point::Point(int nx, int ny) : x(nx), y(ny)
     3  {
     4  }

     5  void Point::draw() const
     6  {
     7      // Notice the use of "*this" ...
     8      cout << "Draw a point @ " << *this << endl;
     9  }
bash$ _
```

Virtual Functions (continued)

```
bash$ cat circles1.h
     1  #ifndef circles1_h_
     2  #define circles1_h_

     3  #include <iostream>     // Needed for ostream
     4  using namespace std;

     5  #include "shapes1.h"
     6  #include "points1.h"

     7  class Circle : public Shape {
     8      Point center;
     9      int radius;
    10    public:
    11      Circle(const Point &p, int rad); // only constructor
    12      virtual void draw() const;       // always virtual
    13      virtual void rotate() const { }  // do nothing
    14      const Point &getCenter() const { return center; }
    15      int getRadius() const { return radius; }
    16  };

    17  inline
    18  ostream &operator <<(ostream &os, const Circle &c)
    19  {
    20      os << "[Circle:" << c.getCenter()
    21         << ',' << c.getRadius() << ']';
    22      return os;
    23  }
    24  #endif // circles1_h_
bash$ _
```

Virtual Functions (continued)

```
bash$ cat circles1.cpp
     1  #include "circles1.h"

     2  Circle::Circle(const Point &p, int rad)
     3      : center(p), radius(rad > 0 ? rad : 0)
     4  {
     5  }

     6  void Circle::draw() const
     7  {
     8      cout << "Draw a circle @ " << *this << endl;
     9  }
bash$ _
```

- And now an example of using the previous classes. This (silly) program creates a few shapes and adds them to the front of a linked list. Then a few randomly positioned shapes (a Circle and a Point) are added at the end of the list.

Virtual Functions (continued)

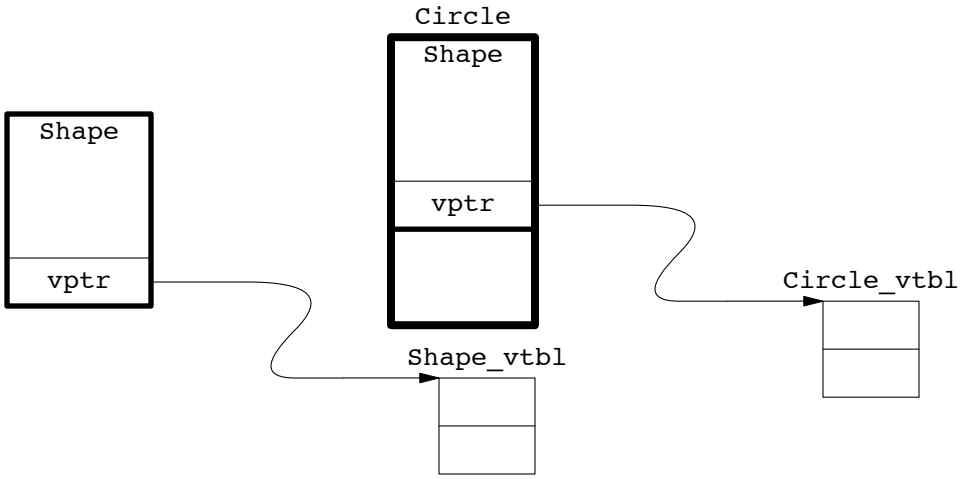
```
bash$ cat demo-virtfuncm.cpp
     1  #include <iostream>
     2  #include <list>         // Example of STL class

     3  using namespace std;

     4  #include <stdlib.h>
     5  #include "shapes1.h"
     6  #include "points1.h"
     7  #include "circles1.h"

     8  // Create a datatype that can be easily changed later
     9  typedef list<Shape*> ListOfShapes;

    10  int main()
    11  {
    12  //  Shape s;            // compile-time error
    13      Point a(1,1);               // @(1,1)
    14      Point b(10,35);             // @(10,35)
    15      Circle c(Point(50,50), 7);  // @(50,50), radius 7

    16      a.draw();   // draw() is called the same way for all of
    17      b.draw();   //   these objects (no pointers used).
    18      c.draw();

    19      cout << "Now create a list..." << endl;
    20      ListOfShapes allShapes;

    21      allShapes.push_front(&a);   // Pointers are stored...
    22      allShapes.push_front(&b);
    23      allShapes.push_front(&c);   // order is c -> b -> a

    24      {
    25          // Add two shapes to the end of the list.
    26          Shape *s;
    27          s = new Circle(
    28                  Point(rand()%100, rand()%100),
    29                  rand()%20);
    30          allShapes.push_back( s );
    31          s = new Point(rand()%100, rand()%100);
    32          allShapes.push_back( s );
    33      }
    34      cout << "Size should be 5: " << allShapes.size() << endl;
```

Virtual Functions (continued)

```
    35      // Draw all of the shapes in the list.
    36      // (Notice that we don't know the iterator's real type)
    37      ListOfShapes::iterator current;

    38      // Start at the first one and iterate until done.
    39      cout << "In front-to-back order..." << endl;
    40      current = allShapes.begin();
    41      while (current != allShapes.end())
    42              (*current++)->draw(); // '*current' is a Shape*
    43      cout << endl;

    44      // Now try it again in reverse order.
    45      cout << "In back-to-front order..." << endl;
    46      current = allShapes.end();
    47      do {
    48              (*--current)->draw();
    49      } while (current != allShapes.begin());
    50  #ifdef WIN32
    51      char ch;
    52      cin.get(ch);
    53  #endif
    54      // Do the shapes in the list need to be delete'd?
    55      return 0;
    56  }
bash$ _
```

Virtual Functions (continued)

- Polymorphism doesn't come without a price, however.
- Somehow the runtime has to know what the real object type is when all it has is a pointer to a base class.
- The answer is rather simple — the runtime **doesn't** know what the data type really is!
- What the runtime **does** know is that the base class contains an extra pointer field hidden inside the class (usually at the end of the memory layout). This pointer was added by the compiler when it saw that the class had a virtual function in it.
- That pointer is the location of a jump table for this object (and all objects of the same type).
- This means that the overhead of using virtual functions equates to an additional pointer inside **every** object which is polymorphic (called the **`vptr`**), and the space required for the jump table (called the **`vtbl`**) which is shared by all objects of the class, so that the overall space requirement is negligible.
- There are also two extra memory fetches: one to retrieve the **`vptr`** from the object and one to get the function address from the **`vtbl`**.



Virtual Functions (continued)

- The upstart of all this is that your objects will increase in size by the size of a void pointer (4 bytes on most 32-bit platforms, and 8 bytes on 64-bit platforms). For small and plentiful objects, this could be significant. For larger objects of which fewer are created, the differential is much smaller.

- The **`vtbl`** will contain one void pointer for each virtual function declared within the class or its base class(es), so it's size is negligible.

- To demonstrate these size differences, examine the following code.

```
bash$ cat sizeof.cpp
     1  #include <iostream>
     2  using namespace std;

     3  #include "sizeof.h"

     4  int main()
     5  {
     6      cout << "Base:                    "
     7           << sizeof(Base) << endl;
     8      cout << "WithoutVirtual:          "
     9           << sizeof(WithoutVirtual) << endl;
    10      cout << "WithVirtual:             "
    11           << sizeof(WithVirtual) << endl;
    12      cout << endl;
    13      cout << "BaseWithData:            "
    14           << sizeof(BaseWithData) << endl;
    15      cout << "WithoutVirtualWithData: "
    16           << sizeof(WithoutVirtualWithData) << endl;
    17      cout << "WithVirtualAndData:      "
    18           << sizeof(WithVirtualAndData) << endl;
    19  #ifdef WIN32
    20      char ch;
    21      cin.get(ch);
    22  #endif
    23      return 0;
    24  }
bash$ _
```

Virtual Functions (continued)

```
bash$ sizeof
Base:                   1
WithoutVirtual:         1
WithVirtual:            4

BaseWithData:           4
WithoutVirtualWithData: 4
WithVirtualAndData:     8
bash$ _
```

Virtual Functions (continued)

```
bash$ cat sizeof.h
     1   #ifndef sizeof_h_
     2   #define sizeof_h_

     3   // How many bytes does your compiler allocate?
     4   class Base {
     5       // no data at all!
     6     public:
     7       void fill() const;
     8   };

     9   // What is your guess for sizeof(WithoutVirtual)??
    10   class WithoutVirtual : public Base {
    11     public:
    12       void rotate() const;
    13   };

    14   // What is your guess for sizeof(WithVirtual)??
    15   class WithVirtual : public Base {
    16     public:
    17       virtual void rotate() const;
    18   };

    19   // How many bytes for this one?
    20   class BaseWithData {
    21       int test;
    22     public:
    23       void fill() const;
    24   };

    25   // What is your guess for sizeof(WithoutVirtualWithData)??
    26   class WithoutVirtualWithData : public BaseWithData {
    27     public:
    28       void rotate() const;
    29   };

    30   // What is your guess for sizeof(WithVirtualAndData)??
    31   class WithVirtualAndData : public BaseWithData {
    32     public:
    33       virtual void rotate() const;
    34   };
    35   #endif // sizeof_h_
bash$ _
```

Name Mangling

- *Name mangling* is the term used to describe how the compiler modifies the symbol name seen by the linker to include information necessary to allow the linker to determine the correct function out of a list of overloaded functions.

- It is compiler's implementation of a language component known as the *function signature*. A function signature is a uniquely identifying attribute of a function which can be determined solely by the function declaration and which includes the function name (including scope) and number and types of all parameters, plus any function modifiers such as **`const`**.

- Basically, it works like this:

  — The C++ compiler **requires** all functions to be prototyped (so that it has a complete list of all functions that are being overloaded).

  — It chooses the appropriate function from the list of prototypes when a function call is made in the source code (determined via the function signature).

  — This results in an "external reference" being made in the resulting object file (with a **.obj** extension on some platforms, a **.o** extension on others).

  — The linker locates that name from the list of all modules available for input, including the object file itself as well as other object files and/or libraries.

- The compiler will produce an error if the function hasn't been prototyped.

- The linker will produce an if error the function hasn't been defined.

- In the case of a function being both declared and defined, the application linking is successful and an executable is generated.

- The next page is a (summarized) listing from a Unix linker of a single object file. The name mangling is quite apparent.

Name Mangling (continued)

```
bash$ cat name-mangling.cpp
     1  class Shape {
     2      void InitShape();
     3    public:
     4      ~Shape();           // destructor

     5      void rotate();
     6      virtual void draw();
     7  };

     8  int main()
     9  {
    10      Shape s;            // Force some references,
    11      s.draw();           // and some more...
    12  }
bash$ _
```

```
ADDRESS MAP FOR a.out
CL TY Sym#  NAME                    SOURCE-FILE(OBJECT)
-- -- ----- --------------------- ------------------
PR ER S1    .__dt__5ShapeFv
UA ER S2    __vft5Shape
PR ER S3    .draw__5ShapeFv
PR SD S4    <>                      name-mangling.cpp(name-mangling.o)
PR LD S5    <.__dftdt__5ShapeFv>
PR LD S6    <.__ct__5ShapeFv>
PR LD S7    .main
RW SD S8    <_$STATIC>              name-mangling.cpp(name-mangling.o)
DS SD S9    main                    name-mangling.cpp(name-mangling.o)
DS SD S10   <__dftdt__5ShapeFv>     name-mangling.cpp(name-mangling.o)
TO SD S11   <TOC>
TC SD S12   <__vft5Shape>
TC SD S13   <_$STATIC>
```

Pointers to Class Members

- One topic doesn't come up very often in discussing class members. That is the ability to create a pointer to a object's members.
- This is not an ordinary pointer, because
  - a pointer to a data member must be based on an offset from the beginning of the object instead of an absolute pointer to a memory location, and
  - a pointer to a virtual function member must take into account the access to the *vptr* and *vtbl* of the class.
- A pointer to the *draw()* method of the *Shape* class is shown being used below.
- The idea that "the pointer is not a simple address" only applies to non-static data members and virtual function members.
  - Static members exist without relation to individual objects, so they are not implemented as offsets from the beginning of an object, but as constant addresses.
  - And non-virtual functions exist in memory at a single location for all objects of the class as well, also resulting in an absolute address.

```
bash$ demo-ptm
++Shape::count is 1
++Shape::count is 2
++Shape::count is 3
++Shape::count is 4
++Shape::count is 5
++Shape::count is 6
sizeof(pfm) = 16
Draw a point @ [Point:7,49]
Draw a point @ [Point:73,58]
Draw a point @ [Point:30,72]
Draw a point @ [Point:44,78]
Draw a point @ [Point:23,9]
Draw a point @ [Point:40,65]
bash$ _
```

Pointers to Class Members (continued)

```
bash$ cat demo-ptm.cpp
     1  #include <iostream>
     2  #include <list>
     3  using namespace std;

     4  #include "points1.h"

     5  // Pointer to Shape Member Function
     6  // (some compilers ignore the "const")
     7  typedef void (Shape::*PSMF)() const;

     8  // Prototype a function that operates on all shapes in a
     9  // list of shapes, invoking some function on each one.
    10  void draw_em(PSMF func, const list<Shape*> &drawing);

    11  int main()
    12  {
    13      list<Shape*> drawing;

    14      // Add 6 Points to the drawing
    15      for (int i=0; i < 6; i++) {
    16          Shape *s = new Point(rand()%100, rand()%100);
    17          drawing.push_back(s);
    18      }
    19      PSMF pfm = &Shape::draw;    // Notice no parens
    20      cout << "sizeof(pfm) = " << sizeof(pfm) << endl;
    21      draw_em(pfm, drawing);
    22      return 0;
    23  }

    24  void draw_em(PSMF func, const list<Shape*> &drawing)
    25  {
    26      list<Shape*>::const_iterator current = drawing.begin();
    27      while (current != drawing.end()) {
    28          Shape *a = *current++;
    29          // Function name is unknown here; it is provided as
    30          // the first parameter to this subroutine, as a
    31          // pointer, hence the "*func" notation, below.
    32          (a->*func)();
    33      }
    34  }
bash$ _
```

Review

- Why objects?
  - data hiding / abstraction
  - encapsulation / modularity
  - code reuse / inheritance / polymorphism
- A Review of:
  - **`const`**
  - **`static`**
  - Virtual functions
  - Name mangling
  - The **`this`** pointer
- Using **`typedef`** inside a class
- Nesting Class Declarations
- Pointers to members

Advanced C++ Programming

Lab for C++ Fundamentals

Lab for C++ Fundamentals

1. Goals for this lab:

   A. Explore the use of `static` data members.

   B. Try using `const` data members within a class.

   C. Use the basic facilities of the STL **`list`** and **`string`** classes.

2. In this lab you will be developing an employee class similar to the one used in the *Introduction to C++* course. However, since you already know most of the language basics, we'll be jumping right in!

3. Here are your functional requirements. You are to do your own design based on these requirements. The requirements will change over the next few labs and you will be adding some code — although hopefully the requirements don't change too drastically!

   A. Create an *Empl* class to record certain information about employees.

   B. For our purposes, this is first and last name, and salary. (Our example class will become more complete later.)

   C. Provide for initializing the employee with known information, i.e. the first name, last name, and current salary are all known in advance.

   D. Also provide for initializing an empty employee object.

   E. Provide a *Print* member function that will be invoked to print the object to whatever stream is given as a parameter. (Hint: the data type of **`cout`** is `ostream`.)

   F. Test the code by creating some objects inside *main* and calling their *Print* functions and passing **`cout`** as a parameter. Some sample test code for *main* is shown next.

Lab for C++ Fundamentals (continued)

```
// ...

int main()
{
    Empl frank("Frank", "Edwards", 250000.00);
    Empl empty;          // NO PARENTHESES!  Why not?

    fixed.Print(cout);
    return 0;
}
```

4. Now that you've got the basic class working, we're going to make some changes and try out some things.

   A. Create a constant employee object inside *main*. Initialize the employee with a first and last name. Did it work?

   B. Call the **`Print`** function of that employee so that the information is displayed. What happened? Discuss the problem with the other students if you can't explain it and work out a theory. Make changes to fix it, if it needs fixing.

   C. Change the first and last name in the employee class to **`const`** strings. Unless you were careful when you coded the constructor, it should no longer compile. Why not? Can you fix it so that it works correctly? Why would you want to make class members constants?

   D. Remove the word **`const`** from the employee's first and last names. Now add a counter which keeps track of how many employees are created. Everytime a constructor is called, the counter should be incremented. We don't want to know how many currently exist, but how many have ever been created, so we don't need to decrement the counter at all. But add some output statements to the constructor(s) so that the value of the counter is displayed when employees are created.

5. **Extra Credit:**

   A. Modify the main program so that it is capable of reading in as many employees as desired from the keyboard. You will need to create a *list* of employee objects.

Lab for C++ Fundamentals (continued)

B. Print out a list of the employees that were read in using an iterator of whatever data type you used in the last step (could have been a *list*, *vector*, or *deque*).

C. Change the data type you used in the last two steps to one of the other data types. Did anything else in the program have to change, or did all you have to do was recompile?

6. All done!

TAB HERE

Advanced C++ Programming

Operator Overloading

Objectives

- What is operator overloading
- The functionality behind overloading
- Special function operators
- Some operators **must** be members, others **cannot** be

What is Operator Overloading?

- *Operator overloading* is the ability to define for a class how various language operators should function.

- For example, the language itself defines the meaning of the **+** character when used with numbers, but the programmer may wish to provide his own meaning when the addition operator is used with a new class.

- Operator overloading provides the ability to make such a change, byt defining a function which will be called when the compiler sees the plus sign.

- First, the compiler translates all operators into function calls, but only when the operand(s) include at least one user-defined data type.

  — For instance, there is no need for the compiler to change the following into a function call, since integers are not user-defined types.

```
int a=0, b=1, c;
c = a + b;
```

  — However, the translation into function calls of operations between user-defined object types results in the following:

```
complex a(1,0), b(2,3);

complex c = a + b;
// c = a.operator+(b);  // when operator is a member of complex
// c = operator+(a, b); // when operator is a global function
```

  — Unary operators are similar, but there is only a single parameter to the overloaded operator function.

```
complex c = !a;          // (for example, cin)
// c = a.operator!();    // when operator is a member of complex
// c = operator!(a);     // when operator is a global function
```

What is Operator Overloading? (continued)

- Second, the compiler always chooses a member function over a global function, if the choice exists.

  — This allows the programmer to define a global operator to handle a (possibly large) category of situations, and fine-tune it with class-specific operators as needed.

  — The scope searched for an operator function is the same as all functions: the scope of each parameter. Then function overloading occurs as normal, based on the results of the scope searches. (This is new behavior in ANSI C++ 3.0 compared to previous versions of the language.)

  — Additionally, the compiler will attempt to promote data types and perform single-step data type conversions of the operands, just as it does with all function overloading. In essence, operator functions are just like any other functions, but with a different calling syntax.

What is Operator Overloading? (continued)

- Some operators **must** be member functions and other operators **cannot** be member functions!

  — In particular, the following functions must be non-static member functions. This guarantees that their first operands will be lvalues, ie. the object being modified must exist.

`operator=(T)`	`c = a;`
`operator[](int) const` `operator[](int)`	`int i = a[0];` `a[0] = 42;`
`operator()(int, int) const` `operator()(int, int)`	`string s = a(3,6);` `a(3,6) = "abcd";`
`operator->()`	`int i = a->draw();`

  — In addition, operators which take a basic data type as their first parameter cannot be member functions (for example, in the statement **`i * k - z`**; if *i* is an `int`, you can't declare a class called **`int`**!).

  — Also, operators cannot be defined which take only pointers as parameters, since implicit pointer conversion would make function choice impossible for the compiler.

What is Operator Overloading? (continued)

- The meanings of some built-in operators are defined to be equivalent to some combination of other built-in operators. For example, the operator **+=** is the same as calling both addition and assignment operators.

```
int a, c;

c += a;          // c.operator+=(a);
                 //      same as
c = c + a;       // c.operator=( c.operator+(a) );
```

- This is not necessarily true of user-defined operators unless the programmer defines them this way. In general, it is a good idea to always do so!

- Operators can be made inaccessible by making them member functions and placing them inside the private or protected section of the class. The access specifier used will control access to the member functions.

- Further, by declaring *but not defining* those functions, if the programmer should accidentally use those operators in the construction of their own class (!), the linker will complain that the functions have not been defined.

Examples

- You've already seen an example of how **`operator+`** might be used, but you have not seen it's declaration.

- There are a few other operators that may appear unusual at first glance, but are quite useful...

```
 1  class String {
 2      // ...
 3    public:
 4      String& operator=(const String &s);          // assign
 5      String& operator+=(const String &s);         // self-add
 6      char  operator[](int loc) const;             // index (r)
 7      char& operator[](int loc);                   // index (w)
 8      String  operator()(int start, int len) const;// extract
 9      String &operator()(int start, int len);      // modify
10      // ...
11  };

12  void f()
13  {
14      String f("Frank"), b;
15      String x;
16      char ch;

17      x = f;              // x.operator=(f)
18      x += " Edwards";    // x.operator+=(String(" Edwards"))
19      ch = x[5];          // x.operator[](5) const
20      x[5] = '_';         // x.operator[](5)
21      b = x(3,5);         // x.operator()(int, int) const
22      x(3,5) = "ZZ";      // x.operator()(int, int)
23  }
```

Optimizations

- In the following code, each statement is functionally the same:

```
1  class Money {
2      // ...
3    public:
4      Money(double val=0);
5  };

6  Money a = Money(3);      // calls Money(3)
7  Money b = 3;             // calls Money(3)
```

- The advantage to this type of constructor (called a *conversion constructor*) is that they allow a double to be used where a *Money* would be required, and the compiler can implicitly invoke these constructors to perform conversions automatically. Note that this is exactly what the **`explicit`** modifier for constructors is designed to prevent.

- For example, let's assume that the *Money* class requires an **`operator==`** for comparing monetary values as shown below. We have overloaded the operator to account for comparisons against doubles as well as *Money* objects.

```
 1  class Money {
 2      // ...
 3  };
 4  bool operator==( Money  a, Money  b );  // global functions
 5  bool operator==( double a, Money  b );
 6  bool operator==( Money  a, double b );

 7  int main()
 8  {
 9      Money x, y;
10      x == y;      // calls operator==(Money, Money)
11      3 == y;      // calls operator==(double, Money)
12      x == 3;      // calls operator==(Money, double)
13  }
```

- In this situation, we need a different overloaded function for each possible combination of data types. This can get tedious, and what is tedious is error-prone.

Optimizations (continued)

- Instead, the conversion constructors allow us to declare a single overloaded operator and let the compiler call the conversion constructor for us.

```
 1  bool operator==( Money a, Money b );

 2  int main()
 3  {
 4      Money x, y;
 5      x == y;      // calls operator==(x, y)
 6      3 == y;      // calls operator==(Money(3), y)
 7      x == 3;      // calls operator==(x, Money(3))
 8  }
```

- At first glance, this might appear to increase the amount of code that might be generated by the compiler, tending to make larger executables. But an optimizing compiler can eliminate the function calls when the functions are defined inline, eliminating the function call overhead as well as a lot of redundancy, as illustrated below.

```
 1  // Assuming that the operator is declared inline ...
 2  inline bool operator==( Money a, Money b )
 3  {
 4      return( a.value == b.value );
 5  }

 6  // So if we used this:
 7      if (x == 3)
 8          do_something();

 9  // It would be converted first into this:
10      if (operator==(x, Money(3)))
11          do_something();

12  // Which will inline the construction of the temporary:
13      if (operator==(x, {value=3} ))
14          do_something();

15  // Which is optimized to just this comparison:
16      if (x.value == 3)
17          do_something();
```

Optimizations (continued)

- One last operator can be usefully applied in the area of optimization.
- The *conversion operator* is an operator which returns an object of a different type for the purpose of data type conversion. (Our previous discussion was on *conversion constructors*, not *conversion operators*.)

```
bash$ cat exam-ch2e1.h
     1  class String {
     2      int length;          // length of string stored at 'str'
     3      char *str;           // location of string
     4      // ...
     5    public:
     6      explicit String(int size);
     7      String(const char *s);

     8      operator int() const;
     9      operator const char*() const;
    10      // ...
    11  };
bash$ _
```

```
bash$ cat exam-ch2e2.cpp
     1  #include "exam-ch2e1.h"

     2  String::operator int() const
     3  {
     4      return length;
     5  }

     6  String::operator const char *() const
     7  {
     8      return str;
     9  }
bash$ _
```

Optimizations (continued)

```
bash$ cat exam-ch2e3.cpp
     1  #include "exam-ch2e1.h"

     2  void function(int howmany);     // overloaded function
     3  void function(const char *where);

     4  int main()
     5  {
     6      String test = "Frank Edwards";
     7      int len;

     8      len = test;          // calls "test.operator int() const"
     9      function(test);      // ambiguous (const char* or int?)
    10      return 0;
    11  }
bash$ _
```

Interesting Operator Uses

- There are a few operators with particularly interesting uses.
- The first is **`operator[]`**, since that is the operator used to access subscripts.
- Imagine a class which defines **`operator[]`** to take a single **`char*`** parameter.
  - Now assume that the operator uses that parameter to do a lookup in a disk file.
  - Perhaps the object caches the information retrieved, saving any changes and writing them back to disk when the object is destroyed.
  - If memory ever got low, objects of the class could be asked to flush themselves to disk and free the associated memory.
  - In addition, each access through the operator could update a timestamp field in the object. The timestamp would be used to decide which objects are asked to free their memory.

Interesting Operator Uses (continued)

- Another possibility is a class which defines **`operator[]`** and acts as a mapping function. The parameter going into the operator is manipulated and the return value is some associated piece of information.

  — For example, a name goes in and a phone number comes out.

  — Or a database record goes in (as an object of a separate class) and the return value describes where that object is stored on disk.

```
 1  class Map {
 2    public:
 3      // ...
 4      Obj  operator[]( const char *name ) const;
 5      Obj& operator[]( const char *name );
 6  };

 7  int main()
 8  {
 9      Map map;
10      Obj x;
11      x = map["Frank"];   // first operator[]
12      map["Frank"] = x;   // second operator[]

13      return 0;
14  }
```

  — Note that the second **`operator[]`** declaration above returns a reference, which provides the lvalue needed by the compiler to make the associated assignment statement work.

Interesting Operator Uses (continued)

- Another interesting operator is **`operator->`**.
- This operator is a unary operator (strange as that may seem; see the example below).
- It returns a pointer to another object such that **`->`** is valid.
  - — It might return a pointer to an ordinary object.
  - — It could also return a pointer to an object which itself defines **`operator->`** and the technique would be used again inside that object's operator.

```
 1  class Empl { ... };     // defined with employee info

 2  class Ptr {
 3      string identifier;
 4      Empl *in_core_address;
 5    public:
 6      Ptr(string &n) : identifier(n), in_core_address(0) { }
 7      ~Ptr();             // free up the memory...
 8      Empl *operator->(); // allocate memory & read from disk
 9  };

10  int main()
11  {
12      Ptr data = "Frank";

13      data->Age = 35;
14      // (data.operator->())->Age = 35;
15      // Empl could also define operator->() !
16  }
```

  - — In the above example, the operator could be required to look up the information on disk and retrieve it, storing the address in its own data member.
  - — Upon destruction, the information would be flushed from memory. In the meantime, the operator would provide access to the underlying information.
- The most interesting thing about this operator is that the return value is used again by the compiler after plugging it into the original statement.

Interesting Operator Uses (continued)

- Just as with other operators, there is an equivalence among **`operator[]`**, **`operator->`**, and **`operator*`** (not discussed) when used with built-in data types.

- This equivalence is expected by the typical programmer, so breaking this tradition in any user-defined class is ill-advised.

- Take a common iterator class from the STL, for example. The interface would typically be declared something like this:

```
 1  class list_iterator // ...
 2  {
 3      typedef list_iterator       self;
 4      typedef list_node_datatype  reference;
 5      typedef list_node_datatype *pointer;
 6      // ...
 7      bool      operator==(const self& x) const;
 8      bool      operator!=(const self& x) const;
 9      reference operator*() const;
10      pointer   operator->() const;
11      self&     operator++();     // "&" is kludge to make
12      self&     operator++(int);  //   these operators work
13      self&     operator--();     //   "
14      self&     operator--(int);  //   "
15      // ...
16  }
```

Interesting Operator Uses (continued)

- Another example would be something called *auto_ptr* by the STL. This is a templated class that implements a "smart pointer" that automatically deletes the memory it points to when it goes out of scope.

```
 1  #include <iostream>     // For 'cout'
 2  #include <memory>       // For 'auto_ptr'
 3  #include <string>       // For 'string'

 4  int main()
 5  {
 6      // 'tmp' holds the pointer-to-string...
 7      std::auto_ptr<std::string>
 8          tmp(new std::string("Fred Flintstone"));

 9      // Dereference 'tmp' to get the actual object.
10      std::cout << *tmp << std::endl;
11      return 0;
12      // The object 'tmp' goes out of scope here, so the
13      // auto_ptr's destructor will automatically delete the
14      // memory that was allocated.
15  }
```

- More details on **`auto_ptr`**s can be found in the *STL Tutorial and Reference Guide* and in a later chapter.

Advice for Operator Overloading

1. Define operators primarily to mimic conventional usage.
2. For large operands, use **`const`** reference argument types.
3. For large results, consider optimizing the return.
4. Prefer the default copy constructor. Otherwise,
5. Redefine or prohibit the use of copying.
6. Prefer member functions if access to the implementation is needed.
7. Use non-member functions for symmetric operators.
8. Use **`operator()`** for multi-dimensional arrays. (Since **`operator[]`** only works with a single argument.)
9. Constructors which take a single "size" argument should be **`explicit`**.
10. Use the standard **`string`** class over any local implementation unless a specialized requirement exists.
11. Be cautious about introducing implicit conversions.

Review

- What is operator overloading?
- What features does it provide?
- Which operators have special considerations?

Notes:

Advanced C++ Programming

Lab for Operators

Lab for Operators

1. Goals for this lab:

   A. Experiment with overloading the math operators, such as addition and subtraction (**+** and **-**), and their associated assignment operators (**+=** and **-=**).

   B. Try overloading the assignment operator (**=**).

   C. Overload the **`char*`** conversion operator.

2. In this lab, we explore operators which might not seem to make much sense when applied to our *Empl* class. For example, the addition and subtraction operators. But we're going to define those to mean the addition or subtraction of the associated salaries of the two employees that are involved (yeah, I know it's a stretch of the imagination to ever consider such a scenario!). A couple of more useful examples might be overloading the assignment operator and/or a conversion operator. So we'll do that as well.

3. After implementing the last set of functional specs (our previous lab), we have realized that certain features would make the class much easier to use. Among them, the ability to add two employees together and obtain their combined salary. So you're going to overload **`operator+`** to do that. It will take two employees and return a float. Write the code and test it.

4. What happens if you try to add three employees together using your operator? For example:

```
// ...

int main()
{
    Empl fred("Fred", "Flint", 38000.00);
    Empl barney("Barney", "Rubble", 36000.00);
    Empl wilma("Wilma", "Flint", 0.00);

    cout << "fred + barney + wilma is "
         << fred + barney + wilma
         << endl;
    return 0;
}
```

Lab for Operators (continued)

5. How can you fix that problem?

   The return value from the **`operator+`** is a float, and the compiler doesn't know how to add a float and an employee together. You could return an employee from your operator, but then the return value couldn't be stored into a float!

   What about an overloaded **`operator+`** which takes an *Empl* as the first parameter and a float as the second? Would that work? Try it and see. But think about the "member operator vs. non-member operator" issue.

6. There is another solution, though. Suppose the *Empl* class had an operator that returned a float when used in a numeric context? Would that suffice in this case? Then the **`operator+`** wouldn't even be needed, since the employee would be implicitly converted. Try implementing this solution. (Note: I'm **not** guaranteeing it will work, so just give it your best shot and ask the instructor if you get stuck!)

TAB HERE

Advanced C++ Programming

Templates

Objectives

- What are templates?
- Template instantiation
- Template parameters
- Function Templates
- User Specializations
- Source Code Organization

What are templates?

- Independent concepts should be independently represented and should be combined only when needed.
- Templates provide a simple way to represent a wide range of general concepts and simple ways to combine them. The resulting classes and functions can match hand-written, more-specialized code in run-time and space efficiency.
- Templates provide direct support for *generic programming*, the idea that an algorithm should be written independent of representation details and without logical contortions. If possible, this provides for reusing the code with multiple various data types without further modification.
- For example, consider the concept of a *stack*.
- It represents the idea of being able to add data at the end of a list and retrieving the data only from the same end (last-in, first-out).
- It is certainly possible to envision stacks of characters, stacks of integers, stacks of floats, even stacks of **`strings`** or stacks of **`Empl`**s.
- This makes a templated stack class a reasonable technique. Consider the implementation on the following pages.

What are templates? (continued)

```
bash$ cat demo-stack1.h
     1  #ifndef stack1_h_
     2  #define stack1_h_

     3  template <class T>
     4  class Stack {
     5      T* v;               // what does the constr have to do?
     6      int max_size;       // maximum size
     7      int top;            // current position
     8    public:
     9      // These are used for exceptions (separate chapter).
    10      class StackUnderflow { };
    11      class StackOverflow { };

    12      Stack(int size) : max_size(size), top(0) {
    13          v = new T[max_size];    // Assumes T has def constr
    14      }
    15      ~Stack() {
    16          delete [] v;
    17      }
    18      void push(T value);         // notice the use of the
    19      T pop();                    //   letter "T"...
    20  };

    21  #ifndef FIXED_COMPILER          // See text re: "export"
    22  # include "demo-stack1.inc"
    23  #endif
    24  #endif // stack1_h_
bash$ _
```

- Notice the **`template`** line immediately above the class declaration? That tells the compiler that any reference to the letter **`T`** should be replaced by whatever data type is actually provided by the programmer when they use the class.

- You should also note that the data type **`T`** appears multiple times throughout the class. Once as a data type for a private member field, once inside the constructor as a data type to allocate memory for, and once each in the **`push()`** and **`pop()`** methods, first as a parameter data type and then as a return type.

- The strange **`#ifndef`** is because some compilers don't properly implement templates and they need to see the source code for all methods of templated classes. If your compiler works, you can define the macro and the **`#include`** will be skipped. More discussion is after the next example.

What are templates? (continued)

```
bash$ cat demo-stack1.inc
     1  #include "demo-stack1.h"

     2  template <class T>
     3  // Assumes elements of type T can be copied and assigned.
     4  void Stack<T>::push(T c)
     5  {
     6      if (top == max_size) throw StackOverflow();
     7      v[top++] = c;
     8  }

     9  // Assumes elements of type T can be copied.
    10  template <class T>
    11  T Stack<T>::pop()
    12  {
    13      if (top == 0) throw StackUnderflow();
    14      return v[--top];
    15  }
bash$ _
```

- Again you'll notice the **`template`** statement, this time in front of a function definition.

- It means the same thing as it did previously: any reference to **`T`** will be replaced with whatever data type the programmer provides when they instantiate the **`Stack`** class. (Hang on, we'll discuss this more on the next page!)

- If you look at the function definitions themselves, you'll see that the templated class name, **`Stack`** in this case, has **`<T>`** after its name. That's because the class is a *template class* that requires a data type for the compiler to complete the definition.

- As mentioned on the previous page, if your compiler is broken then this source code is included directly into the header file and everything will work. If your compiler works as the language intended, then you would define the macro `FIXED_COMPILER` and create a new source file called **demo-stack1.cpp** which only contains a single line that includes the above file. (Hence, all source code for the class is in the `.cpp` file.)

What are templates? (continued)

```
bash$ cat demo-stack1m.cpp
     1  #include "demo-stack1.h"
     2  #include <iostream>

     3  int main()
     4  {
     5      Stack<char> sc(5);          // stack of chars
     6      Stack<float> sf(10);        // stack of floats

     7      sc.push('c');
     8      std::cout << sc.pop() << std::endl;

     9      sf.push(3.1415);
    10      std::cout << sf.pop() << std::endl;

    11      return 0;
    12  }
bash$ _
```

```
bash$ demo-stack1m
c
3.1415
bash$ _
```

- Similarly, we can define lists, vectors, maps (associative arrays), sets, and many other data structures, as templates. Such classes are called *container classes*. (Java uses the term *collection class* and groups all collection classes together into the *Collections* framework.)

- The C++ template allows a data type to be a parameter in the definition of a function or class (as shown previously). Classes can contain functions that use the templated data type. Templated functions can create local classes that use the data type. But templated classes cannot have templated functions that use templated data types other than the ones defined for the class.

- Templates are a compile-time mechanism and incur no run-time overhead compared to hand-written code.

Template Instantiation

- The process of generating a class declaration from a templated class is called *template instantiation*.

- Similarly, a function generated via templates is also "instantiated".

- In general, it is the implementation's job – not the programmer's – to ensure that the appropriate versions of a template function are generated for a particular set of template arguments. For example,

```
bash$ cat exam-ch3a.cpp
     1  void f()
     2  {
     3      Stack<string> sf;   // stack of strings
     4      Stack<int> xyzzy;

     5      sf.push("compiler's job to define this!");
     6  }
bash$ _
```

- The functions for creating and destroying stacks of *string*s and *int*s must be generated, and the *push* function for the stack of *string*s must also be generated, but no other functions need be defined by the template nor instantiated by the compiler.

- The generated classes and functions are ordinary; that is, they obey all the normal rules for classes and functions that ordinary classes and functions must obey (such as scoping rules, pointers to members, inheriting functions, nested classes, etc).

Template Parameters

- A template can take data type parameters, either one or more.
- Such parameters are used in the generation of the code created by instantiation of the template.
- For templated functions, the compiler will instantiate the function for each use of the function with different characteristics as determined by the template parameters, and function overloading will be used to resolve any ambiguity, as usual.
- Integer arguments for templates are handy for specifying size fields. For example,

```
bash$ cat exam-ch3b.cpp
     1  template <class T, int size>
     2  class Stack {
     3      T v[size];          // Assumes T has default constructor
     4      int top;
     5    public:
     6      class StackOverflow { };
     7      // ...

     8      Stack() : top(0) { }
     9      // Destructor not needed in this case since
    10      // memory is not being dynamically allocated.

    11      void push(T c) {    // Assumes T can be copied/assigned
    12          if (top == size) throw StackOverflow();
    13          v[top++] = c;
    14      }
    15      // ...
    16  };
bash$ _
```

Function Templates

- Soon after recognizing the need for template classes, the need for template functions becomes apparent. For example,

```
bash$ cat exam-ch3c.cpp
     1  // Template function declaration (defined elsewhere)
     2  template <class T>
     3  void sort(vector<T> &v);

     4  void f(vector<int> &vi, vector<string> &vs)
     5  {
     6      sort(vi);
     7      sort(vs);
     8  }
bash$ _
```

- When a template function is called, the types of the function arguments determine which version of the template is used; that is, the template arguments are deduced from the function arguments.

- Of course, the template function must actually be **defined** somewhere!

```
bash$ cat demo-sort1.cpp
     1  // Shell sort (Knuth, _Art of Computer Programming_)
     2  #include <vector>

     3  template <class T>
     4  void sort1(std::vector<T> &v)
     5  {
     6      const size_t n = v.size();

     7      for (int gap=n/2; 0 < gap; gap /= 2)
     8          for (int i=gap; i < n; i++)
     9              for (int j=i-gap; 0 <= j; j -= gap)
    10                  if (v[j+gap] < v[j]) {  // operator<
    11                      T temp = v[j];      // copy constructor
    12                      v[j] = v[j+gap];    // assignment oper
    13                      v[j+gap] = temp;    // assignment oper
    14                  }
    15  }
bash$ _
```

Function Templates (continued)

- Note some of the details of this templated version of the sort function.
  - — Cleaner and shorter (than an implementation using function pointers) since it can rely on information about the type being sorted.
  - — Doesn't rely on function pointers for comparisons, so is likely faster.
  - — Also implies no indirect function calls are required and that inlining of a simple **`operator<`** (inside the `if` statement) can be done easily.
- A further simplification is to use the standard library template **`swap`** function to reduce the if statement block:

```
if (v[j+gap] < v[j])              // Function: operator< (T, T)
    swap(v[j], v[j+gap]);
```

- This example requires that there be an **`operator<`** defined for the type being sorted, but this is easily avoided by defining templated helper classes, as shown next.

Function Templates (continued)

- Template functions are useful in general to provide algorithms without defining the data types which the algorithm supports.

- But some algorithms will require certain operations be available on any types that they will be manipulating, such as the **`operator<`** from the last example.

- However, not all types will provide such operators. And it is not logically correct for either the class to define them or for the algorithm to define them (for example, because sorting employees depends on the context).

- Take the sorting of strings, for examples. It is simple to imagine that strings may be sorted either case-sensitive or case-insensitive.

- The most efficient way to implement this in a template sorting function is to provide a second template parameter which identifies a class capable of performing the comparison:

```
bash$ cat demo-sort2.cpp
     1  // Shell sort (Knuth, _Art of Computer Programming_)
     2  #include <vector>

     3  template <class T, class C>
     4  void sort2(std::vector<T> &v)
     5  {
     6      const size_t n = v.size();

     7      for (int gap=n/2; 0 < gap; gap /= 2)
     8          for (int i=gap; i < n; i++)
     9              for (int j=i-gap; 0 <= j; j -= gap)
    10                  // This requires a static C::lt() function.
    11                  // Not too tough, since C is a template
    12                  // parameter.
    13                  if (C::lt(v[j+gap], v[j]))
    14                      swap(v[j], v[j+gap]);
    15  }
bash$ _
```

Function Templates (continued)

- Now the only step is to define a class capable of comparing two objects of type T:

```
bash$ cat demo-sort3.cpp
     1  #include "demo-sort2.cpp"
     2  #include <string>

     3  template <class T>
     4  class Cmp {
     5    public:
     6      static bool lt(T a, T b) { return a<b; }
     7      static bool eq(T a, T b) { return a==b; }
     8  };

     9  void f(std::vector<std::string> &vs)
    10  {
    11      // Explicit specification of template parameters
    12      sort2< std::string, Cmp<std::string> >(vs);
    13  }
bash$ _
```

- But how does this help?

- Suppose that a particular data type **does** support **`operator<`** — specifying the name of the class as shown above is all that's needed.

- If the objects being compared do not define **`operator<`**, then a new class can be created (such as **`Cmp`**, above) which defines the **`lt`** and **`eq`** functions to perform as required by the application. Then that class can be used in the call to **`sort`**.

Function Templates (continued)

- Templates may have default arguments, just as functions may. Default arguments allow the compiler to choose the parameters, based on whether or not they were specified by the programmer.

```
bash$ cat demo-sort4a.cpp
     1  // Note that the second parameter is defaulted to Cmp<T>,
     2  // which allows the programmer to simply call sort() for
     3  // the common case...

     4  #include "demo-sort3.cpp"

     5  // Doesn't work in GCC 4.x
     6  // error: default template arguments may not be used in
     7  //        function templates
     8  template <class T, class C = Cmp<T> >
     9  void sort4(std::vector<T> &v)
    10  {
    11      // ...
    12  }
bash$ _
```

- Or, if you prefer it formatted a little differently:

```
     1  #include "demo-sort3.cpp"

     2  template <class T,
     3            class C = Cmp<T> >
     4  void sort4(std::vector<T> &v)
     5  {
     6      // ...
     7  }
```

- Template functions also find use as members of classes, both template classes and ordinary classes.

- They follow all the same rules for definition and instantiation that normal template functions do, except that templated functions inside templated classes can only use the template parameters of the class when templating the function.

- There is yet one other way to accomplish the same thing using a technique known as *function objects*. We'll take a look at that solution right after we talk about *template specialization*.

User Specializations

- By default, a template gives a single definition to be used for every template argument (or combination) that a user can think of.
- This doesn't make sense if the template writer wants to say, "if the template argument is a pointer, use this implementation; if it is not, use this implementation" or "give an error unless the template argument is a pointer to a class derived from *My_Class*."
- Such design concerns can often be addressed by providing alternative definitions of the template and having the compiler choose between them based on the template arguments provided.
- Such alternative definitions are called *user specializations*.
- Consider the likely uses of a **`Vector`** template:

```
bash$ cat demo-vector1.cpp
     1  template <class T> class Vector {
     2      T* v;
     3      int sz;
     4    public:
     5      Vector();
     6      Vector(int s);
     7      T& elem(int i)              { return v[i]; }
     8      T& operator[](int i);       // includes bounds checking

     9      void swap(Vector &v);
    10      // ...
    11  };
    12  Vector<int> vi;
    13  Vector<Shape*> vps;
    14  Vector<string> vs;
    15  Vector<char*> vpc;
    16  Vector<Node*> vpn;
bash$ _
```

- Most **`Vector`**s will be vectors of pointers. There are several reasons, but the primary reason is to preserve run-time polymorphic behaviour (i.e., calling virtual functions is only done when a pointer to a base class is used).

User Specializations (continued)

- The default behaviour of most C++ implementations is to replicate the template code when used with different template arguments. This is good for run-time performance, but can cause serious code bloat.

- Fortunately, there is an easy solution. Since most **`Vector`** instantiations will be of a pointer type, we can provide a concrete class which manages **`void`** pointers, and use template classes as front-ends to this concrete class (writing the template class's functions as inlines guarantees very little code increase; see below).

```
bash$ cat demo-vector2.cpp
     1  // No arguments in the "template" statement and specifying
     2  // "void*" in the class declaration means that this
     3  // definition is to be used as the implementation of every
     4  // Vector for which T is a (void*).  This is called "full
     5  // specialization".

     6  template <>
     7  class Vector<void*> {
     8      void ** p;
     9      // ...
    10      void* & operator[](int i);
    11  };

    12  // Partial specialization: this template will be used for
    13  // all cases where the template parameter is a pointer but
    14  // IS NOT a void* (because that was taken care of above).

    15  template <class T>
    16  class Vector<T*> : private Vector<void*> {
    17      typedef Vector<void*> Base;
    18    public:
    19      Vector() : Base() { }
    20      explicit Vector(int i) : Base(i) { }

    21      T* &elem(int i) {
    22          return static_cast<T* &>(Base::elem(i));
    23      }
    24      T* &operator[](int i) {
    25          return static_cast<T* &>(Base::operator[](i));
    26      }
    27  };
bash$ _
```

User Specializations (continued)

- Note that the partial specialization uses a template parameter of **`T*`**, which is to say that any type expressable as a pointer will use that template definition, except for **`void*`** which has already been defined.

```
Vector<char*> vpc;      // T is "char"
Vector<string*> vps;    // T is "string"
Vector<int**> vppi;     // T is "int*"
```

- Since each method is inlined and is basically just a call to the base class definitions, the template causes no code bloat. So we have the compiler checking and enforcing data types, but no runtime overhead in the form of extra method calls or executable size!

- We could have given different names to the Vectors of objects and the Vectors of pointers, but experience shows that even seasoned programmers will forget and use the object Vectors, resulting in larger code than expected. This technique preserves the idea of a common interface with differing implementations.

- Order of specialization describes the idea that the compiler must choose between template classes based on how specific a particular use of a template is.

```
template <class T> class Vector;        // general
template <class T> class Vector<T*>;    // specialized for pointers
template <> class Vector<void*>;        // specifically for void*
```

- Naturally, specialization can also be valuable for template functions. Unfortunately, not all compilers implement specialization properly or completely. You can use the sample code on these past few pages to test your compiler. Even if your instructor doesn't assign this test as a lab step, it's very instructive to try it on your own.

Function Objects

- Previously, we showed how a templated comparison class with static functions for less-than and equal-to could be used as stand-ins for simple cases and more specific versions used when needed. *Function objects* are another possible solution based on a similar idea.

- Function objects are instantiations of a class which defines an **`operator()`** (and possibly some data members if the overloaded operator requires them).

- This allows the object to be passed to a function and the function can use the object name as a function call and pass it parameters.

- The previous example using a separate class with **`lt()`** and **`eq()`** functions would instead become the code shown below. Notice the comparison in **`sort()`** and the definition of **`Cmp`** and its associated operator.

```
bash$ cat demo-sort5b.cpp
     1  #include "demo-sort5a.cpp"

     2  template <class T>
     3  void sort5(vector<T> &v, Cmp<T> &fobj)
     4  {
     5      const size_t n = v.size();

     6      for (int gap=n/2; 0 < gap; gap /= 2)
     7          for (int i=gap; i < n; i++)
     8              for (int j=i-gap; 0 <= j; j -= gap)
     9                  if (fobj(v[j+gap], v[j]))
    10                      swap(v[j], v[j+gap]);
    11  }
bash$ _
```

Function Objects (continued)

```
bash$ cat demo-sort5a.cpp
     1  #include "demo-sort4a.cpp"

     2  // General template
     3  template <class T>
     4  class Cmp {
     5    public:
     6      bool operator()(T &a, T &b) { return a<b; }
     7  };

     8  // Full specialization which is only for (Empl*)'s
     9  template <>
    10  class Cmp<Empl*> {
    11    public:
    12      bool operator()(const Empl *a, const Empl *b) {
    13          return a->GetName() < b->GetName();
    14      }
    15  };

    16  void f(vector<Empl*> &vs)
    17  {
    18      Cmp<Empl*> fobj;
    19      sort4(vs, fobj);
    20  }
bash$ _
```

- One of the motivations for using function objects, or *functors*, as they are sometimes called, is that because they are objects they have all the attributes of an object. Likely the most important ones are that they can contain their own state, and the **`operator()`** can be virtual.

Source Code Organization

- Template use involves the same constraints seen for normal classes and functions:
  - They must be *declared* where they can be seen by all code which uses them, and
  - They must be *defined* where the linker will find them.
- This implies that template declarations should appear in a header file and be **`#include`**'d where needed, then the definition provided in a separate source file which is compiled separately, but linked together with the application code.
- This is not the way the language is defined to work, however:
  - Note that to be accessible from other compilation units, however, a template definition must be explicitly declared **`export`** (see section 9.2.3 of [BS3rd], pg 203).
  - This can be done by adding **`export`** to the definition or to a preceding declaration. Otherwise, the definition must be in scope wherever the template is used. (In other words, it works similar to inlined functions.)
  - This is another area where many compilers are not completely supporting the C++ language standard. Fortunately, most students will not be writing their own template classes but using the template classes already written by someone else, perhaps as part of the standard C++ library or perhaps in a product purchased from a vendor. Either way, they've already dealt with this issue – just follow their documentation on the use of the library and everything will work.

Contents of the STL

- The Standard Template Library contains the following declarations of classes and functions. The STL is covered in more detail in a separate chapter.

- Container classes (for a full discussion, see *STL Tutorial and Reference Guide* [Musser and Saini, Addison-Wesley Publishing], chapters 6 and 7)

Container Type	Declaration	Attributes and/or Description
Sequence	`vector<T>`	random access; constant time modifications at the beginning and end
	`deque<T>`	random access; variable length; constant time modifications at the beginning and end
	`list<T>`	linear time access to a sequence of varying length, but with constant time modifications at *any* position in the sequence
Sorted, Associative	`set<Key>`	unique keys; fast retrieval of keys
	`multiset<Key>`	duplicate keys; fast retrieval of keys
	`map<Key, T>`	unique keys; fast retrieval of another type T based on keys
	`multimap<Key, T>`	duplicate keys; fast retrieval of another type T based on keys

Contents of the STL (continued)

- Sample of generic algorithms (q.v. *STL Tutorial and Reference Guide*)

Algorithm	Description
`find`	locate first occurrence of a value within a range
`for_each`	applies given function to each element
`adjacent_find`	returns an iterator to the first consecutive duplicate element
`count`	counts the number of matching elements
`mismatch`	compares corresponding pairs of elements from two iterators and returns the first mismatched pair
`equal`	returns `true` if two ranges are equal for a given length
`search`	returns an iterator that points to the first subsequence that matches the search iterator
`merge`	merge values from two (sorted) sequences together, generating a third sequence

- Sample iterators (q.v. *STL Tutorial and Reference Guide*)

Iterator Type	Description
`Input`	Read-only; unidirectional
`Output`	Write-only; unidirectional
`Forward`	Read/write; unidirectional; may be multi-pass
`Bidirectional`	Read/write; bidirectional
`Random Access`	Read/write; bidirectional; "big jumps"; iterator subtraction; iterator comparisons

Review

- Why templates?
- Template instantiation
- Template parameters
- Function templates
- User specializations
- Source code organization
- Overview of the STL

Notes:

Advanced C++ Programming

Lab for Templates

Lab for Templates

1. Goals for this lab:

   A. Define a template class that can be used to contain integers

   B. Extend the integer container so it works with other data types

   C. Define a template stack class whose parameter is a container; thus, the stack class provides an interface independent of the implementation.

2. In this lab you will be developing a class to maintain a stack of integers. It will be written first without the use of templates. Then you'll parameterize the *stack* class to allow stacks of other data types (namely, employees). And last, the stack class will be separated into implementation and interface, so that the stack class can specify a *list*, *vector*, or *deque* as an implementation (as a template parameter).

3. The first step is to write the *stack* class. Here are the requirements for this class:

   A. The constructor takes a size parameter (integer) to specify the maximum size of the stack.

   B. The functions *int push(int value)*, *int pop()*, and *void clear()* are the only public member functions.

      i) *push* takes a value to put onto the stack and returns the new size of the stack. Zero is returned if the stack is full and the new value could not be added.

      ii) *pop* returns the value on the top of the stack and removes it. If the stack is empty, it returns -1 (a real class would throw an exception — we'll be doing that later).

      iii) *clear* simply pops all the elements off the stack and empties it. (This can be implemented much more quickly inside the class than having the application use *pop* in a loop.)

   C. Test the class by writing a *main* program which creates an object of class *stack* and pushes 10 integer values onto it. Try various sizes of stacks (from 8 to 12, for instance), in order to test the code which executes the *push()* and *pop()* methods (be sure you check the return value). Verify that the

Lab for Templates (continued)

destructor, if any, does whatever is required. I would suggest some **`cout`** statements in the constructor/destructor...

D. Here is a possible declaration for the *stack* class, although I don't guarantee that it's complete:

```
class stack {
    int *v;             // v = new int[max_size];
    int max_size;       // size set from constructor
    // ...
  public:
    stack(int size);
    int push(int value);        // add `value' to stack
    int pop();                  // retrieve value from stack
    void clear();               // clear all elements
    // ...
};
```

4. When the stack of integers works, modify the class into a template class which can hold a stack of anything. Try changing the program to create a stack of employees (you may need to define both a default constructor and a copy constructor for the *Empl* class — why?). What are you going to do about the data type of the return value from the *pop* function? Should it be a reference or a copy of the object? Be prepared to discuss the options during the lab review. (It would be especially instructional if you were to try using both.)

Lab for Templates (continued)

```
// ...

int main()
{
    // ...
    stack<int> stack_of_int(10);

    // Put 10 elements on the stack
    for (int i=0; i < 10; i++)
        stack_of_int.push(i+100);

    for (int i=1; i < 11; i++)
        cout << "Element #" << i << " is "
             << stack_of_int.pop() << endl;

    // ...
    return 0;
}
```

5. If you get here, you've done good so far!

   Try using your stack template to create a stack of employee pointers. Before you actually write the code, though, go through the stack class and determine if the individual member functions will be handling the pointers correctly.

   Then modify your *main* program to use **`new`** to create 10 employee objects which are added to the stack object. (Modify your employee constructors/destructors to have a **`cout`** statement in them, since I want you to see what happens.)

   When you run the program, are the employee objects ever **`delete`**'d? Why or why not? What would you change to make it work? Would your changes still work if the stack class were changed back to containing the actual employee object instead of a pointer? This is an example of where *partial specialization* would be used; pointer containers would have **`delete`** in their destructors, and object containers wouldn't.

6. **Extra Credit:**

   This is only if you still have some time left!

   Change the stack class so that the *std::list* class can be passed as a template

Lab for Templates (continued)

parameter. This would allow the stack to be implemented in a number of ways, always presenting the same interface to the application. You will need to know that the *std::list* class defines a data type, *std::list<T>::value_type*, which the stack class will use to determine the parameter type and return type for *push* and *pop*.

```
// ...

int main()
{
    stack< float, std::list > stack_of_floats(20);
    stack< int, std::vector > stack_of_integers(20);

    stack_of_floats.push( 1.0 );
    stack_of_floats.push( 2.0 );
    stack_of_floats.push( 3.0 );

    stack_of_integers.push( 1 );
    stack_of_integers.push( 2 );
    stack_of_integers.push( 3 );

    return 0;
}
```

Why would you want to do this in a real application? (You may want to refer back to the STL chart in the lecture notes.)

TAB HERE

Advanced C++ Programming

Stream I/O

Objectives

- Why call it "Stream I/O" instead of "File I/O"?
- The organization of the I/O classes
  - Formatting
  - Buffering
  - Application interface
- Adding overloaded insertion and extraction operators
- Using **`ifstream`** and **`ofstream`**
- Using **`istringstream`** and **`ostringstream`**

Stream I/O vs. File I/O

- Input-output mechanisms for programing languages are notoriously difficult to build.
  - — The ideal system would provide easy and intuitive access to the built-in data types of the language and still allow the programmer to extend those facilities for use with user-defined data types.
  - — They must be relatively inexpensive to use, or programmers will spend much time and energy attempting to subvert the standard system and invent their own.
  - — They must be logically structured and understandable, or again the programmers will not use them.
  - — They must not introduce large amounts of "code bloat" or system implementors will shy away from them as being to memory consumptive.
- All of the above aspects make it difficult to design an I/O system which will be a pleasure to use, easy to enhance and extend, and small and efficient in terms of its memory footprint.
- The I/O library for C++ comes close to achieving those goals. But only experience will tell whether the promise has been kept.

Stream I/O vs. File I/O (continued)

- One of the first things that a programmer new to C++ will notice about the I/O system is its consistency.

- All items to be input or output are treated similarly. This includes built-in data types such as integers and floats, as well as user-defined data types.

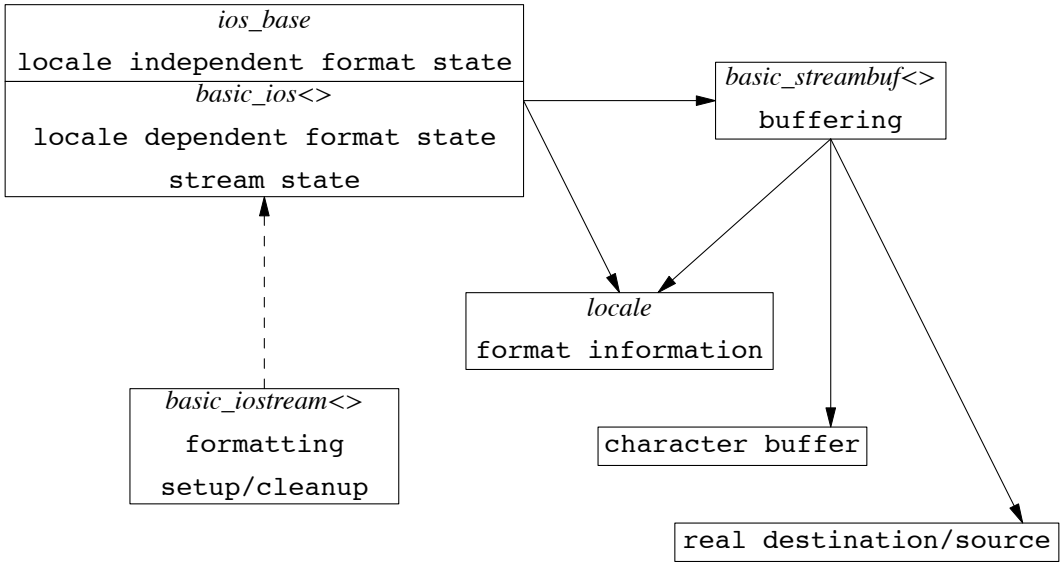
```
bash$ cat exam-ch4a.cpp
     1  #include <iostream>
     2  using namespace std;

     3  #include "demo-review1.h"

     4  // ...

     5  int main()
     6  {
     7      Empl fred, barney;
     8      int i, j;
     9      Date now, then;
    10      // ...
    11      cout << fred   << i << now  << endl;
    12      cout << barney << j << then << endl;
    13      return 0;
    14  }
bash$ _
```

- But the I/O system doesn't consist only of *cin* and *cout*.

- In fact, because of the modular design of the class hierarchy, the file buffering components can be replaced with string buffers, which provides the programmer with the opportunity to perform in-memory formatting or numeric interpretation.

The Design of the I/O System

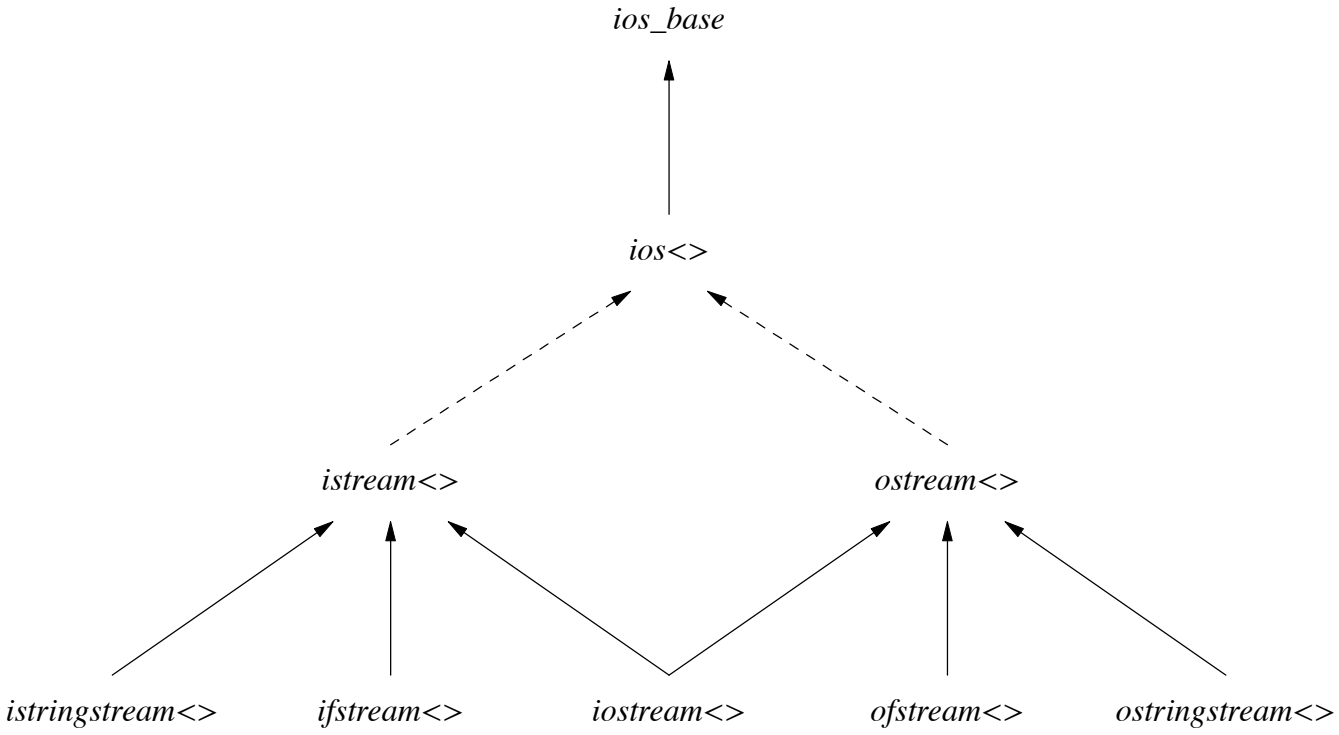
- The C++ I/O system consists of various classes which each implement a smaller part of the entire picture.



- The dashed arrow indicates virtual inheritance; the solid arrows represent pointers.
- The classes marked with <> are templates parameterized by a character type and containing a locale. Typical character types are **`char`** and **`wchar_t`**.

The Design of the I/O System (continued)

- Furthermore, there are other related classes which will be discussed in this chapter, as shown below. Names suffixed by **<>** are templates parameterized on the character type and their names have a *basic_* prefix. A dashed line indicates a virtual base class.




```
typedef basic_ios<char>                 ios;
typedef basic_streambuf<char>           streambuf;
typedef basic_istream<char>             istream;
typedef basic_ostream<char>             ostream;
typedef basic_iostream<char>            iostream;
typedef basic_stringbuf<char>           stringbuf;
typedef basic_istringstream<char>       istringstream;
typedef basic_ostringstream<char>       ostringstream;
typedef basic_stringstream<char>        stringstream;
typedef basic_filebuf<char>             filebuf;
typedef basic_ifstream<char>            ifstream;
typedef basic_ofstream<char>            ofstream;
typedef basic_fstream<char>             fstream;
```

Formatting

- Formatting operations are a large part of any I/O system. The C++ I/O library is no different.
- The formatting provided by the *ios_base* class is quite robust in terms of supported facilities for built-in data types.
- Most conversions from a binary representation to a representation appropriate for human consumption (generally ASCII text) is handled by the **`operator<<`** and **`operator>>`** functions.
- These routines convert the internal data type to an external form and vice versa.
- These conversions are controlled by parameters which specify field width, number of digits after the decimal point, output radix, and others.
- The following conversion functions are defined:

```
class ostream : virtual public ios {
    // ...
    ostream &operator<< (ostream &, char);
    ostream &operator<< (ostream &, signed char);
    ostream &operator<< (ostream &, unsigned char);
    ostream &operator<< (ostream &, short);
    ostream &operator<< (ostream &, int);
    ostream &operator<< (ostream &, long);
    ostream &operator<< (ostream &, long long); // optional
    ostream &operator<< (ostream &, float);
    ostream &operator<< (ostream &, double);
    ostream &operator<< (ostream &, long double);
    // ...
};
```

Formatting (continued)

- Specifying field width, fill characters, decimal digits, and the like, are handled through *manipulators* defined in **iostream** and **iomanip**.

- Here are those manipulators and their function:

Manipulator	Persistent	Description
`skipws`	Yes	Skips whitespace on input (default)
`noskipws`	Yes	Whitespace is significant on input
`dec`	Yes	Output integers in decimal (default)
`hex`	Yes	Output integers in hexadecimal
`oct`	Yes	Output integers in octal
`setbase(n)`	Yes	Output integers in base *n* (supported values vary)
`setprecision(n)`	Yes	Change number of digits after decimal point to *n* (for floating point numbers; requires **`fixed`** as well)
`fixed`	Yes	Change floating point output to fixed notation
`scientific`	Yes	Change floating point output to scientific notation
`setw(n)`	No	Change field width to *n*
`setfill(ch)`	No	Change the fill character for short field values to *ch*

Buffering

- The buffering classes of the I/O package provide for efficient I/O even in the face of the application performing single character operations.
- This is accomplished by reading (writing) full buffers, the size of which is implementation dependent, and returning (sending) the requested number of characters.
- By providing buffering as a separate function of the I/O system, various types of buffers can be created and used easily in an application.
- A particularly common form of buffering is to use strings (the concept, not the class!).
- For example, an output stream can be created which is not connected to an external file but to an internal memory buffer, allowing the application to "pre-format" information in memory before using it.
- The classes involved in this are **`istringstream`**, **`ostringstream`**, **`stringstream`**, **`strstreambuf`**, and **`streambuf`**.
- **`streambuf`** is a base class for all stream buffer classes which might be used with the I/O library, including file buffers (**`filebuf`**) and string buffers (**`strstreambuf`**).

Buffering (continued)

```
bash$ cat demo-iobuff.cpp
     1  #include <iostream>
     2  #include <iomanip>
     3  #include <sstream>      // Used to be <strstream>
     4  using namespace std;

     5  int main()
     6  {
     7      int i = 42;
     8      char *characters = "This is a test.";
     9      float f = 3.1415926535;

    10      ostringstream tmp;
    11      tmp << fixed << setprecision(2);
    12      tmp << "Integer: "  << setw(6) << setfill('*') << i;
    13      tmp << ", String: " << characters;
    14      tmp << ", Float: " << f;            // dollars and sense

    15      string copy = tmp.str();
    16      cout << copy << endl;
    17      return 0;
    18  }
bash$ _
```

```
bash$ demo-iobuff
Integer: ****42, String: This is a test., Float: 3.14
bash$ _
```

Application Interface

- The application's interface to all this is typically through the insertion and extraction operators.
- But there are times when the application may want to directly send or receive binary data on a stream, or process information on some physical boundary which occurs in the data, such as newline.
- The stream objects provide such an interface.
- The table below summarizes the methods provided and their functionality.

Method	**Description**
`write(const char *cp, streamsize n)` `read(char *cp, streamsize n)`	Writes *n* bytes from *cp* Reads *n* bytes into *cp*
`get(char *cp, int len, char delim='\n')`	Reads up to *len* characters or *delim*, whichever comes first. **Does not** remove the delimiter from the input stream.
`getline(char *cp, int len, char delim='\n')`	Reads up to *len* characters or *delim*, whichever comes first. **Does** remove the delimiter from the input stream.
`gets(char **s, char delim='\n')`	Reads up to *delim*, allocating memory for the pointer *s* as necessary. The user must **`delete`** *s* when no longer used.

Application Interface (continued)

- One of the issues in using the stream library is that character strings cannot be read back in the same way they are written out.

- When writing a string, any embedded spaces are printed as is. But *whitespace* characters are delimiters when reading a string back in.

- If the programmer is already using the string class, deriving another class which adds quotes around a string during output and uses them as delimiters during input would solve this problem.

Adding Overloaded I/O Operators

- This section will use the previous problem of writing strings and reading them back in as an example in coding I/O operators.

```
bash$ cat demo-stringm.cpp
     1  #include "demo-string.h"

     2  int main()
     3  {
     4      dstring a = "This is a test";

     5      cout << a << endl;          // will have quotes

     6      if (isatty(fileno(stdin)))
     7          cout << "Enter a quoted string:" << endl;
     8      cin >> a;
     9      cout << "You entered this string" << endl;
    10      cout << "===|" << a << "|===" << endl;
    11  #ifdef WIN32
    12      char ch;
    13      cin.get(ch);
    14  #endif
    15      return 0;
    16  }
bash$ _
```

Adding Overloaded I/O Operators (continued)

```
bash$ cat testdata0
Just a regular test, no quotes or anything.
bash$ demo-stringm < testdata0
"This is a test"
You entered this string
===|Just|===
bash$ _
```

```
bash$ cat testdata1
"This isn't a test.  (Note the single quote.)"
bash$ demo-stringm < testdata1
"This is a test"
You entered this string
===|"This isn't a test.  (Note the single quote.)"|===
bash$ _
```

```
bash$ cat testdata2
'This "is" a test.  (Note the double quotes.)'
bash$ demo-stringm < testdata2
"This is a test"
You entered this string
===|'This "is" a test.  (Note the double quotes.)'|===
bash$ _
```

Adding Overloaded I/O Operators (continued)

```
bash$ cat demo-string.h
     1  #ifndef dstring_h_
     2  #define dstring_h_

     3  #include <iostream>
     4  #include <string>

     5  using namespace std;    // don't need "std::string"...

     6  class dstring : public string {
     7    public:
     8      dstring() : string() { }
     9      dstring(string &cp) : string(cp) { }
    10      dstring(const char *cp) : string(cp) { }
    11      // destructor provided by compiler is just fine
    12      // but I really should define some non-inherited
    13      // operators, such as operator=().
    14  };

    15  // Not necessary that these be friends of the class
    16  // since they only access public methods.
    17  ostream &operator<< (ostream &os, const dstring &d);
    18  istream &operator>> (istream &os,       dstring &d);

    19  #endif // dstring_h_
bash$ _
```

Adding Overloaded I/O Operators (continued)

```
bash$ cat demo-string-o.cpp
     1  #include "demo-string.h"

     2  ostream &operator<< (ostream &os, const dstring &d)
     3  {
     4      // No need for a delimited string if it doesn't contain
     5      // any whitespace characters.  Unfortunately, the only
     6      // way to *really* check for whitespace is to build an
     7      // array populated via the isspace() function.  So I'm
     8      // taking a shortcut here.

     9      if (d.find_first_of(" '\"\t\n\f") == string::npos) {
    10          const string &s = d;    // no ws, so output as-is
    11          os << s;
    12      } else {
    13          char delim = '"';       // default delimiter...
    14          if (d.find('"') != string::npos)
    15              delim = '\'';       // change to single quote

    16          os << delim;
    17          string::size_type len, pos = 0;
    18          while ((len = d.find(delim, pos)) != string::npos) {
    19              os << d.substr(pos, pos+len-1);
    20              os << '\\';
    21              os << delim;
    22              pos += len+1;
    23          }
    24          os << d.substr(pos);
    25          os << delim;
    26      }
    27      return os;
    28  }
bash$ _
```

Adding Overloaded I/O Operators (continued)

```
bash$ cat demo-string-i.cpp
     1  #include "demo-string.h"

     2  istream &operator>> (istream &is, dstring &d)
     3  {
     4      // If first character is a delimiter, use it.
     5      // Otherwise, just read a "string".
     6      char delim;
     7      is >> delim;                // skips ws before char

     8      if (is) {                   // If no error on "delim"
     9          if (delim == '"' || delim == '\'') {
    10              // Probably more efficient to use get(),
    11              // but this is just a simple example, not
    12              // a robust one.
    13              char ch;
    14              d.clear();  // Start with empty string
    15              while (is.get(ch)) {
    16                  if (ch == delim)
    17                      break;
    18                  if (ch == '\\')
    19                      is.get(ch);
    20                  d += ch;
    21              }
    22          } else {
    23              d.clear();  // Start with empty string
    24              string &s = d;
    25              is.putback(delim);
    26              is >> s;    // overwrite string
    27          }
    28      }
    29      return is;
    30  }
bash$ _
```

Using File I/O Objects

- The next few pieces of example code show a class which selects a file from a directory, then prints the contents of that file to stdout.
- It demonstrates both file I/O and how to portably perform directory access.

```
bash$ cat signatures1.h
     1  #ifndef signatures_h_
     2  #define signatures_h_

     3  #include <sys/stat.h>

     4  #include <string>
     5  #include <list>

     6  using namespace std;

     7  class Signatures : public list<string> {
     8      iterator current;
     9      time_t last_mtime;
    10      struct stat sbuf;
    11      string signatureDirectory;

    12      void ReadDirectory();
    13      int statDirectory();
    14    public:
    15      Signatures();
    16      void Empty();
    17      void ReFill();
    18      string GetOneFilename();
    19  };

    20  #ifndef MAXPATHLEN
    21  # define MAXPATHLEN              (MAXPATH)
    22  #endif

    23  #endif // signatures_h_
bash$ _
```

Using File I/O Objects (continued)

```
bash$ cat signatures1.cpp
     1  #include <iostream>

     2  #include <dirent.h>     // For DIR and opendir() family
     3  #include <sys/param.h>  // For MAXPATHLEN
     4  #include <stdio.h>      // For strerror()
     5  #include <sys/errno.h>  // For errno

     6  #include "signatures1.h"

     7  // Fill the base class' list with directory entries
     8  // from the signatureDirectory.  In order to speed up
     9  // the loop, we change directory to the signatureDirectory
    10  // and change back when complete.  (Note that any
    11  // 'finally' block would also have to change back.)
    12  void Signatures::ReadDirectory()
    13  {
    14      struct dirent *de;          // From <dirent.h>
    15      DIR *dp;                    // From <dirent.h>
    16      char olddir[MAXPATHLEN+10];

    17      dp = opendir(signatureDirectory.data());
    18      if (dp == 0) {
    19          cerr << "Can't open '"
    20               << signatureDirectory << "'" << endl;
    21          exit(1);
    22      }
    23      getcwd(olddir, sizeof(olddir));
    24      chdir(signatureDirectory.data());
    25      while ((de = readdir(dp)) != 0) {
    26          if (de->d_name[0] != '.') {
    27              if (stat(de->d_name, &sbuf) != 0) {
    28                  cerr << "Can't stat() entry '"
    29                       << de->d_name << "'" << endl;
    30                  exit(2);
    31              }
    32              if ( !S_ISDIR(sbuf.st_mode) )
    33                  push_back( string(de->d_name) );
    34          }
    35      }
    36      chdir(olddir);
    37      closedir(dp);
    38      sort();             // inherited
    39  }
```

Using File I/O Objects (continued)

```
40  Signatures::Signatures()
41      : last_mtime(0)
42  {
43      const string DEFAULT_DIR = "/usr/local/etc/sigs";
44      signatureDirectory = getenv("HOME");
45      if (signatureDirectory.length() != 0 &&
46              statDirectory() == 0) {
47          // HOME variables exists and so does directory
48          signatureDirectory.append("/.sigs");
49          if (statDirectory() != 0) {
50              // That directory doesn't exist.  Try another.
51              signatureDirectory = DEFAULT_DIR;
52          }
53      } else {
54          // We don't have a HOME directory?!
55          signatureDirectory = DEFAULT_DIR;
56      }
57      if (statDirectory() != 0) {
58          cerr << "Directory not accessible: "
59               << strerror(errno) << endl;
60      }
61      ReadDirectory();
62  }

63  string Signatures::GetOneFilename()
64  {
65      if (statDirectory() != 0) {
66          cerr << "Directory '" << signatureDirectory
67               << "' no longer accessible?!  "
68               << strerror(errno) << endl;
69          exit(3);
70      }
71      if (sbuf.st_mtime > last_mtime) {
72          Empty();
73          last_mtime = sbuf.st_mtime;
74      }
75      if (size() == 0) {          // Where is size()?
76          ReadDirectory();
77          if (size() == 0) {
78              cerr << "No filenames available." << endl;
79              exit(4);
80          }
81      }
```

Using File I/O Objects (continued)

```
    82      // Where does front() come from?  And why use two calls
    83      // when just pop_front() could be used??
    84      string copy = signatureDirectory + '//' + front();
    85      pop_front();
    86      return copy;
    87  }

    88  void Signatures::ReFill()
    89  {
    90      Empty();
    91      ReadDirectory();
    92  }

    93  void Signatures::Empty()
    94  {
    95      erase(begin(), end());      // Where is erase()?
    96  }

    97  int Signatures::statDirectory()
    98  {
    99      return stat(signatureDirectory.data(), &sbuf);
   100  }
bash$ _
```

Using File I/O Objects (continued)

```
bash$ cat fileiom.cpp
     1  #include <iostream>
     2  #include <fstream>
     3  #include <string>

     4  using namespace std;

     5  #include "signatures1.h"

     6  int main()
     7  {
     8      Signatures filelist;

     9      // Randomly print out each filename to prove it works...
    10      for (int i=filelist.size(); i-- > 0; )
    11          cout << filelist.GetOneFilename() << endl;

    12      // Now pick one file and print out its contents.
    13      filelist.ReFill();
    14      string filename = filelist.GetOneFilename();
    15      ifstream input(filename.data(), ios::in|ios::binary);
    16      cout << input.rdbuf();

    17  #ifdef WIN32
    18      char ch;
    19      cin.get(ch);
    20  #endif
    21      return 0;
    22  }
bash$ _
```

```
bash$ fileiom
/usr/local/etc/sigs/four
/usr/local/etc/sigs/one
/usr/local/etc/sigs/three
/usr/local/etc/sigs/two
This is file 'four'.
bash$ _
```

Using the String Streams

- File I/O objects are convenient and suffice for the most common cases, but there are times that in-memory formatting is required.
- For example, suppose that a data file contains multiple lines of information. The information changes based on the number of fields in the input record.
- When the input record contains information that doesn't allow full processing until the entire record has been read, the best technique is probably to read the entire record into a buffer and then decide how to process it based on its contents.
- After determining the correct input record format, the string would be treated as an input stream and the appropriate data types read from it.
- The following example demonstrates this technique.

```
bash$ cat demo-strstr.inc
This is,the value of PI,3.14159265358979323844
Only four,strings on,this line,.
And the sin(),of 45 degrees is,0.70710678118654752439
Testing error handling...
bash$ _
```

```
bash$ demo-strstr
This is|the value of PI|3.14159
Only four|strings on|this line|.
And the sin()|of 45 degrees is|0.707107
=== Error: Testing error handling... ===
bash$ _
```

Using the String Streams (continued)

```
bash$ cat demo-strstr.cpp
     1  #include <iostream>
     2  #include <fstream>
     3  #include <sstream>

     4  using namespace std;

     5  const int BUFSIZE = 512;

     6  int main(int argc, char **argv)
     7  {
     8      // Open and read an input file line-by-line.
     9      // Each line contains either 3 or 4 fields.
    10      // Those with 3 fields contain two strings and a float.
    11      // The lines with 4 fields contain all strings.
    12      const char *InputName = "demo-strstr.inc";
    13      if (argc == 2)
    14          InputName = argv[1];
    15      ifstream input(InputName);

    16      if (!input) {
    17          cerr << "Can't open file " << InputName << endl;
    18          return 0;
    19      }
    20      while (input) {
    21          char buff[BUFSIZE];

    22          input.getline(buff, sizeof(buff));
    23          if (input.eof())
    24              return 0;
    25          if (!input) {
    26              cerr << "Error reading: " << InputName << endl;
    27              return 0;
    28          }
    29          // Count the number of commas.
    30          int total = 0;
    31          char *p = buff-1;       // 'cuz of +1 in while loop
    32          while ((p = strchr(p+1, ',')) != 0)
    33              total++;

    34          istringstream tmp(buff);
    35          if (total == 2) {
    36              // Two commas means three fields...
    37              char s1[BUFSIZE], s2[BUFSIZE];
```

Using the String Streams (continued)

```
    38              float flt = -1.0;

    39              tmp.getline(s1, sizeof(s1), ',');
    40              tmp.getline(s2, sizeof(s2), ',');
    41              tmp >> flt;
    42              cout << s1 << '|' << s2 << '|' << flt << endl;
    43          } else if (total == 3) {
    44              // Three commas means four fields...
    45              char s1[BUFSIZE], s2[BUFSIZE];
    46              char s3[BUFSIZE], s4[BUFSIZE];

    47              tmp.getline(s1, sizeof(s1), ',');
    48              tmp.getline(s2, sizeof(s2), ',');
    49              tmp.getline(s3, sizeof(s3), ',');
    50              tmp.getline(s4, sizeof(s4), ',');
    51              cout << s1 << '|' << s2 << '|'
    52                   << s3 << '|' << s4 << endl;
    53          } else {
    54              // Should be 'cerr'...
    55              cout << "=== Error: "
    56                   << buff << " ===" << endl;
    57          }
    58      }
    59      return 0;
    60  }
bash$ _
```

Review

- The I/O class organization
- Adding insertion and extraction operators
- Using **`fstream`** classes
- Using **`stringstream`** classes

Notes:

Advanced C++ Programming

Lab for Streams I/O

Lab for Streams I/O

1. Goals for this lab:

   A. Experiment with file I/O and buffer streams

   B. Use the string stream classes for in-memory formatting

2. In this lab, you're going to overload the << operator for your employee class so that you can output employees directly. This will give you some experience with creating I/O operators. You'll then derive a new class, *Mgr*, from *Empl* and create a stack of employee pointers, some of which are actually managers (at least 2 or 3). Then overload an insertion operator for the stack class so that all of the employees are output (correctly) as either employees or managers.

3. Your functional requirements are:

   A. The employee class should be able to be output just like other data types (such as `int` and `float`). Consider what would be necessary to read the employee back in, but don't attempt it unless you have time left at the end of this lab.

   B. We want this output function to work as though it was virtual, so make modifications to it as necessary. (Create a public virtual member function which prints the object. The compiler will call the correct function at run time, since it's virtual. Then call this function in your overloaded insertion operator for the base class.)

   C. Now that you have this virtual function, you'll need to define that function for the manager class. Then your insertion operator (which prints the employee object) should print the managers correctly also. (Hint: what causes the compiler to use a virtual function instead of just generating the mangled function name?) (Double hint: pointers)

      Define the manager to have one additional field, a string which contains the department that the manager supervises. Make sure that the department is printed when the employee insertion operator is called.

Lab for Streams I/O (continued)

```
bash$ cat exam-lab05a.cpp
#include "empl.h"
#include "mgr.h"

// Print an employee by calling the virtual function
// and passing it the ostream reference...  No other
// overloaded operators are required.
//
// (This next line is missing one character.  Where?)
ostream& operator<< (ostream& os, const Empl e);

int main()
{
    Empl fred("Fred", "Flintstone");
    Mgr  wilma("Wilma", "Flintstone");

    wilma.SetDept("Finance");
    cout << fred << endl;       // calls insertion operator
    cout << wilma << endl;
    // What would be needed to make the following line work?
    // cin >> fred;
    return 0;
}
bash$ _
```

D. It would be useful if the stack class that we built previously also had an overloaded output operator which printed each element in the stack (without removing them). Add an overloaded insertion operator to do this. (Hint: you'll need a template function for this.)

4. **Extra Credit:**

A. How would you implement a merge sort? Consider the building of a *MergeSort* class. If the class could be given a list of filenames and then told to perform the merge, the class could be used over and over again in various applications. What would be the general design of the class? Which classes would it inherit from? use? be composed of?

Sketch out a rough idea about how you would implement such a class, after you've answered the previous questions about the design. As you start thinking about how you might actually perform the work, modify your answers to the above questions to accommodate. For instance, would your class be

Lab for Streams I/O (continued)

given individual filenames to merge? Or would you give it a list of filenames all at once? Or perhaps a wildcard that it would interpret to generate its own list of filenames? Or maybe the calling function is supposed to open the files and pass **`ifstream`** pointers to the new class?

This is an example of how a design might say, "do it this way", but when you actually start writing code, you realize that other techniques may be more useful or more efficient (in space or time). Always build as complete a design as possible, and always be willing to go back and modify it!

B. Performing I/O in ASCII is easier for humans to read, and hence it's easier to debug the code which generates it! But it is much quicker to write the information out in binary instead of converting it to characters on output and back to binary on input.

Consider: derive a new class called **`obfstream`** which does all of it's output in binary instead of ASCII. What class(es) should it be derived from? Would looking at **`ofstream`** help you decide? Consider overloading the insertion operator for each of the built-in data types; what about lists, vectors, and so on?.

Then overload the insertion operator for this class so that employees are written in binary also. (Actually, the employee only contains *float*s and *string*s, so those are the only "built-in" types you need to support in your new class.)

C. Problem to ponder: how would you implement a system that allows objects to be written to a stream in binary, then read back in without the application knowing what the next item in the stream was?

For example, an application writes out 3 employees and a manager. How would you design the **`obfstream`** class (and any supporting classes) so that a different application could read in the objects without knowing their types in advance (assume that this other application is linked with the employee and manager classes).

TAB HERE

Advanced C++ Programming

Inheritance

Objectives

- What does inheritance attempt to accomplish?
- An Example of Composition
- Upcasting vs. Downcasting
- The Type-field Solution
- Virtual Functions
- Abstract Base Classes
- Multiple Inheritance

What is (and Why Use) Inheritance?

- Try to explain to a foreigner what a car is; soon you'll be explaining engines, gasoline, service stations, oil, wheels and tires, seats, doors, headlights, and so on.

- Or try to explain how circles relate to triangles or rectangles, or how spheres relate to cubes or cylinders.

- In such cases as these, there are three distinct relationships being modeled:

  Inheritance
  which describes the *is-a-specialization-of* or *is-a-kind-of* relationship.

  Composition
  which describes the *is-composed-of* or *has-a* relationship.

  Uses which describes the *uses-a* relationship.

- Composition is somewhat intuitive; consider the example of the car containing an engine, four wheels, four doors, two headlights, and so on.

- A class could be created which *encapsulates* the concept of a car by being composed of many other objects, each of which may have intricacies of its own, but which the car needn't be bothered with.

- Composition is not inheritance, but in some cases may be used in similar ways, as we'll see later.

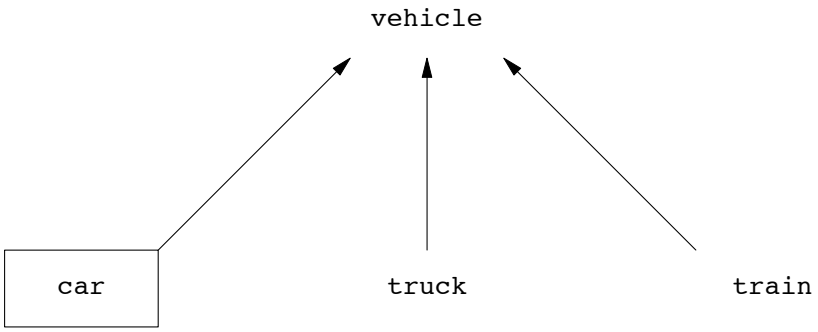
- Similarly, the *uses-a* relationship is not inheritance — it simply indicates that a particular object uses the interface of another object. This condition is used during the design phase to determine if perhaps there are too many relationships to or from a particular class.

- If there are many relationships with a class, the ability to reuse the class in other applications is restricted. For instance, the gas pump at a service station is *used by* the car. But any car can use the gas pump, and any gas pump can service a particular car.

What is (and Why Use) Inheritance? (continued)

- Inheritance is used when classes representing a *car*, a *truck*, and a *train* exist, and an astute analyst or designer notices that these classes all have common components and are logically related.

- Those common components are factored out into another class (called *vehicle*), and then *car*, *truck*, and *train* are inherited from *vehicle*.



- In this chapter, we'll be exploring in more detail how inheritance is used in designing class hierarchies, and towards the end of the chapter we'll be looking into multiple inheritance and some of the caveats when using it.

An Example of Composition

- Looking at the code below might give a human the idea that a Manager is an Employee, but there's nothing to tell the compiler that.

```
bash$ cat demo-comp.h
     1  class Employee {
     2      string name;
     3      float salary;
     4      // ...
     5  public:
     6      void SetName(string n)  { name = n; }
     7      void SetSalary(float s) { salary = s; }
     8      // ...
     9  };

    10  class Manager {
    11      Employee emp;               // manager's employee record
    12      set<Employee*> group;       // people managed
    13      short level;
    14      // ...
    15  public:
    16      void SetName(string s)  { emp.SetName(s); }
    17      void SetSalary(float s) { emp.SetSalary(s); }
    18      // ...
    19  };
bash$ _
```

- The compiler cannot implicitly convert a *Manager** into an *Employee**, so we cannot create a list of all employees without special code.

An Example of Composition (continued)

- For example, this code won't compile:

```
// ...
int main()
{
    Manager boss("Frank", "Edwards");
    list<Employee*> allEmployees;

    allEmployees.push_back(&boss);      // Compile error!
    // ...
}
```

- Also, virtual functions of the employee will not be passed to the manager without significant support code from both classes.

- We could either write a conversion operator (from *Manager* to *Employee*) or put the address of the *emp* member into the list, but both solutions are inelegant and obscure, and every new class would need similar code.

- Also, virtual functions could be "faked" by providing the *Employee* class with a field that refers to the containing class and the functions in the *Employee* could forward requests to the containing class if there was one. However, the difficulty is in defining a data type for the reference that could apply to all possible containing classes...

Using Inheritance Instead

- A better approach is to state that a *Manager* **is** an *Employee*, thus:

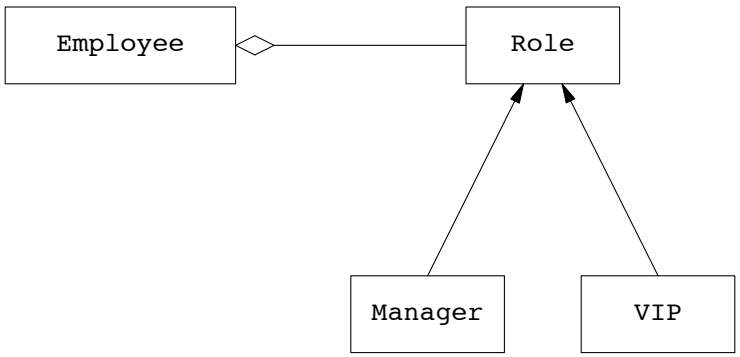
```
bash$ cat demo-inher1.h
     1  class Employee {
     2      string name;
     3      float salary;
     4      // ...
     5  public:
     6      void SetName(string n)  { name = n; }
     7      void SetSalary(float s) { salary = s; }
     8      // ...
     9  };

    10  class Manager : public Employee {
    11      set<Employee*> group;       // people managed
    12      short level;
    13      // ...
    14  public:
    15      void SetLevel(short l)  { level = l; }
    16      // ...
    17  };
bash$ _
```

- The *base* class (*Employee*, in this case) is sometimes also called the *superclass*. The *derived* class is *Manager*, also called the *subclass*, and it *inherits* from its base class all of the data and function members. The derived class can also be thought of as a subtype.

- In actuality, the derived class is the larger of the two classes, as it contains everything in the base class and adds its own functionality as well.

Using Inheritance Instead (continued)

- It should be pointed out that the preceding examples of inheritance may not always be appropriate.
- For instance, it is quite possible for an employee to become a manager. Yet conversion of an object in the hierarchy to another data type is very problematic and should not be done.
- A better technique is to separate the functionality of what it means to be a manager into another class (called "roles", in the OOA&D world).
- Then particular roles can be referenced from within the employee, and roles can be changed by changing a pointer or reference. No conversion of data types is necessary.



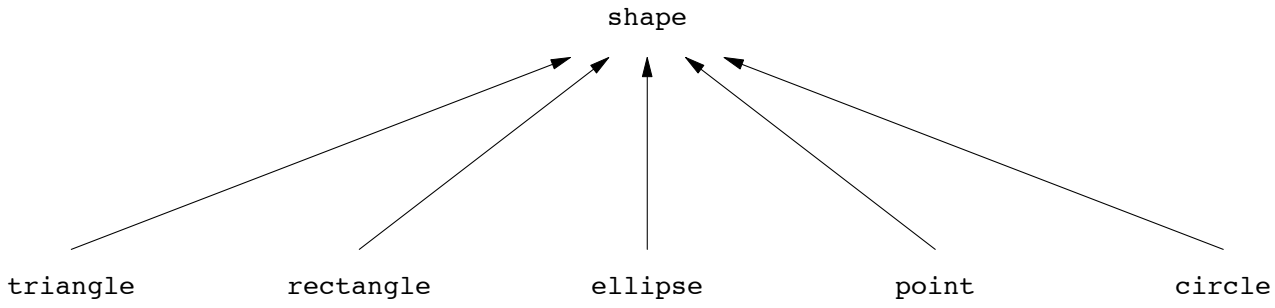
Notes:

Upcasting vs. Downcasting

- The terms *upcasting* and *downcasting* refer to the conversion of objects somewhere on the class hierarchy into objects which exist at a different level of the hierarchy but along the same bloodline (so to speak).

- Upcasting is the conversion of an object into an object of a class higher up on the chain. This can be done implicitly by the compiler if the derivation used the word `public` or `protected` and the code attempting the conversion has the appropriate access.

- Downcasting is problematic since there is no (compile-time) guarantee that the smaller object really is the "smaller part" of a larger object.



Upcasting vs. Downcasting (continued)

- There are four possible solutions for the programmer when a downcast is desired:

  [1] Place a type field in the base class for functions to inspect

  [2] Ensure that only objects of a single type are pointed to

  [3] Use a `dynamic_cast` operator (which does a run-time check of a down-cast)

  [4] Use virtual functions

- Solution 1 is not extensible and requires modification to the base class when classes are added to the inheritance hierarchy. In other words, the base class is never *closed* to future modification.

- Solution 2 is often used with pointers to base classes – container classes such as *list*, *vector*, and *set* are examples.

  — This is the approach commonly taken in Java (prior to v1.5). The downside is that while the downcast is always known to work, the cast is still required. Templates solve this particular problem in a much cleaner fashion in C++, eliminating the need to downcast at all.

- Solution 3 is a language-supported variant of solution 1 that allows *closure* of the base class, but which may require modification of derived classes when the hierarchy changes. This makes it unsuitable in the general case.

Upcasting vs. Downcasting (continued)

- Solution 4 is a special type-safe variation of solution 1.
  - This is the ideal case because both the base class and derived classes can be closed against modifications. Only the newly added class needs to be considered.
  - However, it does require that all type-dependent code be implemented as virtual functions in the base class at the top of the hiearchy. This is not always feasible when the programmer doesn't have control over the base class, such as when using third-party libraries.
- Combinations of solution 2 and 4 are particularly interesting and powerful as in almost all situations they yield cleaner code than do solutions 1 and 3. This can often be implemented using templates.
- Solution 1 is only discussed here to show why it is commonly an inappropriate answer, while solution 4 is described fully as it is typically the best choice.
- Solution 3 is discussed elsewhere in this course.

Solution 1 — The Type-field

- This involves an integer or enumerated type buried inside the base class which identifies the type of object the base class actually represents. This allows functions to perform casts on the object to the appropriate type.

```
bash$ cat demo-type.h
     1  class Employee {
     2      char etype;         // 'E' or 'M' or ...
     3      // ...
     4    public:
     5      Employee(char typ = 'E') : etype(typ) { }
     6      void print() const;
     7      friend void print_list(const Employee *e);
     8  };

     9  class Manager : public Employee {
    10      // ...
    11    public:
    12      Manager() : Employee('M') { }
    13      void print() const;
    14  };
bash$ _
```

Solution 1 — The Type-field (continued)

- Given the previous code, a function to print a list of employees is fairly simple (remember that there are no virtual functions here):

```
bash$ cat demo-type.cpp
     1  #include <iostream>     // For 'cerr'
     2  #include "demo-type.h"

     3  void print_list(const Employee *e)
     4  {
     5      const Manager *m = 0L;
     6      switch (e->etype) {
     7          case 'E' :
     8              e->print();         // Print the employee
     9              break;
    10          case 'M' :
    11              // Shouldn't use old-style casts (next chapter)
    12              m = (const Manager *)(e);
    13              m->print();         // Print the manager
    14              break;
    15          default :
    16              std::cerr << "Unknown employee type!\n";
    17      }
    18  }
bash$ _
```

- The use of the explicit type conversion (the cast operation) is a strong hint that improvement is possible.

Virtual Functions

- Virtual functions overcome the problems of the type-field solution by allowing the programmer to extend the functionality in a derived class without modification to the base class.

```
bash$ cat demo-virt.h
     1  class Employee {
     2      // ...
     3    public:
     4      Employee();
     5      // Notice the use of virtual and const
     6      virtual void print() const;
     7  };

     8  class Manager : public Employee {
     9      // ...
    10    public:
    11      Manager();
    12      // Once virtual, always virtual
    13      void print() const;
    14  };
bash$ _
```

```
bash$ cat demo-virt.cpp
     1  #include <list>
     2  #include "demo-virt.h"

     3  // A reference to an entire set of employees is passed
     4  // in, and we iterate through the set and print the
     5  // information.
     6  void print_list( const std::list<Employee*> &s )
     7  {
     8      // Note the use of "const_iterator" ...
     9      std::list<Employee *>::const_iterator p;

    10      // `*p' is one item in the list -- an Employee pointer
    11      for (p=s.begin(); p != s.end(); p++)
    12          (*p)->print();
    13  }
bash$ _
```

- Notice that this code works even when new derived classes are created not even conceived of by the writer of the *Employee* class!

Virtual Functions (continued)

- Virtual functions are invoked only for pointers or references, never when the compiler has the actual object itself (and hence, the actual data type and function name is known at compile time).

- Virtual function calls can be turned off by using `::` (the scoping operator) to specify a particular class method.

- **Remember**: any class with virtual functions should (probably) have a virtual destructor.

Abstract Base Classes

- Use abstract classes as an interface. When you think of "interface" or "building block", think of abstract base classes. (Java includes direct support for this concept via its *interface* and *implements* keywords.)

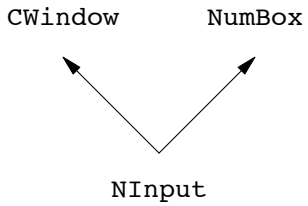
- Consider the *Shape* class discussed earlier in the course. It provides a common interface for all shapes, whether circles, points, triangles, rectangles, or any other shape.

- With careful planning, the application can be configured to not need any information about the actual object types at all.

- For example, a global *list* object is used to keep track of objects of types derived from shape, which could be used by a function which populates a menu with choices.

Multiple Inheritance

- It sometimes makes good sense for a derived class to have two base classes.
- In many situations, one of the base classes can be made into a member object. Unfortunately, member objects don't pass virtual functions through to the containing object as a base class does. (Composition vs. inheritance was discussed earlier in the chapter.)
- Take the example of a generic numeric input field on a user-interface screen, combined with an implementation-specific vendor library.



- In this example, *CWindow* is the implementation of the graphical interface, and *NumBox* is the concept encapsulating a box displayed for numerical input.

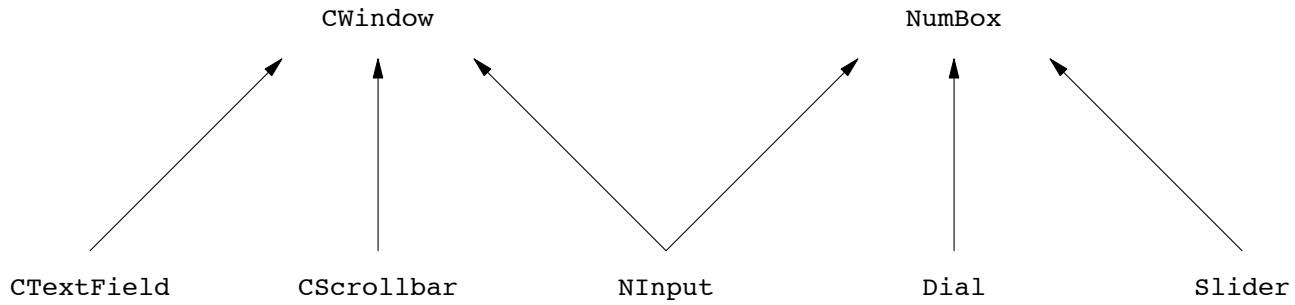
```
class NumBox {
  public:
    virtual void set_value() = 0;       // ... and display
    virtual int get_value() = 0;        // retrieves box value
    virtual void prompt() = 0;          // prompt user
    // ...
};
```

Multiple Inheritance (continued)

- Notice how the numerical box class simply provides an interface for all of the classes derived from it. For example, numerical input based on a slider might be a derived class class called *NInput_slider*, numerical input based on a dial might be a derived class called *NInput_dial*, and so on.

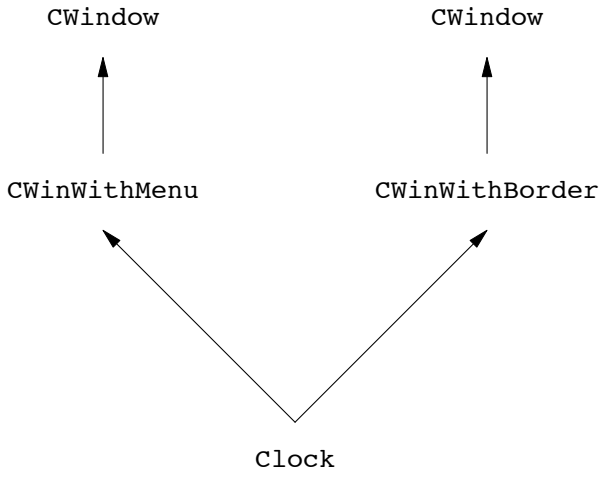
- In each case, those derived classes must implement each of the virtual functions before the compiler will allow instantiation.

- Notice that no data is in the base class itself. It is important in the design stages to avoid putting data in the abstract base classes unless a solid argument for it can be developed. Otherwise, the temptation is to put "other stuff" in there as well, and the design deteriorates so that the base class is simply a data repository instead of an abstraction of a concept.



Multiple Inheritance (continued)

- Multiple inheritance has a few language subtleties associated with it.
- For example, if the classes used as bases have a common ancestor, that ancestor object will appear twice in the *most-derived* class (used to denote the class at the bottom of the hierarchy chart).



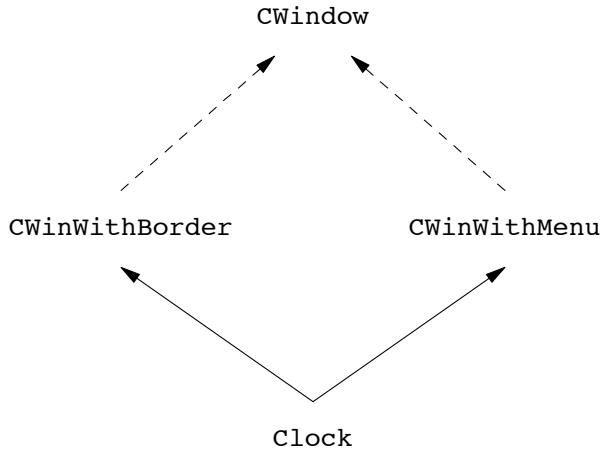
- As you can see, there will be two *CWindow* objects embedded inside the clock. One of them will have a menu and the other will have a border!
- The solution to this problem is for derived classes of *CWindow* to use the `virtual` keyword in the derivation. The next page has an example.

Virtual Base Classes

```
bash$ cat demo-vbase.h
     1  class CWindow {
     2      // ...
     3  };

     4  class CWinWithMenu : virtual public CWindow {
     5      // ...
     6  };

     7  class CWinWithBorder : virtual public CWindow {
     8      // ...
     9  };
bash$ cat demo-vbase2.h
     1  #include "demo-vbase.h"

     2  class Clock : public CWinWithMenu, public CWinWithBorder {
     3      // application-defined details here
     4  };
bash$ _
```

- The application then can derive its own clock class from the classes provided by the library.

- Now the compiler does the same thing with the base class as it does with virtual functions: it creates a hidden pointer within the derived class which points to the actual object (of type *CWindow*, in this example).



Virtual Base Classes (continued)

- There's a different problem now, however — when does the constructor for *CWindow* get called?
  - — It has to be called before the derived class ("base class constructors are called first")
  - — It has to be called before the two base classes' constructors are called
  - — It has to be called only once – each base class constructor will be trying to invoke the constructor for its base class, but the two derived classes of *CWindow* share the same base object, so constructing it twice won't work.

Virtual Base Classes (continued)

- The compiler solves this dilemma by moving the constructor call out of the base classes and into the most-derived class instead.

- This problem only shows its head when the base class doesn't have a default constructor and so the compiler requires an explicit invocation. If a default constructor does exist, the author of the *Clock* needn't worry about the constructor call.

- Because of this problem, two things become obvious:

  (a) an initial design which includes the possibility of multiple inheritance is important since the development of the derived classes will depend on it, and

  (b) a library of classes should already do the multiple inheritance and provide the classes that the application is likely to want, so that the application programmer doesn't have to become entrenched in the details of the library implementation (which is exactly what the library designer is trying to avoid).

- Otherwise, multiple inheritance can be very beneficial when used with care.

- It should be pointed out that multiple inheritance only occasionally produces a hierarchy with a common base class. Most often, the classes using MI come from different vendors, and hence a common base class is unlikely, if not outright impossible.

Method Disambiguation

- Similar to the "when is the constructor called" problem is the problem of multiple in-scope method names and the ambiguity that produces. For example, take a look at the following code:

```
class CWindow {
  public:
    void paint();
};

class CWinWithMenu : public CWindow {
  public:
    void paint();
};

class CWinWithBorder : public CWindow {
  public:
    void paint();
};
```

- Now what happens to the **`paint()`** method when we inherit from both *CWinWithMenu* and *CWinWithBorder*? Since the class with multiple inheritance will get both `paint` methods, how will the compiler choose which one to call?

- The answer is: it doesn't. Having two `paint` methods as shown above will produce a compile-time error in the class that tries to inherit from both. Problem solved!

- In the above situation, the designer of the library will need to include a `paint` method in the new class that disambiguates and gives the compiler a single method to call.

Functional Separation

- One other aspect of multiple inheritance that becomes importance is what to do with methods that call a base class method and then add their own functionality.

- For example, assume that the *CWinWithMenu* and *CWinWithBorder* each have a **`paint()`** method. The typical situation would be for the *Clock*'s **`paint()`** method to simply call its base class methods and be done with it.

- However, this doesn't work.

```
class CWindow {
  public:
    void paint() {
    // clear the window to the background color
};

class CWinWithMenu : virtual public CWindow {
  public:
    void paint();
    // draw the menu after clearing the background
};

class CWinWithBorder : virtual public CWindow {
  public:
    void paint();
    // draw the border after clearing the background
};

class Clock : public CWinWithMenu, public CWinWithBorder {
  public:
    void paint() {
        CWindow::paint();
        CWinWithMenu::paint();
        CWinWithBorder::paint();
        // draw the clock face after drawing the menu and border
    }
};
```

- Notice what happens when the most-derived class's **`paint()`** method is called: the window is cleared, then it's cleared again and a menu painted, then the window is cleared a third time and the border is painted. The clock face is added last. (Yes, the window is cleared three times and the last one will remove the menu.)

Functional Separation (continued)

- The solution to this problem (which occurs in other forms fairly often) is to break the functional requirements of the **`paint()`** method into two separate methods and invoke each of them at the appropriate time.
  - There is a `public` method that the application calls (in this example, it would be **`paint()`**).
  - And there is a `protected` method that only derived classes may access.
- Then each class calls only the protected methods of its base class, because those method implement only the functionality of that particular class and do not call the base class methods as the public functions do.
- Here are the protected functions:

```
class CWindow {
    // ...
  protected:
    void _paint() {
        // clear the window to bkgrd color
    }
};

class CWinWithMenu : virtual public CWindow {
    // ...
  protected:
    void _paint() {
        // draw the menu
    }
};

class CWinWithBorder : virtual public CWindow {
    // ...
  protected:
    void _paint() {
        // draw the border
    }
};
```

Functional Separation (continued)

- And here are the public functions.

```
class CWindow {
    // ...
  public:
    void CWindow::paint() {
        _paint();               // clear the window to bkgrd color
    }
};

class CWinWithMenu : virtual public CWindow {
    // ...
  public:
    void _paint() {
        CWindow::_paint();      // clear the window to bkgrd color
        _paint();               // draw the menu
    }
};

class CWinWithBorder : virtual public CWindow {
    // ...
  public:
    void _paint() {
        CWindow::_paint();      // clear the window to bkgrd color
        _paint();               // draw the border
    }
};
```

Functional Separation (continued)

- When the application programmer wants to create a *Clock* class, they add the following `protected` code, which only implements the functionality specific to the *Clock*:

```
class Clock : public CWinWithMenu, public CWinWithBorder {
    // ...
  protected:
    void _paint() {
        // draw the clock face
    }
};
```

- And a public function that the application can call:

```
class Clock : public CWinWithMenu, public CWinWithBorder {
    // ...
  public:
    void paint() {
        CWindow::_paint();              // clear the background
        CWinWithMenu::_paint();         // draw the menu
        CWinWithBorder::_paint();       // draw the border
        _paint();                       // draw the clock face
    }
};
```

Review

- Why Inheritance?
- Composition
- Upcasting vs. Downcasting
- Virtual Functions
- Abstract Base Classes
- Multiple Inheritance

Advanced C++ Programming

Lab for Inheritance

Lab for Inheritance

1. Goals for this lab:

   A. Experiment with inheritance and virtual functions

   B. Build a sample library hierarchy which uses multiple inheritance

2. In this lab, you'll be creating an example library which implements a derived class with multiple inheritance. Because we don't want to get bogged down in GUIs, we'll use simple **`cout`** statements to determine what is occurring in our classes.

3. Our immediate goal is simply to show that virtual base classes consume more space (hence, they contain the *vcls* pointer) than ordinary base classes. Of course, when you start deriving classes and using multiple inheritance, the size requirement actually decreases, as discussed in the lecture.

4. Make sure that your *Empl* class has only one constructor; the constructor should take a first and last name (`char*`'s). Derive two classes from your *Empl* class, but don't use virtual inheritance. Call them *Cler* and *Tech* (yes, they're the same as the *Introduction to C++* course). Add an integer data field to each of the new classes (use whatever name you like for the data members). The constructors and destructors in all classes should only contain **`cout`** statements and nothing else -- we don't really care about initializing the data members at this point, but we **DO** want to know when they are called.

Lab for Inheritance (continued)

```
bash$ cat exam-lab06a.h
class Empl {
    char fn[20], ln[20];
  public:
    Empl(const char* f, const char* l)
            { cout << "Empl::Empl"  << endl; }
    ~Empl() { cout << "Empl::~Empl" << endl; }
};

class Cler : public Empl {
    int wpm;
  public:
    Cler() : Empl("Cfirst", "Clast")
            { cout << "Cler::Cler"  << endl; }
    ~Cler() { cout << "Cler::~Cler" << endl; }
};

class Tech : public Empl {
    int education;
  public:
    Tech() : Empl("Tfirst", "Tlast")
            { cout << "Tech::Tech"  << endl; }
    ~Tech() { cout << "Tech::~Tech" << endl; }
};
```

5. Now write a *main* function which simply prints out the **`sizeof`** for each class (but **DON'T** actually create any of these objects!) and fill in the sizes for each class in the first data column of the table below. (Leave the last row empty for now.)

Class	**Without** `virtual`	**With** `virtual`
Empl	_____	_____
Cler	_____	_____
Tech	_____	_____
OfficeAdmin	_____	_____

Lab for Inheritance (continued)

6. Now change the *Cler* and *Tech* classes so that they are derived from the *Empl* class with the word `virtual`. Run the program again and record the sizes in the table above. Were the numbers what you expected? (Again, don't create any employees, clerical workers, or technical workers.)

7. Now derive another class from both *Cler* and *Tech*. Call it *OfficeAdmin* (short for Office Administrator). Record the size of this class, when its base classes do and do not use the `virtual` keyword in their derivation. Can you explain the size differences? You may have to run a short test program which prints out the sizes of various built-in data types, such as **`void*`** or **`int`**...

8. Make sure that virtual base classes are **NOT** being used. Now actually create an *OfficeAdmin* in your *main* program. Before running the program, consider (and estimate) the number of times that the constructor and destructor will be called for the *Empl* class. Run the program. What do these results tell you?

9. Now add the `virtual` back into the derivation syntax and run the program again. What did you find out this time? (Hint: it shouldn't compile!) Review your lecture notes to see if you can determine why. Try your various ideas by modifying the constructor for the *OfficeAdmin* constructor...

10. What happens when multiple base classes declare the same function? For example, if *Cler* and *Tech* both declared a *Print* function, how would the compiler resolve a call to *Print* when the actual object was of type *OfficeAdmin*? Try it. Now see if you can figure out a way to make it work. (Note: it may already work, but I had to ask this question just to get you thinking!)

```
// ...

int main()
{
    Empl *e;
    OfficeAdmin joe;

    e = &joe;
    e->Print();
    return 0;
}
```

Lab for Inheritance (continued)

11. **Extra Credit:**

In many circumstances, inheritance can be avoided by using composition instead. The lecture covered situations where this is not true. Given the *Empl* class as it currently exists (from the end of Lab 4), would composition work for either the *Cler* or *Tech* classes? Why or why not? If not, can you think of a way to make it work?

TAB HERE

Advanced C++ Programming

Standard Template Library

Objectives

- What is the Standard Template Library ("STL")?
- What are the goals for the STL?
- Which container classes and iterators are provided?
- What is an **`auto_ptr`** (pronounced "auto pointer")?

What is the STL?

- The Standard Template Library is a general-purpose library of *generic algorithms* and data structures.
- It makes a programmer more productive in two ways:
  - — First, it contains a lot of different components that can be plugged together and used in an application, and
  - — Second, it provides a framework into which different programming problems can be decomposed.
- These two differences, algorithms and data structures, are the yin-yang of programming.
  - — Algorithms are written in terms of iterator categories to access data in a generic manner. The programmer can concentrate on *how* they want to manipulate the data rather than the details of a particular implementation.
  - — Data structures are designed to hold application information as efficiently as possible, allowing the programmer to focus on *using* the data instead of *managing* the data.

What are the Goals of the STL?

- The most important goal is to provide a framework that can easily and efficiently be used by the C++ programmer with their own data structures.

- However, as implementing data structures over and over is tedious and error-prone, the STL has as a secondary goal to provide the common data structures itself. Structures such as linked lists and hash tables are already included.

- By providing these components as part of the library, the programmer can work on application code and know that the library authors have already spent much time and effort on tuning the framework for their particular platform.

- Another auxiliary goal is to produce readable source code. As any experienced programmer will tell you, the worst part of getting up to speed on new code is understanding it at both a very high level ("What does this code do? And how does it do it?") and at a much lower iplementation level ("I want to average the salaries for all employees in these three departments. How do I do that?").

  — By providing generic algorithms for iterating over a list of employees, a programmer who knows the basics of the STL can quickly get an idea of the big picture behind the application. (Of course, documentation would be better, but we all know how that goes, don't we?)

  — Similarly, those same algorithms might be used on a dozen different classes throughout the application, but they always do the same thing so the implementation details of an application are easier to focus on when necessary.

Which Container Classes are Provided?

Classname	Iterator Type	Filename	Description
Associative Containers			
`set<K>` `multiset<K,T>`	Bidirectional	`<set>`	Sets have keys, but no values. Multiset keys are not unique.
`map<K>` `multimap<K,T>`	Bidirectional	`<set>`	Maps have keys and values. Multimap keys are not unique.
Sequence Containers			
`T a[n]`	Random	`-`	Built-in array type in C/C++
`vector<T>`	Random	`<vector>`	Vectors mimic dynamically sizable arrays
`deque<T>`	Random	`<deque>`	Deques are queues that can be accessed from either end
`list<T>`	Bidirectional	`<list>`	Lists contain nodes that are connected from one to the next

- The following requirements must hold true for classes to be used by the generic algorithms in the STL (some algorithms may work without all of these being defined).

  — Any object to be stored in a container class must implement the **==** operator so that the container class can make comparisons.

  — For merging and sorting, the object must have the **<** operator defined so that the container can make decisions about lexical ordering. Interestingly, this operator is defined for the container classes themselves. This allows containers to be stored inside other containers and to be "sorted" (what that means depends on how the container defines these operators).

Which Iterator Types are Provided?

- The generic algorithms discussed in the next section require parameters that are capable of iterating through a sequence of values.

- Some iterators have more features than others, so the algorithms are defined in terms of the simplest possible implementation.

  — For example, the **Input** iterator is for reading values from a list. It can move forward through a sequence, but not backwards, and it does not support random access. It defines **`operator==`** and **`operator!=`** so that iterators themselves can be compared to determine whether they point to the same element of a sequence. It also requires **`operator++()`**, **`operator++(int)`**, and **`operator*()`**.

  — Another iterator very similar to **Input** is **Output**. The only difference is that it supports writing new values instead of reading them, and it does not require the equality and inequality operators.

  — And the **Forward** iterator is a combination of the previous two; it allows both read and write, but only moving in a forward direction through the sequence. It also requires **`operator=()`** to be implemented.

  — **Bidirectional** iterators add the ability to move backwards, so they require **`operator--()`** and **`operator--(int)`**.

  — **Random Access** iterators can do everything the above can do and also can jump around within the sequence. This means adding **`operator+(int)`**, **`operator+=(int)`**, **`operator-(int)`**, and **`operator-=(int)`**. It also requires the relational operators **`operator<()`**, **`operator<=()`**, **`operator>()`**, and **`operator>=()`**.

Which Iterator Types are Provided? (continued)

- There are even iterators for data streams: Input iterators for reading data and output iterators for writing data. A previous example in the *Templates* chapter demonstrated **`ostream_iterator`**.

Iterator name	Description
`Input†`	Unidirectional; reading
`Output†`	Unidirectional; writing
`Forward`	Unidirectional; reading and writing; capable of multipass
`Bidirectional†`	Bidirectional; reading and writing
`Random Access`	Jump to any element; reading and writing

†Note: The prefix and postfix operators must be constant time operations or performance will suffer greatly.

- You might think that the iterator with the most features will be the one you'll want to use most often, but actually that's not true; you want the iterator with the *least* features that still meets your criteria!

  — The reason for this is that your iterator will work with a larger set of generic algorithms if it is a simple iterator, and the algorithm itself can be more efficient. This is an argument similar to using RISC technology instead of CISC.

  — However, if you need the ability to jump around within a sequence of values, then by all means use the random access iterators. But be aware that they require more code (and often more *complex* code) to support their use.

Which Generic Algorithms are Provided?

- The iterators of the previous section are designed to work with the generic algorithms of the STL. These algorithms are defined in terms of which iterators are acceptable as parameters.

- All algorithms come from a single header file, **`<algorithm>`**.

Algorithm	Iterator	Description
`accumulate`	`Input`	Adds all elements from first to last
`find`	`Input`	Returns an iterator to the first occurence of a value
`merge`	`Input,Output`	Merges two containers into a third container

- For example, the **`accumulate`** algorithm requires two input iterators. It reads data from the first iterator until it reaches the second iterator. All of the values are added to an initializer value provided as the third parameter. And the result is returned to the caller.

- Because all other iterators are built on top of the input and output iterators, any kind of iterator can be used in calling the **`accumulate`** function.

- The **`merge`** function requires four input iterators (the begin and end iterators for two sequences) and an output iterator for the destination.

- A huge advantage of the algorithms is that normal C/C++ arrays are a valid data structure and pointers are valid iterators! A pointer is a random access iterator, since it supports all of the features up the hierarchy. Pointers can point to **`const`** data if they represent values that should be considered read-only.

Some Examples of What We've Seen

- Okay, enough with the basics. Let's look at some code to see how these features are used.
- You've already seen the **`list`** class used a few times in previous examples and we've asked you to just bear with us. Now we're ready to look at them in detail.
- We'll start with a simple program that generates a series of random numbers to store into a **`list`**, then we'll go back and process those values in a number of different ways.
- First we'll take a look at the output, then the actual code...

Some Examples of What We've Seen (continued)

```
bash$ exam-stl-list1 20
List of all random values being used:
   102   113    84    19   -97   -34   -30   104    64   -81
   -53    68   -66  -115    61    75    72  -116   -34    87
Result of adding them up: 223

Number of negative values: 9
Number of positive values: 11

Number of times each bit was set (average is 10):
  bit:     7     6     5     4     3     2     1     0
  qty:     9    13     7     9    12    12    11     9
bash$ _
```

```
bash$ exam-stl-list1 20
List of all random values being used:
    60    87   114    65   -25   117   -25    -3     2    10
    23   -18    37  -115    18   -76     2   -37    95   -39
Result of adding them up: 292

Number of negative values: 8
Number of positive values: 12

Number of times each bit was set (average is 10):
  bit:     7     6     5     4     3     2     1     0
  qty:     8    11     9    11     8    12    12    12
bash$ _
```

```
bash$ exam-stl-list1 20
List of all random values being used:
   -40    51   -60   -86   109   114    -2   -21    15   -97
   -16    39   -21    46   -66   -87     8    76    67  -102
Result of adding them up: -73

Number of negative values: 11
Number of positive values: 9

Number of times each bit was set (average is 10):
  bit:     7     6     5     4     3     2     1     0
  qty:    11    10    12     8    14     9    13     9
bash$ _
```

Some Examples of What We've Seen (continued)

```
bash$ cat exam-stl-list1.cpp
     1  #include <iostream>     // For cout
     2  #include <iomanip>      // For setw()
     3  #include <list>         // For list<T>
     4  #include <numeric>      // All numeric algorithms
     5
     6  using namespace std;
     7  typedef list<int> MySequence;
     8
     9  int main(int argc, char **argv)
    10  {
    11      int qty = 20;               // Default to 20 elements
    12      if (argc > 1)
    13          qty = atoi(argv[1]);    // but let the user specify
    14
    15      MySequence nums;
    16      srandomdev();
    17      for (int i=qty; i-- > 0; )
    18          nums.push_back(static_cast<int>(random()&255)-128);
    19
    20      // Start by printing out the values.
    21      {
    22          cout << "List of all random values being used:"
    23               << endl;
    24          int count = 0;
    25          MySequence::const_iterator begin = nums.begin();
    26          while (begin != nums.end()) {
    27              cout << setw(6) << *begin++;
    28              if (++count % 10 == 0)
    29                  cout << endl;
    30          }
    31          if (count % 10 != 0)
    32              cout << endl;
    33      }
    34
    35      // If the random number generator is truly random, we
    36      // should be able to add up all the number and get zero,
    37      // or at least something close to zero.
    38
    39      // Initialize sum to zero, then start adding'em up...
    40      int result = accumulate(nums.begin(), nums.end(), 0);
    41      cout << "Result of adding them up: "
    42           << result << endl;
    43
```

Some Examples of What We've Seen (continued)

```
    44      // Count how many are negative vs. positive.
    45      {
    46          int count[2] = { 0 };   // 0=negative, 1=positive
    47          MySequence::const_iterator begin = nums.begin();
    48          while (begin != nums.end()) {
    49              count[ (*begin++ < 0) ? 0 : 1 ]++;
    50          }
    51          cout << endl;
    52          cout << "Number of negative values: "
    53                    << count[0] << endl;
    54          cout << "Number of positive values: "
    55                    << count[1] << endl;
    56      }
    57
    58      // Only bits 0..7 are used in the values.  Let's count
    59      // how many times each bit was turned on.  There are
    60      // two ways to implement this; this is the easy way!
    61      cout << endl;
    62      cout << "Number of times each bit was set (average is "
    63           << qty/2 << "):" << endl;
    64      cout << "  bit:";
    65      int bits[8] = { 0 };
    66
    67      for (int bit=8; --bit >= 0; ) {
    68          cout << setw(6) << bit;
    69          MySequence::const_iterator begin = nums.begin();
    70          while (begin != nums.end()) {
    71              bits[bit] += (*begin++ & (1 << bit)) ? 1 : 0;
    72          }
    73      }
    74      cout << endl << "  qty:";
    75      for (int bit=8; --bit >= 0; ) {
    76          cout << setw(6) << bits[bit];
    77      }
    78      cout << endl;
    79      return 0;
    80  }
bash$ _
```

Some Examples of What We've Seen (continued)

- Using the **`list`** class is recommended when data needs to be inserted into the middle of the sequence on a regular basis. This is because the **`list`** class has a constant-time insertion anywhere in the list, while the **`vector`** class, for instance, is linear-time insertion: the closer to the front of the sequence that the new data is inserted, the longer the insertion takes because more elements need to be shifted.

- Because our previous example didn't do any insertions, we shouldn't see much of a performance difference between **`list`**'s and **`vector`**'s.

- To prove that, let's do some timings, first with **`list`**s and then with **`vector`**s.

```
bash$ time exam-stl-list1 2000000 > /dev/null
real    0m1.879s
user    0m1.802s
sys     0m0.052s
bash$ time exam-stl-vec1 2000000 > /dev/null
real    0m1.321s
user    0m1.265s
sys     0m0.028s
bash$ _
```

- Wow! The **`vector`** was about 30% faster than the **`list`**! But why would that be? We know that the code is the same except for the data structure used. Iterating through a data structure that's been built already should be virtually identical.

- The only other portion that might vary significantly is the loop that populates the data structure in the first place. Apparently, putting 2 million entries into a **`vector`** is quite a bit faster than putting 2 million entries into a **`list`**.

- However, in most applications, the creation and population of the data structure is amortized over a longer period of time. For example, the database is read in chunks of a thousand or so before being presented to the user.

- But what about using those two classes together? Here are some example programs that use the **`merge`** and **`find`** algorithms to show that it's the iterators that provide the flexibility!

Some Examples of What We've Seen (continued)

```
bash$ cat exam-stl-mix1.cpp
     1  #include <iostream>     // For cout
     2  #include <iomanip>      // For setw()
     3  #include <list>         // For list<T>
     4  #include <vector>       // For vector<T>
     5  #include <numeric>      // All numeric algorithms
     6  using namespace std;
     7
     8  typedef list<int>   MySequence1;
     9  typedef vector<int> MySequence2;
    10
    11  template <typename T>
    12  void printContainer(T from, T to)
    13  {
    14      const int valsPerLine = 8;
    15      int count = 0;
    16      while (from != to) {
    17          cout << setw(6) << *from++;
    18          if (++count % valsPerLine == 0)
    19              cout << endl;
    20      }
    21      if (count % valsPerLine != 0)
    22          cout << endl;
    23  }
    24
    25  int main(int argc, char **argv)
    26  {
    27      MySequence1 nums1;
    28      MySequence2 nums2;
    29
    30      srandomdev();
    31      // Fill the list with positive numbers and the vector
    32      // with negative numbers.
    33      for (int i=8; i-- > 0; ) {
    34          nums1.push_back( static_cast<int>(random()&255) );
    35          nums2.push_back(static_cast<int>(random()&255)-256);
    36      }
    37      cout << "List of all random values in the list:"
    38           << endl;
    39      printContainer(nums1.begin(), nums1.end());
    40
    41      cout << "List of all random values in the vector:"
    42           << endl;
    43      printContainer(nums2.begin(), nums2.end());
```

Some Examples of What We've Seen (continued)

```
    44
    45      // Make room for the results
    46      MySequence2 combo(nums1.size() + nums2.size());
    47      merge(nums1.begin(), nums1.end(),
    48            nums2.begin(), nums2.end(),
    49            combo.begin());
    50      cout << "List of all values after merging:" << endl;
    51      printContainer(combo.begin(), combo.end());
    52
    53      sort(combo.begin(), combo.end());
    54      cout << "List of all values after sorting:" << endl;
    55      printContainer(combo.begin(), combo.end());
    56      return 0;
    57  }
bash$ _
```

```
bash$ exam-stl-mix1
List of all random values in the list:
   206    33   146   194    29   167   136   114
List of all random values in the vector:
  -103   -89  -232  -102  -126   -35  -194    -3
List of all values after merging:
  -103   -89  -232  -102  -126   -35  -194    -3
   206    33   146   194    29   167   136   114
List of all values after sorting:
  -232  -194  -126  -103  -102   -89   -35    -3
    29    33   114   136   146   167   194   206
bash$ _
```

Some Examples of What We've Seen (continued)

- Did you notice that the *combo* object has to have it's size preallocated in the constructor call? This is because the **`merge`** function uses the iterator to perform **`*i = ...`** and that is an overwrite operation. However, the next code snippet shows how we can work around that.

```
MySequence combo;
std::merge(nums1.begin(), nums1.end(),
           nums2.begin(), nums2.end(),
           back_inserter(combo));
```

- The templated function called **`back_inserter`** will use the passed in parameter to create an *iterator adaptor*; that is, an iterator that adapts the object (in this case, *combo*) for a different use.

  — Any class which provides a **`push_back()`** method can be used with **`back_inserter`**. The iterator adaptor modifies **`*i = ...`** to be an insert (via **`push_back()`**) instead of an overwrite.

Some Examples of What We've Seen (continued)

```
bash$ cat exam-stl-mix2.cpp
     1  #include <iostream>     // For cout
     2  #include <iomanip>      // For setw()
     3  #include <list>         // For list<T>
     4  #include <vector>       // For vector<T>
     5  #include <algorithm>    // All non-numeric algorithms
     6  using namespace std;
     7
     8  typedef list<int>   MySequence1;
     9  typedef vector<int> MySequence2;
    10
    11  template <typename T>
    12  void printContainer(T from, T to)
    13  {
    14      const int valsPerLine = 8;
    15      int count = 0;
    16      while (from != to) {
    17          cout << setw(6) << *from++;
    18          if (++count % valsPerLine == 0)
    19              cout << endl;
    20      }
    21      if (count % valsPerLine != 0)
    22          cout << endl;
    23  }
    24
    25  int main(int argc, char **argv)
    26  {
    27      MySequence1 nums1;
    28      MySequence2 nums2;
    29
    30      srandomdev();
    31      // Fill the list with positive numbers and the vector
    32      // with negative numbers.
    33      for (int i=8; i-- > 0; ) {
    34          nums1.push_back( static_cast<int>(random()&255) );
    35          nums2.push_back(static_cast<int>(random()&255)-256);
    36      }
    37      cout << "List of all random values in the list:"
    38           << endl;
    39      printContainer(nums1.begin(), nums1.end());
    40
    41      cout << "List of all random values in the vector:"
    42           << endl;
    43      printContainer(nums2.begin(), nums2.end());
```

Some Examples of What We've Seen (continued)

```
    44
    45      // Make room for the results
    46      MySequence2 combo;
    47      merge(nums1.begin(), nums1.end(),
    48            nums2.begin(), nums2.end(),
    49            back_inserter(combo));
    50      cout << "List of all values after merging:" << endl;
    51      printContainer(combo.begin(), combo.end());
    52
    53      sort(combo.begin(), combo.end());
    54      cout << "List of all values after sorting:" << endl;
    55      printContainer(combo.begin(), combo.end());
    56      return 0;
    57  }
bash$ _
```

```
bash$ exam-stl-mix2
List of all random values in the list:
    26    59    89   230   238   126   250   129
List of all random values in the vector:
   -47   -43    -8  -243  -109  -181   -12  -162
List of all values after merging:
   -47   -43    -8  -243  -109  -181   -12  -162
    26    59    89   230   238   126   250   129
List of all values after sorting:
  -243  -181  -162  -109   -47   -43   -12    -8
    26    59    89   126   129   230   238   250
bash$ _
```

Review

- The STL contain *generic algorithms* that can be used with a wide variety of data types.
- These algorithms are implemented as efficiently as possible for the architecture of the machine by the compiler vendor (some do a better job than others!).
- Many of these algorithms are designed to work with different types of *iterators*. If the algorithms work with iterators and the container classes provide those iterators, the algorithms can be used with many container classes. And adding a new container that the STL authors didn't think of only requires adding an efficient iterator.
- Iterators are fail-fast, meaning that if some operation invalidates the iterator (such as modifying a link list while iterating over it) the iterator will throw an exception.
- The flexibility of the STL comes at a price: when used properly it can greatly increase the readability and performance of an application, but when used improperly it can increase the size of the application code and require extra layers of software to manage compatibility with multiple container types.

TAB HERE

Advanced C++ Programming

Miscellaneous

Objectives

- Iterator Concepts
- Exception Handling
- New-style Cast Operations
- Run-Time Type Identification

Miscellaneous

- This chapter covers some miscellaneous topics which have not already been covered in any detail in previous chapters.

- For example, the idea of iterators and how they're used was discussed briefly in the chapter on templates, but the full discussion has been deferred until now.

- Exception handling is a technique for returning error codes to calling functions without the use of return values. This allows a function to return, for example, an `int` but to throw a *DivisionByZeroException* as an error condition independent of any declared return value.

- Run-time type identification hasn't been mentioned previously, and this is due partly to the author's opinion that, like multiple inheritance, it should be avoided unless absolutely necessary. And generally, beginning designers/programmers will not have a good grasp of what is "necessary" until they have worked in C++ for some time.

Iterator Concepts

- The first topic is iterators.
- The Standard C++ Library provides quite a few *container classes* as templates, ready to be used by the application (or library) programmer.
- These containers need to provide some technique to allow the programmer to examine each element in turn.
- This is done with iterators. They provide something analagous to using a pointer to iterate through an array.
- Each container class declares a typedef for *iterator*, *const_iterator*, *reverse_iterator*, and *reverse_const_iterator* for iterating through the container.
- The actual type of the iterator will vary from class to class (after all, a vector iterator would function differently than a linked-list iterator).
- In fact, iterator classes are usually *friend*s of the container class so that internal information about the container can be obtained easily without exposing it with a public interface.

```
bash$ cat demo-template2.cpp
     1  #include <iostream>
     2  #include <iterator>
     3  #include <list>

     4  using namespace std;

     5  template <class T>
     6  ostream &operator << (ostream &os, const list<T>& t)
     7  {
     8      // Copy from begin to end...
     9      copy(t.begin(), t.end(), ostream_iterator<T>(os, "\n"));
    10  }
bash$ _
```

- Other member functions of the container class return reverse iterators, *rbegin()* and *rend()*. Of course, they are simply the opposite ends of the sequence.

Exception Handling

- Exception handling is the handling of errors without the programmer having to check the return value of every funtion call.
- For example, assume that a particular loop performs I/O operations on up to 4 files at a time.
- The easiest way to handle error checking would be to have the individual functions being called notify the calling subroutine that something had gone wrong (this would be much easier than checking every return value).
- This is precisely the functionality provided by exception handling.
- Of course, there are a few details. Such as stack unwinding and constructed objects being destroyed...
- The following example implements a *merge sort*.
  - — It opens up to 20 files at once.
  - — If any attempt to open a file fails, an exception is thrown.
  - — In this example, the exception is thrown locally and handled locally, but that's unusual. Typically the exception is thrown in a low-level function and caught at a higher level in the call stack.

Exception Handling (continued)

```
bash$ cat demo-excpt.cpp
     1  #include <list>
     2  #include <string>
     3  #include <iostream>
     4  #include <fstream>
     5  using namespace std;

     6  struct BadFileOpen {
     7      string which;
     8      BadFileOpen(string s) : which(s) { }
     9  };

    10  class FileList : public list<ifstream*> {
    11    public:
    12      ~FileList() {
    13          // Loop through our internal list of pointers and
    14          // delete each one.  This closes the stream as well.
    15          while (size() > 0) {
    16              delete back();
    17              pop_back();
    18          }
    19      }
    20  };

    21  int main()
    22  {
    23      list<string> filelist;
    24      FileList openfiles;

    25      filelist.push_back("demo-excpt.cpp");
    26      try {
    27          // If we exit the "try" block, call the destructor.
    28          ofstream output("output.txt");

    29          // do a merge sort...
    30          const int MAXFILES = 20;
    31          while (filelist.size() > 0) {
    32              // inner loop 1 - open files
    33              while (filelist.size() > 0 &&
    34                     openfiles.size() < MAXFILES) {
    35                  ifstream *next =
    36                      new ifstream(filelist.back().data());
    37                  if (!next)
    38                      throw BadFileOpen( filelist.back() );
```

Exception Handling (continued)

```
    39                  filelist.pop_back();
    40                  openfiles.push_back(next);
    41              }
    42              // Here when MAXFILES files are open or
    43              // list of files is empty.

    44              // inner loop 2 - merge/sort files
    45              while (openfiles.size() > 0) {
    46                  // Merge sort the data streams that are
    47                  // in 'openfiles'.  When all streams
    48                  // have been merged, the loop will break.
    49                  //
    50                  // Actual implementation of the merge is
    51                  // left as an exercise for the student. :)
    52              }
    53          }
    54      }
    55      catch (BadFileOpen &bfo) {
    56          // Display an error and terminate the merge
    57          cerr << bfo.which << endl;
    58      }
    59      catch (...) {
    60          cerr << "Some other error occurred..." << endl;
    61          // Rethrow exception up to the next level.  This
    62          // will terminate the application since we're
    63          // inside main().
    64          throw;
    65      }
    66      // This message is printed if:
    67      //    the BadFileOpen exception is thrown, or
    68      //    the merge completes successfully.
    69      cout << "Completed!" << endl;
    70      return 0;
    71  }
bash$ _
```

Exception Handling (continued)

- The following are the primary motivations for exceptions:
  - State variables to keep track of "an error occurred" are not needed when using exceptions.
  - It is a better strategy to keep error-handling code separate from "normal" code as much as possible.
  - Doing error-handling at the same level of abstraction as the code that caused the error might repeat the same error that triggered the error handling in the first place!
  - It is generally more work to modify the "normal" code to add error-handling code than to add separate error-handling routines.
- Exception handling is designed for dealing with non-local errors. If an error can be handled locally, it almost always should be.
- Lastly, don't use exceptions when loop control structures are sufficient.

New-style Cast Operations

- The latest version of the language (ISO/IEC 14882:2011, as of this writing) includes the specification of 4 versions of C++-specific cast operations.
- The use of C language casts is frowned upon since there are many ambiguities with their application.
- There are four *new-style casts*, as they are called. They are listed on the following page.

New-style Cast Operations (continued)

`const_cast<``type`**`>(`***arg*`)`**

Can only be used to remove `const`-ness from a object, pointer, or reference. *Type* and *arg* must be of exactly the same types except for the **`const`** modifier.

`static_cast<``type`**`>(`***arg*`)`**

The **`static_cast`** operator converts between related types such as one pointer to another, an enumeration to an integral type, or a floating-point type to an integral type. Some of these casts are portable.

`dynamic_cast<``type`**`>(`***arg*`)`**

This cast converts a pointer (or reference) *arg* into a pointer (or reference) of type "*type*" if such a conversion is valid, otherwise it returns a null pointer (or throws an exception). A null pointer is also returned if *arg* is a null pointer. The conversion requires that *arg* be a unique base class of *type* and that it is a polymorphic type (that is, that it contain virtual functions).

— The requirement for being polymorphic makes the implementation easier since a pointer to a "type information object" can be included in the vtbl for the class. Such an implementation makes sense from a logical perspective as well.

— *Type* need not be polymorphic.

— **`dynamic_cast`**s of references cannot return a null reference, so such cast operations throw the *bad_cast* exception instead of returning some kind of null value.

`reinterpret_cast<``type`**`>(`***arg*`)`**

This cast handles conversions between unrelated types such as an integer to a pointer. Few of these are portable.

Run-Time Type Identification

- **`dynamic_cast`** serves most purposes for determining the run-time type information for objects involved in a class hierarchy.
- But it is occasionally necessary to know the *exact* type of an object.
- This information can be obtained with the `typeid` operator. It returns an object representing the type of its operand.

```
#include <typeinfo>
const type_info& typeid(type_name) throw(bad_typeid);
const type_info& typeid(expression);
```

- If the value of a pointer or reference operand is 0, a *bad_typeid* exception is thrown.
- There is no guarantee that only a single `type_info` structure exists as the information block for a type. (In fact, it can be nearly impossible to guarantee when shared libraries are considered.) Consequently, the `==` operator should be used on `type_info` objects, and not `type_info` pointers.
- Usually we want to know the exact type of an object to provide some kind of common service. Ideally, such a service would be provided as a virtual function.
- Unfortunately, no assumption can be made that such a function exists for all types in use within a given application.
- Another simpler use is to obtain the name of an object class for diagnostic purposes.
- The programmer should use RTTI only when necessary. Static checking is safer, implies less overhead, and — where applicable — leads to better-structured programs.

Review

- Iterator Concepts
- Exception Handling
- New-style Cast Operations
- Run-Time Type Identification
- C++ is still evolving. The most recent iteration includes lambda functions, initializer lists for arrays of objects, new `for` loop syntax, a `nullptr` keyword, thread-local storage, and many more features. A good place to start is the C++11 page on wikipedia: *http://en.wikipedia.org/wiki/C%2B%2B11*

Notes: