C++ is fun – Part 14

at Turbine/Warner Bros.!

Russell Hanson

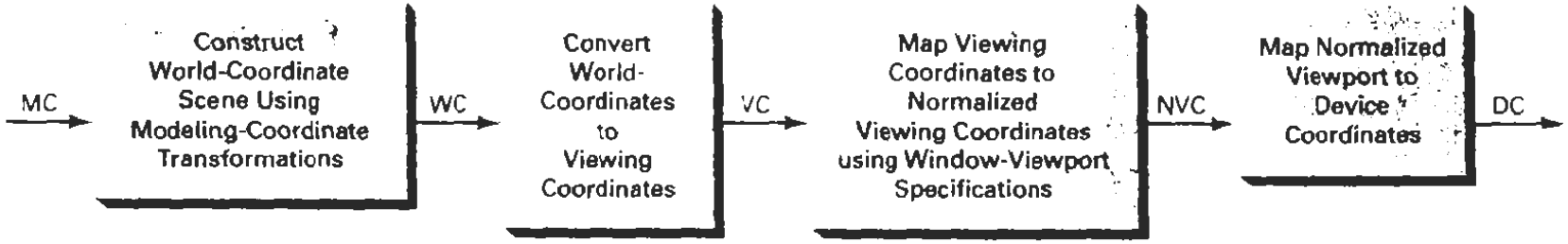
Syllabus

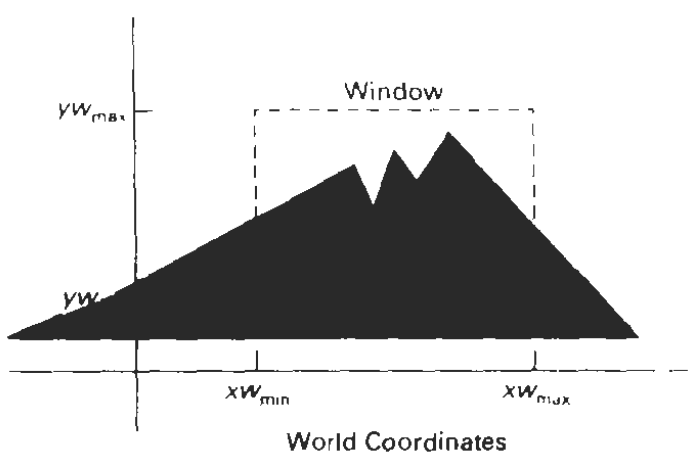
1) First program and introduction to data types and control structures with applications for games learning how to use the programming environment Mar 25-27
2) Objects, encapsulation, abstract data types, data protection and scope April 1-3
3) Basic data structures and how to use them, opening files and performing operations on files – April 8-10
4) Algorithms on data structures, algorithms for specific tasks, simple AI and planning type algorithms, game AI algorithms April 15-17
Project 1 Due – April 17
5) More AI: search, heuristics, optimization, decision trees, supervised/unsupervised learning – April 22-24
6) Game API and/or event-oriented programming, model view controller, map reduce filter – April 29, May 1
7) Basic threads models and some simple databases SQLite May 6-8
8) Graphics programming, shaders, textures, 3D models and rotations May 13-15
Project 2 Due May 15
9) How to download an API and learn how to use functions in that API, Windows Foundation Classes May 20-22
10) Designing and implementing a simple game in C++ May 27-29
11) Selected topics – Gesture recognition & depth controllers like the Microsoft Kinect, Network Programming & TCP/IP, OSC June 3-5
12) Working on student projects - June 10-12
Final project presentations Project 3/Final Project Due June 12

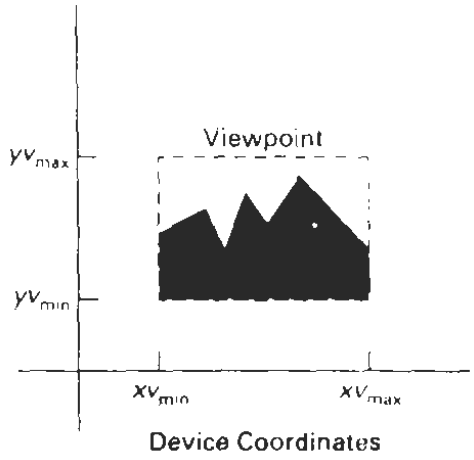
Projects 2 Due in Class This Wednesday

- Please prepare a ~10 minute discussion (PowerPoint, or other) about your program, or your goal with your project, challenges you encountered along the way, how you overcame them, how they overcame you, etc.
- Share it with the class! 8)

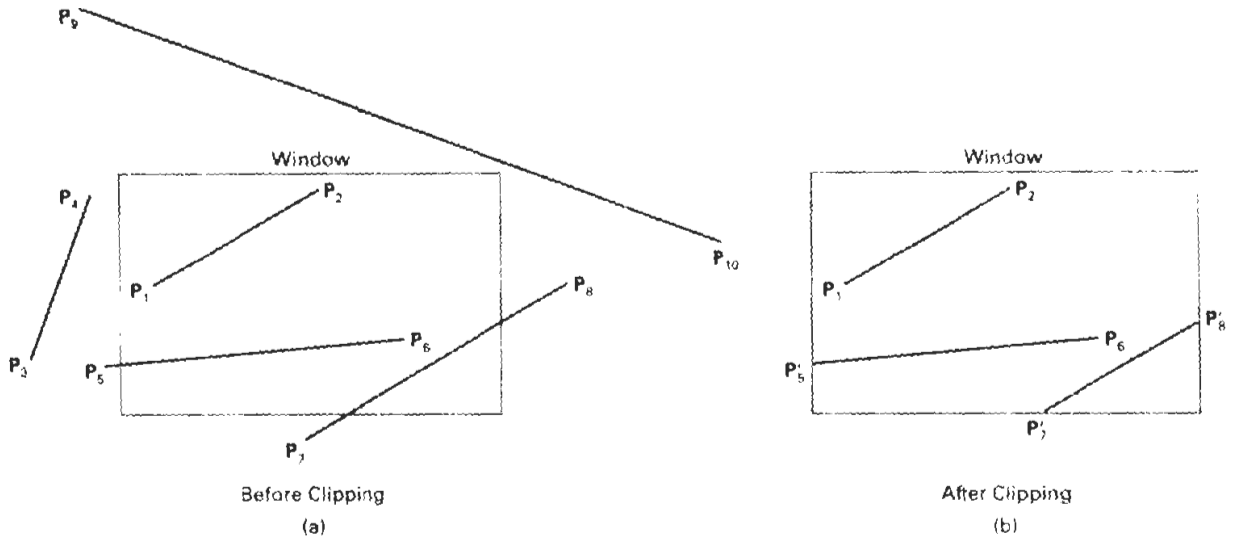
Two dimensional viewing-transformation pipeline







Line clipping against a rectangular clip window



Line Clipping Algorithms

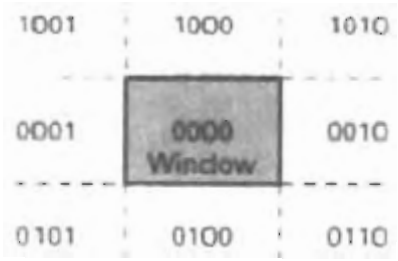


Figure 6-8
Binary region codes assigned to line endpoints according to relative position with respect to the clipping rectangle.

bit 1: left
bit 2: right
bit 3: below
bit 4: above

A value of 1 in any bit position indicates that the point is in that relative position; otherwise, the bit position is set to 0. If a point is within the clipping rectangle, the region code is 0000. A point that is below and to the left of the rectangle has a region code of 0101.

Bit values in the region code are determined by comparing endpoint coordinate values (x, y) to the clip boundaries. Bit 1 is set to 1 if $x < xw_{min}$. The other three bit values can be determined using similar comparisons. For languages in which bit manipulation is possible, region-code bit values can be determined with the following two steps: (1) Calculate differences between endpoint coordinates and clipping boundaries. (2) Use the resultant sign bit of each difference calculation to set the corresponding value in the region code. Bit 1 is the sign bit of $x - xw_{min}$; bit 2 is the sign bit of $xw_{max} - x$; bit 3 is the sign bit of $y - yw_{min}$; and bit 4 is the sign bit of $yw_{max} - y$.

Once we have established region codes for all line endpoints, we can quickly determine which lines are completely inside the clip window and which are clearly outside. Any lines that are completely contained within the window boundaries have a region code of 0000 for both endpoints, and we trivially accept these lines. Any lines that have a 1 in the same bit position in the region codes for each endpoint are completely outside the clipping rectangle, and we trivially reject these lines. We would discard the line that has a region code of 1001 for one

Linked Lists

Introduction to the Linked List ADT

CONCEPT: **Dynamically allocated data structures may be linked together in memory to form a chain.**

A linked list is a series of connected *nodes*, where each node is a data structure. A linked list can grow or shrink in size as the program runs. This is possible because the nodes in a linked list are dynamically allocated. If new data need to be added to a linked list, the program simply allocates another node and inserts it into the series. If a particular piece of data needs to be removed from the linked list, the program deletes the node containing that data.

Advantages of Linked Lists over Arrays and `vectors`

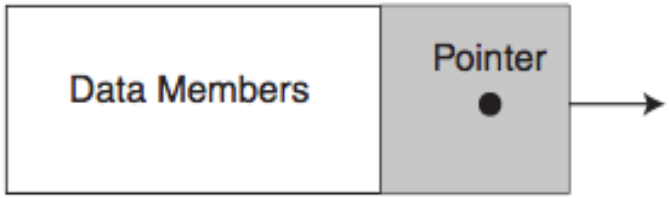
Although linked lists are more complex to code and manage than arrays, they have some distinct advantages. First, a linked list can easily grow or shrink in size. In fact, the programmer doesn't need to know how many nodes will be in the list. They are simply created in memory as they are needed.

One might argue that linked lists are not superior to `vectors` (found in the Standard Template Library), because `vectors`, too, can expand or shrink. The advantage that linked lists have over `vectors`, however, is the speed at which a node may be inserted into or deleted from the list. To insert a value into the middle of a `vector` requires all the elements below the insertion point to be moved one position toward the `vector`'s end, thus making room for the new value. Likewise, removing a value from a `vector` requires all the elements below the removal point to be moved one position toward the `vector`'s beginning. When a node is inserted into or deleted from a linked list, none of the other nodes have to be moved.

The Composition of a Linked List

Each node in a linked list contains one or more members that represent data. (Perhaps the nodes hold inventory records, or customer names, addresses, and telephone numbers.) In addition to the data, each node contains a pointer, which can point to another node. The makeup of a node is illustrated in Figure 17-1.

Figure 17-1



A linked list is called "linked" because each node in the series has a pointer that points to the next node in the list. This creates a chain where the first node points to the second node, the second node points to the third node, and so on. This is illustrated in Figure 17-2.

Figure 17-2

Declarations

So how is a linked list created in C++? First you must declare a data structure that will be used for the nodes. For example, the following `struct` could be used to create a list where each node holds a `double`:

```
struct ListNode
{
   double value;
   ListNode *next;
};
```

The first member of the `ListNode` structure is a `double` named `value`. It will be used to hold the node's data. The second member is a pointer named `next`. The pointer can hold the address of any object that is a `ListNode` structure. This allows each `ListNode` structure to point to the next `ListNode` structure in the list.

Because the `ListNode` structure contains a pointer to an object of the same type as that being declared, it is known as a *self-referential data structure*. This structure makes it possible to create nodes that point to other nodes of the same type.

The next step is to define a pointer to serve as the list head, as shown here.

```
ListNode *head;
```

Before you use the `head` pointer in any linked list operations, you must be sure it is initialized to NULL, because that marks the end of the list. Once you have declared a node data structure and have created a NULL `head` pointer, you have an empty linked list. The next step is to implement operations with the list.

```
// Specification file for the NumberList class
#ifndef NUMBERLIST_H
#define NUMBERLIST_H

class NumberList
{
private:
  // Declare a structure for the list
  struct ListNode
  {
    double value;          // The value in this node
    struct ListNode *next;  // To point to the next node
  };
  ListNode *head;           // List head pointer
public:
  // Constructor
  NumberList()
    { head = NULL; }
  // Destructor
  ~NumberList();
  // Linked list operations
  void appendNode(double);
  void insertNode(double);
  void deleteNode(double);
  void displayList() const;
};
#endif
```

```
// Implementation file for the NumberList class
#include <iostream>  // For cout  and NULL
#include "NumberList.h"
using namespace std;

//************************************************
// appendNode appends a node containing the      *
// value pased into num, to the end of the list.  *
//************************************************

void NumberList::appendNode(double num)
{
  ListNode *newNode;  // To point to a new node
  ListNode *nodePtr;  // To move through the list

  // Allocate a new node and store num there.
  newNode = new ListNode;
  newNode->value = num;
  newNode->next = NULL;

  // If there are no nodes in the list
  // make newNode the first node.
  if (!head)
    head = newNode;
  else  // Otherwise, insert newNode at end.
  {
    // Initialize nodePtr to head of list.
    nodePtr = head;

    // Find the last node in the list.
    while (nodePtr->next)
      nodePtr = nodePtr->next;

    // Insert newNode as the last node.
    nodePtr->next = newNode;
  }
}

// displayList shows the value                  *
// stored in each node of the linked list       *
// pointed to by head.                          *
//************************************************

void NumberList::displayList() const
{
  ListNode *nodePtr;  // To move through the list

  // Position nodePtr at the head of the list.
  nodePtr = head;

  // While nodePtr points to a node, traverse
  // the list.
  while (nodePtr)
  {
    // Display the value in this node.
    cout << nodePtr->value << endl;

    // Move to the next node.
    nodePtr = nodePtr->next;
  }
}

//************************************************
// The insertNode function inserts a node with    *
// num copied to its value member.                *
//************************************************

void NumberList::insertNode(double num)
{
  ListNode *newNode;           // A new node
  ListNode *nodePtr;           // To traverse the list
  ListNode *previousNode = NULL; // The previous node

  // Allocate a new node and store num there.
  newNode = new ListNode;
  newNode->value = num:
```

```
//************************************************
// The insertNode function inserts a node with     *
// num copied to its value member.                 *
//************************************************

void NumberList::insertNode(double num)
{
  ListNode *newNode;           // A new node
  ListNode *nodePtr;           // To traverse the list
  ListNode *previousNode = NULL; // The previous node

  // Allocate a new node and store num there.
  newNode = new ListNode;
  newNode->value = num;

  // If there are no nodes in the list
  // make newNode the first node
  if (!head)
  {
    head = newNode;
    newNode->next = NULL;
  }
  else  // Otherwise, insert newNode
  {
    // Position nodePtr at the head of list.
    nodePtr = head;

    // Initialize previousNode to NULL.
    previousNode = NULL;

    // Skip all nodes whose value is less than num.
    while (nodePtr != NULL && nodePtr->value < num)
    {
      previousNode = nodePtr;
      nodePtr = nodePtr->next;
    }

    // link the previous node to the node after
    // nodePtr, then delete nodePtr.
    if (nodePtr)
    {
      previousNode->next = nodePtr->next;
      delete nodePtr;
    }
  }
}

//************************************************
// Destructor                                    *
// This function deletes every node in the list.  *
//************************************************

NumberList::~NumberList()
{
  ListNode *nodePtr;   // To traverse the list
  ListNode *nextNode;  // To point to the next node

  // Position nodePtr at the head of the list.
  nodePtr = head;

  // While nodePtr is not at the end of the list...
  while (nodePtr != NULL)
  {
    // Save a pointer to the next node.
    nextNode = nodePtr->next;

    // Delete the current node.
    delete nodePtr;

    // Position nodePtr at the next node.
    nodePtr = nextNode;
  }
}
```

Class Exercise: Append nodes to LinkedList

```
Russells-MacBook-Pro-2:Chapter 17 russell$ cat Pr17-1.cpp
// This program demonstrates a simple append
// operation on a linked list.
#include <iostream>
#include "NumberList.h"
using namespace std;

int main()
{
  // Define a NumberList object.
  NumberList list;

  // Append some values to the list.
  list.appendNode(2.5);
  list.appendNode(7.9);
  list.appendNode(12.6);
  list.displayList();
  return 0;
}
Russells-MacBook-Pro-2:Chapter 17 russell$ g++ Pr17-1.cpp
NumberList.cpp
Russells-MacBook-Pro-2:Chapter 17 russell$ ./a.out
2.5
7.9
12.6
```

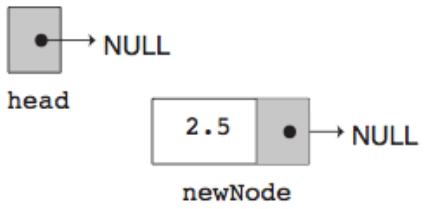
```
1  // This program demonstrates a simple append
2  // operation on a linked list.
3  #include <iostream>
4  #include "NumberList.h"
5  using namespace std;
6
7  int main()
8  {
9     // Define a NumberList object.
10    NumberList list;
11
12    // Append some values to the list.
13    list.appendNode(2.5);
14    list.appendNode(7.9);
15    list.appendNode(12.6);
16    return 0;
17 }
```

The first call to appendNode in line 13 passes 2.5 as the argument. In the following statements, a new node is allocated in memory, 2.5 is copied into its value member, and NULL is assigned to the node's next pointer:

```
newNode = new ListNode;
newNode->value = num;
newNode->next = NULL;
```

Figure 17-3 illustrates the state of the head pointer and the new node.

Figure 17-3

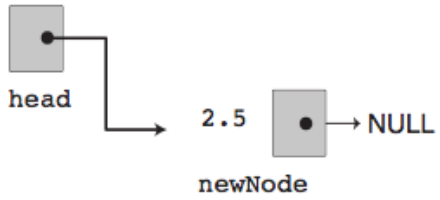


The next statement to execute is the following if statement:

```
if (!head)
   head = newNode;
```

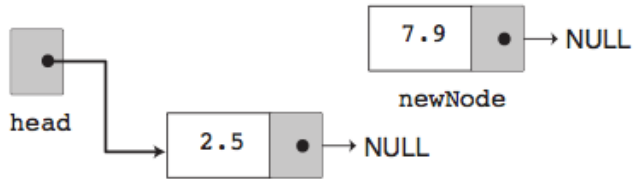
Because head points to NULL, the condition !head is true. The statement head = newNode; is executed, making newNode the first node in the list. This is illustrated in Figure 17-4.

Figure 17-4



There are no more statements to execute, so control returns to function main. In the second call to appendNode, in line 14, 7.9 is passed as the argument. Once again, the first three statements in the function create a new node, store the argument in the node's value member, and assign its next pointer to NULL. Figure 17-5 illustrates the current state of the list and the new node.

Figure 17-5



Because `head` no longer points to NULL, the `else` part of the `if` statement executes:

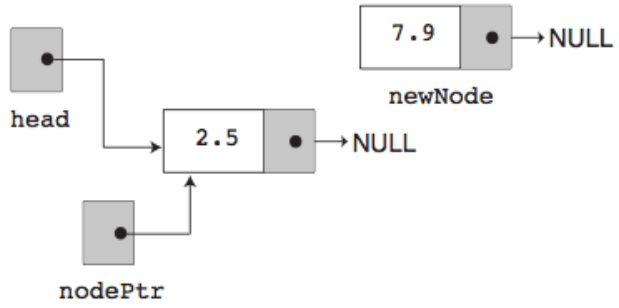
```
else   // Otherwise, insert newNode at end.
{
      // Initialize nodePtr to head of list.
      nodePtr = head;

      // Find the last node in the list.
      while (nodePtr->next)
         nodePtr = nodePtr->next;

      // Insert newNode as the last node.
      nodePtr->next = newNode;
}
```

The first statement in the `else` block assigns the value in `head` to `nodePtr`. This causes `nodePtr` to point to the same node that `head` points to. This is illustrated in Figure 17-6.

Figure 17-6



Look at the `next` member of the node that `nodePtr` points to. Its value is NULL, which means that `nodePtr->next` also points to NULL. `nodePtr` is already at the end of the list, so the `while` loop immediately terminates. The last statement, `nodePtr->next = newNode;` causes `nodePtr->next` to point to the new node. This inserts `newNode` at the end of the list as shown in Figure 17-7.

Traversing a Linked List

The `appendNode` function demonstrated in the previous section contains a `while` loop that traverses, or travels through the linked list. In this section we will demonstrate the `displayList` member function that traverses the list, displaying the `value` member of each node. The following pseudocode represents the algorithm.

```
Assign List head to node pointer.
While node pointer is not NULL
   Display the value member of the node pointed to by node pointer.
   Assign node pointer to its own next member.
End While.
```

The function is shown here:

```
45  void NumberList::displayList() const
46  {
47     ListNode *nodePtr;  // To move through the list
48
49     // Position nodePtr at the head of the list.
50     nodePtr = head;
51
52     // While nodePtr points to a node, traverse
53     // the list.
54     while (nodePtr)
55     {
56        // Display the value in this node.
57        cout << nodePtr->value << endl;
58
59        // Move to the next node.
60        nodePtr = nodePtr->next;
61     }
62  }
```

Program 17-4

```
1  // This program demonstrates the deleteNode member function.
2  #include <iostream>
3  #include "NumberList.h"
4  using namespace std;
5
6  int main()
7  {
8     // Define a NumberList object.
9     NumberList list;
10
11    // Build the list with some values.
12    list.appendNode(2.5);
13    list.appendNode(7.9);
14    list.appendNode(12.6);
15
16    // Display the list.
17    cout << "Here are the initial values:\n";
18    list.displayList();
19    cout << endl;
20
21    // Delete the middle node.
22    cout << "Now deleting the node in the middle.\n";
23    list.deleteNode(7.9);
24
25    // Display the list.
26    cout << "Here are the nodes left.\n";
27    list.displayList();
28    cout << endl;
29
30    // Delete the last node.
31    cout << "Now deleting the last node.\n";
32    list.deleteNode(12.6);
33
34    // Display the list.
35    cout << "Here are the nodes left.\n";
36    list.displayList();
37    cout << endl;
38
39    // Delete the only node left in the list.
40    cout << "Now deleting the only remaining node.\n";
41    list.deleteNode(2.5);
42
43    // Display the list.
44    cout << "Here are the nodes left.\n";
45    list.displayList();
46    return 0;
47 }
```

```
$ ./4out
Here are the initial values:
2.5
7.9
12.6

Now deleting the node in the middle.
Here are the nodes left.
2.5
12.6

Now deleting the last node.
Here are the nodes left.
2.5

Now deleting the only remaining node.
Here are the nodes left.
```

Recall, Project 2 due this Wednesday

Boost Libraries

Array [2]

Boost.Array is a wrapper for fixed-size arrays that enhances built-in arrays by supporting most of the STL container interface described in Section 22.1. Class `array` allows you to use fixed-size arrays in STL applications rather than `vectors` (dynamically sized arrays), which are not as efficient when there is no need for dynamic resizing. To use class `array` with compilers that support this C++0x feature, include the `<array>` header.

Bind [3]

Boost.Bind extends the functionality of the standard functions `std::bind1st` and `std::bind2nd`. The `bind1st` and `bind2nd` functions are used to adapt binary functions (i.e., functions that take two arguments) to be used with the standard algorithms which take unary functions (i.e., functions that take one argument). Class `bind` enhances that functionality by allowing you to adapt functions that take up to nine arguments. Class `bind` also makes it easy to reorder the arguments passed to the function using placeholders. To use class `bind` with compilers that support this C++0x feature, include the `<functional>` header.

Function [4]

Boost.Function allows you to store function pointers, member-function pointers and function objects in a function wrapper. A `function` can hold any function whose arguments and return type can be converted to match the signature of the function wrapper. For example, if the function wrapper was created to hold a function that takes a `string` and returns a `string`, it can also hold a function that takes a `char*` and returns a `char*`, because a `char*` can be converted to a `string`, using a conversion constructor. To use class `function` with compilers that support this C++0x feature, include the `<functional>` header.

Random [5]

Boost.Random allows you to create various random number generators and random number distributions. The `std::rand` and `std::srand` functions in the C++ Standard Library generate pseudo-random numbers. A **pseudo-random number generator** uses an initial state to produce seemingly random numbers—using the same initial state produces the same sequence of numbers. The `rand` function always uses the same initial state, therefore it produces the same sequence of numbers every time. The function `srand` allows you to set the initial state to vary the sequence. Pseudo-random numbers are often used in testing—the predictability enables you to confirm the results. `Boost.Random` provides pseudo-random number generators as well as generators that can produce **nondeterministic random numbers**—a set of random numbers that can't be predicted. Such random number generators are used in simulations and security scenarios where predictability is undesirable.

`Boost.Random` also allows you to specify the distribution of the numbers generated. A common distribution is the **uniform distribution**, which assigns the same probability to each number within a given range. This is similar to rolling a die or flipping a coin—each possible outcome is equally as likely. You can set this range at compile time. `Boost.Random`

---

2. Documentation for Boost.Array: `www.boost.org/doc/libs/1_45_0/doc/html/array.html`.
3. Documentation for Boost.Bind: `www.boost.org/doc/libs/1_45_0/libs/bind/bind.html`.
4. Documentation for Boost.Function: `www.boost.org/doc/libs/1_45_0/doc/html/function.html`.
5. Jens Maurer, "A Proposal to Add an Extensible Random Number Facility to the Standard Library," Document Number N1452, April 10, 2003, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1452.html`.

Regex [6]

Boost.Regex provides support for processing **regular expressions** in C++. Regular expressions are used to match specific character patterns in text. Many modern programming languages have built-in support for regular expressions, but C++ does not. With `Boost.Regex`, you can search for a particular expression in a `string`, replace parts of a `string` that match a regular expression, and split a `string` into tokens using regular expressions to define the delimiters. These techniques are commonly used for text processing, parsing and input validation. To use regular expressions with compilers that support this C++0x feature, include the `<regex>` header. We discuss some regular expression capabilities in more detail in Section 23.5.

Smart_ptr [7]

Boost.Smart_ptr defines smart pointers that help you manage dynamically allocated resources (e.g., memory, files and database connections). Programmers often get confused about when to deallocate memory or simply forget to do it, especially when the memory is referenced by more than one pointer. Smart pointers take care of these tasks automatically. TR1 includes several smart pointers from the `Boost.Smart_ptr` library. We discussed the `unique_ptr` class in Chapter 16. **shared_ptrs** handle lifetime management of dynamically allocated objects. The memory is released when there are no `shared_ptrs` referencing it. **weak_ptrs** allow you to observe the value held by a `shared_ptr` without assuming any management responsibilities. We discuss the `shared_ptr` and `weak_ptr` in more detail in Section 23.6. To use the smart pointer classes with compilers that support these C++0x features, thisinclude the `<regex>` header.

Tuple [8]

A **tuple** is a set of objects. **Boost.Tuple** allows you to create sets of objects in a generic way and allows generic functions to act on those sets. The library allows you to create tuples of up to 10 objects; that limit can be extended. Class `tuple` is basically an extension to the STL's `std::pair` class template. Tuples are often used to return multiple values from a function. They can also be used to store sets of elements in an STL container where each set of elements is an element of the container. Another useful feature is the ability to set the values of variables using the elements of a tuple. To use class `tuple` with compilers that support this C++0x feature, include the `<tuple>` header.

Type_traits [9]

The **Boost.Type_traits** library helps abstract the differences between types to allow generic programming implementations to be optimized. The `type_traits` classes allow you

Boost Libraries

Character class	Matches	Character class	Matches
`\d`	any decimal digit	`\D`	any non-digit
`\w`	any word character	`\W`	any non-word character
`\s`	any whitespace character	`\S`	any non-whitespace character

Installing the Boost Libraries

The Boost libraries can be used with minimal setup on many platforms and compilers. BoostPro Computing offers a free installer for using Boost with Visual Studio at `www.boostpro.com/download`. Most Linux distributions offer packages for Boost, though it is sometimes split up into separate packages for the headers and libraries. An installation guide available at `www.boost.org/more/getting_started/index.html` provides setup instructions for many compilers and platforms.

23.5 Regular Expressions with the regex Library

[*Note:* The C++0x library features used in this section's examples were not fully implemented in GNU C++ at the time of this writing. For now, if you wish to use these features in GNU C++, you can install the Boost version of the regular expressions library as discussed in Section 23.3]

Regular expressions are specially formatted `string`s that are used to find patterns in text. They can be used to validate data to ensure that it is in a particular format. For example, a zip code must consist of five digits, and a last name must start with a capital letter.

The `std::tr1::regex` library (from header `<regex>`) provides several classes and algorithms (in namespce `std::tr1`) for recognizing and manipulating regular expressions. Class template **`basic_regex`** represents a regular expression. The algorithm **`regex_match`** returns true if a `string` matches the regular expression. With `regex_match`, the entire string must match the regular expression. The `regex` library also provides the algorithm **`regex_search`**, which returns `true` if any part of an arbitrary `string` matches the regular expression.

```
$ g++ fig23_02.cpp
fig23_02.cpp:5:17: error: regex: No such file or directory
fig23_02.cpp: In function ‘int main()’:
fig23_02.cpp:11: error: ‘regex’ was not declared in this scope
fig23_02.cpp:11: error: expected `;' before ‘expression’
fig23_02.cpp:20: error: ‘smatch’ was not declared in this scope
fig23_02.cpp:20: error: expected `;' before ‘match’
fig23_02.cpp:23: error: ‘match’ was not declared in this scope
fig23_02.cpp:23: error: ‘expression’ was not declared in this scope
fig23_02.cpp:24: error: ‘regex_constants’ has not been declared
fig23_02.cpp:24: error: ‘regex_search’ was not declared in this scope
```

```
// Demonstrating regular expressions.
#include <iostream>
#include <string>
#include <regex>
using namespace std; // allows use of features in both std and std::tr1
int main()
{
  // create a regular expression
  regex expression( "J.*\\d[0-35-9]-\\d\\d-\\d\\d" );
  // create a string to be tested
  string string1 = "Jane's Birthday is 05-12-75\n"
    "Dave's Birthday is 11-04-68\n"
    "John's Birthday is 04-28-73\n"
    "Joe's Birthday is 12-17-77";
  // create a std::tr1::smatch object to hold the search results
  smatch match;

  // match regular expression to string and print out all matches
  while ( regex_search( string1, match, expression,
    regex_constants::match_not_eol ) )
  {
    cout << match.str() << endl; // print the matching string

    // remove the matched substring from the string
    string1 = match.suffix();
  } // end while
} // end of function main
```

```
#include <boost/regex.hpp>
#include <iostream>
#include <string>

int main()
{
    std::string text(" 192.168.0.1 abc 10.0.0.255 10.5.1 1.2.3.4a 5.4.3.2 ");
    const char* pattern =
        "\\b(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)"
        "\\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)"
        "\\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)"
        "\\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\b";
    boost::regex ip_regex(pattern);

    boost::sregex_iterator it(text.begin(), text.end(), ip_regex);
    boost::sregex_iterator end;
    for (; it != end; ++it) {
        std::cout << it->str() << "\n";
        // v.push_back(it->str()); or something similar
    }
}
```

Output:

```
192.168.0.1
10.0.0.255
5.4.3.2
```

The following example takes a C++ source file and builds up an index of class names, and the location of that class in the file.

```
#include <string>
#include <map>
#include <fstream>
#include <iostream>
#include <boost/regex.hpp>

using namespace std;

// purpose:
// takes the contents of a file in the form of a string
// and searches for all the C++ class definitions, storing
// their locations in a map of strings/int's

typedef std::map<std::string, std::string::difference_type, std::less<std::string> > map_type;

const char* re =
   // possibly leading whitespace:
   "^[[:space:]]*"
   // possible template declaration:
   "(template[[:space:]]*<[^;:{]+>[[:space:]]*)?"
   // class or struct:
   "(class|struct)[[:space:]]*"
   // leading declspec macros etc:
   "("
      "\\<\\w+\\>"
      "("
         "[[:blank:]]*\\([^)]*\\)"
      ")?"
      "[[:space:]]*"
   ")*"
   // the class name
   "(\\<\\w*\\>)[[:space:]]*"
   // template specialisation parameters
   "(<[^;:{]+>)?[[:space:]]*"
   // terminate in { or :
   "(\\{|:[^;\\{()]*\\{)";
```

```
boost::regex expression(re);
map_type class_index;

bool regex_callback(const boost::match_results<std::string::const_iterator>& what)
{
   // what[0] contains the whole string
   // what[5] contains the class name.
   // what[6] contains the template specialisation if any.
   // add class name and position to map:
   class_index[what[5].str() + what[6].str()] = what.position(5);
   return true;
}

void load_file(std::string& s, std::istream& is)
{
   s.erase();
   s.reserve(is.rdbuf()->in_avail());
   char c;
   while(is.get(c))
   {
      if(s.capacity() == s.size())
         s.reserve(s.capacity() * 3);
      s.append(1, c);
   }
}

int main(int argc, const char** argv)
{
   std::string text;
   for(int i = 1; i < argc; ++i)
   {
      cout << "Processing file " << argv[i] << endl;
      std::ifstream fs(argv[i]);
      load_file(text, fs);
      // construct our iterators:
      boost::sregex_iterator m1(text.begin(), text.end(), expression);
      boost::sregex_iterator m2;
      std::for_each(m1, m2, &regex_callback);
      // copy results:
      cout << class_index.size() << " matches found" << endl;
      map_type::iterator c, d;
      c = class_index.begin();
      d = class_index.end();
      while(c != d)
      {
         cout << "class \"" << (*c).first << "\" found at index: " << (*c).second << endl;
         ++c;
      }
      class_index.erase(class_index.begin(), class_index.end());
   }
   return 0;
}
```

Validating User Input with Regex & Boost

```
// Validating user input with regular expressions.
#include <iostream>
#include <string>
#include <regex>
using namespace std;

bool validate( const string&, const string& ); // validate prototype
string inputData( const string&, const string& ); // inputData prototype

int main()
{
   // enter the last name
   string lastName = inputData( "last name", "[A-Z][a-zA-Z]*" );

   // enter the first name
   string firstName = inputData( "first name", "[A-Z][a-zA-Z]*" );

   // enter the address
   string address = inputData( "address",
      "[0-9]+\\s+([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" );

   // enter the city
   string city =
      inputData( "city", "([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" );

   // enter the state
   string state = inputData( "state",
      "([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" );

   // enter the zip code
   string zipCode = inputData( "zip code", "\\d{5}" );

   // enter the phone number
   string phoneNumber = inputData( "phone number",
      "[1-9]\\d{2}-[1-9]\\d{2}-\\d{4}" );

   // display the validated data
   cout << "\nValidated Data\n\n"
      << "Last name: " << lastName << endl
      << "First name: " << firstName << endl
      << "Address: " << address << endl
      << "City: " << city << endl
      << "State: " << state << endl
      << "Zip code: " << zipCode << endl
      << "Phone number: " << phoneNumber << endl;
} // end of function main
```

```
// validate the data format using a regular expression
bool validate( const string &data, const string &expression )
{
   // create a regex to validate the data
   regex validationExpression = regex( expression );
   return regex_match( data, validationExpression );
} // end of function validate

// collect input from the user
string inputData( const string &fieldName, const string &expression )
{
   string data; // store the data collected

   // request the data from the user
   cout << "Enter " << fieldName << ": ";
   getline( cin, data );

   // validate the data
   while ( !( validate( data, expression ) ) )
   {
      cout << "Invalid " << fieldName << ".\n";
      cout << "Enter " << fieldName << ": ";
      getline( cin, data );
   } // end while

   return data;
} // end of function inputData
```

23.6 Smart Pointers

[*Note:* The C++0x library features used in this section's examples work in both Microsoft Visual C++ 2010 Express and GNU C++ 4.5. GNU C++ considers these features experimental and requires you to use the command line option `-std:c++0x` to compile the examples correctly.]

Many common bugs in C and C++ code are related to pointers. **Smart pointers** help you avoid errors by providing additional functionality to standard pointers. This functionality typically strengthens the process of memory allocation and deallocation. Smart pointers also help you write exception safe code. If a program throws an exception before `delete` has been called on a pointer, it creates a memory leak. After an exception is thrown, a smart pointer's destructor will still be called, which calls `delete` on the pointer for you.

Section 16.11 showed one of the smart pointer classes—`unique_ptr`—which is responsible for managing dynamically allocated memory. A `unique_ptr` automatically calls `delete` to free its associated dynamic memory when the `unique_ptr` is destroyed or goes out of scope. A `unique_ptr` is a basic smart pointer. C++0x provides other smart pointer options with additional functionality.

23.6.1 Reference Counted shared_ptr

shared_ptrs (from header <memory>) hold an internal pointer to a resource (e.g., a dynamically allocated object) that may be shared with other objects in the program. You can have any number of shared_ptrs to the same resource. shared_ptrs really do share the resource—if you change the resource with one shared_ptr, the changes also will be "seen" by the other shared_ptrs. The internal pointer is deleted once the last shared_ptr to the resource is destroyed. shared_ptrs use **reference counting** to determine how many shared_ptrs point to the resource. Each time a new shared_ptr to the resource is created, the **reference count** increases, and each time one is destroyed, the reference count decreases. When the reference count reaches zero, the internal pointer is deleted and the memory is released.

shared_ptrs are useful in situations where multiple pointers to the same resource are needed, such as in STL containers. shared_ptrs can safely be copied and used in STL containers.

shared_ptrs also allow you to determine how the resource will be destroyed. For most dynamically allocated objects, delete is used. However, some resources require more complex cleanup. In that case, you can supply a custom **deleter** function, or function object, to the shared_ptr constructor. The deleter determines how to destroy the resource. When the reference count reaches zero and the resource is ready to be destroyed, the shared_ptr calls the custom deleter function. This functionality enables a shared_ptr to manage almost any kind of resource.

C++0x/C++11 Support in GCC

C++0x was the working name of the ISO C++ 2011 standard, which introduced a host of new features into the standard C++ language and library. This project sought to implement new C++11 features in GCC, and made it the first compiler to bring feature-complete C++11 to C++ programmers.

C++11 features are available as part of the "mainline" GCC compiler in the trunk of GCC's Subversion repository and in GCC 4.3 and later. To enable C++0x support, add the command-line parameter `-std=c++0x` to your `g++` command line. Or, to enable GNU extensions in addition to C++0x extensions, add `-std=gnu++0x` to your `g++` command line. GCC 4.7 and later support `-std=c++11` and `-std=gnu++11` as well.

Important: GCC's support for C++11 is still **experimental**. Some features were implemented based on early proposals, and no attempt will be made to maintain backward compatibility when they are updated to match the final C++11 standard.

C++11 Language Features

The following table lists new language features that have been accepted into the C++11 standard. The "Proposal" column provides a link to the ISO C++ committee proposal that describes the feature, while the "Available in GCC?" column indicates the first version of GCC that contains an implementation of this feature (if it has been implemented).

For information about C++11 support in a specific version of GCC, please see:

- GCC 4.3 C++0x Status
- GCC 4.4 C++0x Status
- GCC 4.5 C++0x Status
- GCC 4.6 C++0x Status
- GCC 4.7 C++11 Status
- GCC 4.8 C++11 Status

Language Feature	Proposal	Available in GCC?
Rvalue references	N2118	GCC 4.3
Rvalue references for `*this`	N2439	GCC 4.8.1
Initialization of class objects by rvalues	N1610	Yes

Example Using **`shared_ptr`**

Figures 23.6–23.7 define a simple class to represent a `Book` with a `string` to represent the title of the `Book`. The destructor for class `Book` (Fig. 23.7, lines 12–15) displays a message on the screen indicating that an instance is being destroyed. We use this class to demonstrate the common functionality of `shared_ptr`.

```
1  // Fig. 23.6: Book.h
2  // Declaration of class Book.
3  #ifndef BOOK_H
4  #define BOOK_H
5  #include <string>
6  using namespace std;
7
8  class Book
9  {
10 public:
11    Book( const string &bookTitle ); // constructor
12    ~Book(); // destructor
13    string title; // title of the Book
14 };
15 #endif // BOOK_H
```

Fig. 23.6 | Book header.

```
1  // Fig. 23.7: Book.cpp
2  // Member-function definitions for class Book.
3  #include <iostream>
4  #include <string>
5  #include "Book.h"
6  using namespace std;
7
8  Book::Book( const string &bookTitle ) : title( bookTitle )
9  {
10 }
11
12 Book::~Book()
13 {
14    cout << "Destroying Book: " << title << endl;
15 } // end of destructor
```

Fig. 23.7 | Book member-function definitions.

Book.h, Book.cpp, and SharedPtrExample.cpp are in the Google Drive

Creating `shared_ptr`*s*

The program in Fig. 23.8 uses `shared_ptr`s (from the header `<memory>`) to manage several instances of class `Book`. We also create a `typedef`, `BookPtr`, as an alias for the type `shared_ptr<Book>` (line 10). Line 28 creates a `shared_ptr` to a `Book` titled `"C++ How to Program"` (using the `BookPtr` `typedef`). The `shared_ptr` constructor takes as its argument a pointer to an object. We pass it the pointer returned from the `new` operator. This creates a `shared_ptr` that manages the `Book` object and sets the reference count to one. The constructor can also take another `shared_ptr`, in which case it shares ownership of the resource with the other `shared_ptr` and the reference count is increased by one. The first `shared_ptr` to a resource should always be created using the `new` operator. A `shared_ptr` created with a regular pointer assumes it's the first `shared_ptr` assigned to that resource and starts the reference count at one. If you make multiple `shared_ptr`s with the same pointer, the `shared_ptr`s won't acknowledge each other and the reference count will be wrong. When the `shared_ptr`s are destroyed, they both call `delete` on the resource.

A smart pointer is a class that wraps a "bare" C++ pointer, to manage the lifetime of the object being pointed to.

With "bare" C++ pointers, the programmer has to explicitly destroy the object when it is no longer useful.

```
// Need to create the object to achieve some goal
MyObject* ptr = new MyObject();
ptr->DoSomething();// Use the object in some way.
delete ptr; // Destroy the object. Done with it.
// Wait, what if DoSomething() raises an exception....
```

A smart pointer by comparison defines a policy as to when the object is destroyed. You still have to create the object, but you no longer have to worry about destroying it.

```
SomeSmartPtr<MyObject> ptr(new MyObject());
ptr->DoSomething(); // Use the object in some way.

// Destruction of the object happens, depending
// on the policy the smart pointer class uses.

// Destruction would happen even if DoSomething()
// raises an exception
```

Note that scoped_ptr instances cannot be copied. This prevents the pointer from being deleted multiple times (incorrectly). You can however pass references to it around to other functions you call.

Scoped pointers are useful when you want to tie the lifetime of the object to a particular block of code, or if you embedded it as member data inside another object, the lifetime of that other object. The object exists until the containing block of code is exitted, or until the containing object is itself destroyed.

A more complex smart pointer policy involves reference counting the pointer. This does allow the pointer to be copied. When the last "reference" to the object is destroyed, the object is deleted. This policy is implemented by `boost::shared_ptr` and `std::tr1:shared_ptr`.

```
void f()
{
    typedef std::tr1::shared_ptr<MyObject> MyObjectPtr; // Nice short alias.
    MyObjectPtr p1; // Empty
    {
        MyObjectPtr p2(new MyObject());
        // There is now one "reference" to the created object
        p1=p2; // Copy the pointer.
        // There is are now two references to the object.
    } // p2 is destroyed, leaving one reference to the object.
} // p1 is destroyed, leaving a reference count of zero.
  // The object is deleted.
```

Reference counted pointers are very useful when the lifetime of your object is much more complicated, and is not tied directly to a particular section of code or to another object.

Function objects in the STL

Many STL algorithms allow you to pass a function pointer into the algorithm to help the algorithm perform its task. For example, the `binary_search` algorithm that we discussed in Section 22.8.6 is overloaded with a version that requires as its fourth parameter a pointer to a function that takes two arguments and returns a `bool` value. The `binary_search` algorithm uses this function to compare the search key to an element in the collection. The function returns `true` if the search key and element being compared are equal; otherwise, the function returns `false`. This enables `binary_search` to search a collection of elements for which the element type does not provide an overloaded equality == operator.

The STL's designers made the algorithms more flexible by allowing any algorithm that can receive a function pointer to receive an object of a class that overloads the parentheses operator with a function named `operator()`, provided that the overloaded operator meets the requirements of the algorithm—in the case of `binary_search`, it must receive two arguments and return a `bool`. An object of such a class is known as a **function object** and can be used syntactically and semantically like a function or function pointer—the overloaded parentheses operator is invoked by using a function object's name followed by parentheses containing the arguments to the function. Together, function objects and functions are known as **functors**. Most algorithms can use function objects and functions interchangeably.

Function objects provide several advantages over function pointers. Since function objects are commonly implemented as class templates that are included into each source code file that uses them, the compiler can inline an overloaded `operator()` to improve performance. Also, since they're objects of classes, function objects can have data members that `operator()` can use to perform its task.

STL function objects	Type	STL function objects	Type
`divides< T >`	arithmetic	`logical_or< T >`	logical
`equal_to< T >`	relational	`minus< T >`	arithmetic
`greater< T >`	relational	`modulus< T >`	arithmetic
`greater_equal< T >`	relational	`negate< T >`	arithmetic
`less< T >`	relational	`not_equal_to< T >`	relational
`less_equal< T >`	relational	`plus< T >`	arithmetic
`logical_and< T >`	logical	`multiplies< T >`	arithmetic
`logical_not< T >`	logical		

```cpp
// Fig. 22.42: Fig22_42.cpp
// Demonstrating function objects.
#include <iostream>
#include <vector> // vector class-template definition
#include <algorithm> // copy algorithm
#include <numeric> // accumulate algorithm
#include <functional> // binary_function definition
#include <iterator> // ostream_iterator
using namespace std;

// binary function adds square of its second argument and the
// running total in its first argument, then returns the sum
int sumSquares( int total, int value )
{
   return total + value * value;
} // end function sumSquares

// binary function class template defines overloaded operator()
// that adds the square of its second argument and running
// total in its first argument, then returns sum
template< typename T >
class SumSquaresClass : public binary_function< T, T, T >
{
public:
   // add square of value to total and return result
   T operator()( const T &total, const T &value )
   {
      return total + value * value;
   } // end function operator()
}; // end class SumSquaresClass
```

Ende