

764

How to: Understand and Use Tapestry Classes

G

G.1 A Library of Useful Classes

This book supplies many classes for you to use in programming and exploring computer science. These classes extend what's available in the base C++ language by supplying off-the-shelf components that you can use to solve more problems than if you had to design and implement the classes from scratch. If someone tells you that you're not really using C++ if you use these supplied classes because the classes are not part of the C++ language, these people are narrow-minded and without a clue as to how people write software today. It may be prudent not to tell them this. The classes introduced in this book have been designed to be powerful but simple so that they are easy for beginning programmers to use. This means the classes may not be as powerful as similar classes that are designed to serve a larger audience of professional programmers. However, the classes are designed to be understandable by novice programmers while still being powerful enough to be used in large, real programs. Sometimes an industrial-strength class that covers 95% of all applications is not as powerful as a class that covers 65% of all applications if the industrial-strength class is much harder to learn and use. The Tapestry classes are used by people programming for a living and programming for fun. Sometimes this is the same group of people.

G.1.1 Summary of Classes and Functions

I refer to the core classes and function libraries introduced in this book as **libtapestry**. The easiest way to use these classes is to create a library which is then linked automatically with every program you write. With Unix you do this with a makefile, with Windows or Macs you do this with a project as part of an IDE. Information on creating libraries is available in How to I. Only the nontemplated classes and functions are part of the library.



There are many other programs used in the book, but the core classes and functions are summarized in Table G.1. The header files for most of these classes are reproduced in the following sections as documentation for each class.

Table G.1 The classes and function libraries introduced in this book that make up `libtapestry`.

Class	Header File	Description
BigInt	<i>bigint.h</i>	unbounded integers
CList	<i>clist.h</i>	immutable lists
ClockTime	<i>clockt.h</i>	clock times (e.g., 13:24:09)
CTimer	<i>ctimer.h</i>	stopwatch timing for code
Date	<i>date.h</i>	dates (e.g., July 16, 2007)
Dice	<i>dice.h</i>	simulate N -sided dice
DirStream and DirEntry	<i>directory.h</i>	access directories
Permuter	<i>permuter.h</i>	permutes int vectors
Point	<i>point.h</i>	two-dimensional points
RandGen	<i>randgen.h</i>	random numbers
SimpleMap	<i>simplemap.h</i>	rudimentary map class
StringSet	<i>stringset.h</i>	sets of strings
tvector	<i>tvector.h</i>	range-checked vector class
tmatrix	<i>tmatrix.h</i>	range-checked 2D matrix
WordStreamIterator	<i>worditer.h</i>	reading files of words
Free Functions	Header File	Description
deg2rad, PI, ...	<i>mathutils.h</i>	math utilities
PromptRange, ...	<i>prompt.h</i>	prompt for values
QuickSort, bsearch, ...	<i>sortall.h</i>	sorting and searching
ToLower, atoi, ...	<i>strutils.h</i>	string functions
WaitForReturn	<i>utils.h</i>	wait for user to press return

The classes `CList`, `tvector`, `tmatrix`, and `SimpleMap` are templated as are the functions in *sortall.h*. The classes in *directory.h* are implemented differently for Unix and Windows platforms. All other classes should be platform independent, although it is possible there are some differences I have not encountered.

G.1.2 Implementations of Tapestry Classes

I have designed the classes and functions in Table G.1 to be used from the beginning of an introductory course though some stress topics not typically covered in the first weeks, such as vectors, matrices, and maps. Although the classes are designed to be used by client programs, most of them can be studied as examples of class design. However, some implementations depend on topics not covered in this book, or rely on platform-specific libraries that aren't of general interest. These include:

- Classes in *directory.cpp* use low-level operating-system specific functions.
- Classes in *tvector.h* allocate built-in arrays using `operator new []`, not covered in this text.

- Classes in *date.cpp*, *randgen.cpp*, and *clockt.cpp* use C-functions for accessing time to determine the current time of the day or the current day of the week.

All other classes have been documented so that their implementations can be studied.

G.2 Header Files for Tapestry Classes

G.2.1 Prompting Functions in `prompt.h`

Each prompting function comes in two forms, one using operator `>>` for input, the other using `getline`. For example, functions `PromptRange` and `PromptLnRange` both request integer input in a specific range though the latter reads an entire line of text while the former reads only the first string. All the functions read strings and convert to the type requested, such as `int` or `double`.

Program G.1 `prompt.h`

```
#ifndef _PROMPT_H
#define _PROMPT_H

#include <string>
using namespace std;

// facilitates prompting for int, double or string
//
// each function has a PromptLnXXX equivalent that reads a line of
// text
//
// PromptRange: used for int or double entry
//
// int PromptRange(const string & prompt,int low, int high)
//             - returns int in range [low..high]
// Example:
// int x = PromptRange("enter weekday",1,7);
//
// generates prompt: enter weekday between 1 and 7
//
// double PromptRange(const string & prompt,double low, double high)
//             - returns int in range [low..high]
// Example:
// double d = PromptRange("enter value",0.5,1.5);
//
// generates prompt: enter value between 0.5 and 1.5
//
// const string & promptString(const string & prompt)
//             - returns a string
// Example:
// string filename = PromptString("enter file name");
//
// bool PromptYesNo(const string & prompt)
```

768

Appendix G How to: Understand and Use Tapestry Classes

```

//                                     - returns true iff user enter yes
// (or any string beginning with y, only strings beginning with y or
// n are accepted)
//
// Example:
//   if (PromptYesNo("continue?"))
//       DoStuff();
//   else
//       Quit();

long int PromptRange(const string & prompt, long int low, long int high);
// precondition: low <= high
// postcondition: returns a value between low and high (inclusive)

long int PromptlnRange(const string & prompt, long int low, long int high);
// precondition: low <= high
// postcondition: returns a value between low and high (inclusive)
//                   reads an entire line

int PromptRange(const string & prompt, int low, int high);
// precondition: low <= high
// postcondition: returns a value between low and high (inclusive)

int PromptlnRange(const string & prompt, int low, int high);
// precondition: low <= high
// postcondition: returns a value between low and high (inclusive)
//                   reads an entire line

double PromptRange(const string & prompt, double low, double high);
// precondition: low <= high
// postcondition: returns a value between low and high (inclusive)

double PromptlnRange(const string & prompt, double low, double high);
// precondition: low <= high
// postcondition: returns a value between low and high (inclusive)
//                   reads an entire line

string PromptString(const string & prompt);
// postcondition: returns string entered by user

string PromptlnString(const string & prompt);
// postcondition: returns string entered by user, reads entire line

bool PromptYesNo(const string & prompt);
// postcondition: returns true iff user enters "yes" (any string with
//                   'y' as first letter, only 'y' and 'n' strings accepted)

bool PromptlnYesNo(const string & prompt);
// postcondition: returns true iff user enters "yes" (any string with
//                   'y' as first letter, only 'y' and 'n' strings accepted)
//                   reads entire line
#endif

```

prompt.h

G.2.2 The Class Date

Program G.2 date.h

```
#ifndef _DATE_H
#define _DATE_H

/*****
This code is freely distributable and modifiable providing you
leave this notice in it.
Copyright @ Owen Astrachan
*****/
#include <iostream>
#include <string>
using namespace std;

// a class for manipulating dates
//
// Date class represents a date in the Gregorian calendar
// works only for dates after October, 1752
//
// attempts to construct invalid dates, e.g., 15 month,
// or 38th day result in month == 1, day == 1. years aren't checked
// for validity
//
// Date() - construct default date (today)
// Date(long days) - construct date given absolute # of days from
// 1 A.D., e.g., 710,347 = November 12, 1945
//
// Date(int m,int d,int y) - constructor requires three parameters:
// month, day, year, e.g.,
// Date d(4,8,1956); initializes d to represent
// the date April 8, 1956. Full year is required
//
// int Month() - return, respectively, month, day, and year
// int Day() corresponding to date with 1 = january,
// int Year() 2 = february, ... 12 = december
//
// string DayName() - return string corresponding to day of week
// either "Monday", "Tuesday", ... "Sunday"
// string MonthName() - return string corresponding to month
// either "January", "February",..."December"
//
// int DaysIn() - return number of days in month
//
// long Absolute() - returns absolute # of date assuming
// that Jan 1, 1 AD is day 1. Has property
// that Absolute() % 7 = k, where k = 0 is sunday
// k = 1 is monday, ... k = 6 is saturday
```

770

Appendix G How to: Understand and Use Tapestry Classes

```

//
// string ToString() - returns string version of date, e.g.,
//                   - d.SetDate(11,23,1963); then d.ToString()
//                   returns string "November 23 1963"
// *****
// arithmetic operators for dates
// *****
//
// dates support some addition and subtraction operations
//
// Date d(1,1,1960); // 1960 is a leap year
// d++; // d represents January 2, 1960
// d--; // d is back to January 1, 1960
// d += 31; // d is February 1, 1960
// d -= 32; // d is December 31, 1959
// Date d2 = d + 1; // d2 is January 1, 1960
// Date d3 = 365 + d2; // d3 is December 31, 1961
// Date d4 = d - 1; // d4 is December 30, 1959
//
// *****
class Date
{
public:
    // constructors
    Date(); // construct date with default value
    Date(long days); // construct date from absolute #
    Date(int m,int d,int y); // construct date with specified values

    // accessor functions

    int Month() const; // return month corresponding to date
    int Day() const; // return day corresponding to date
    int Year() const; // return year corresponding to date
    int DaysIn() const; // return # of days in month
    string DayName() const; // "monday", "tuesday", ... or "sunday"
    string MonthName() const; // "january","february",... or "december"
    long Absolute() const; // number of days since 1 A.D. for date
    string ToString() const; // returns string for date in ascii

    bool Equal(const Date & rhs) const; // for implementing <, >, etc
    bool Less(const Date & rhs) const;

    // mutator functions

    Date operator ++(int); // add one day, postfix operator
    Date operator --(int); // subtract one day, postfix operator
    Date& operator +=(long dx); // add dx, e.g., jan 1 + 31 = feb 1
    Date& operator -=(long dx); // subtract dx, e.g., jan 1 - 1 = dec 31

private:
    int myDay; // day of week, 0-6
    int myMonth; // month, 0-11
    int myYear; // year in four digits, e.g., 1899

```

G.2 Header Files for Tapestry Classes

771

```

    void CheckDate(int m, int d, int y); // make sure that date is valid
};

Date operator + (const Date & d, long dx); // add dx to date d
Date operator + (long dx, const Date & d); // add dx to date d
Date operator - (const Date & d, long dx); // subtract dx from date d
long operator - (const Date & lhs, const Date & rhs);

ostream & operator << (ostream & os, const Date & d);
bool operator == (const Date & lhs, const Date & rhs);
bool operator != (const Date & lhs, const Date & rhs);
bool operator < (const Date & lhs, const Date & rhs);
bool operator > (const Date & lhs, const Date & rhs);
bool operator <= (const Date & lhs, const Date & rhs);
bool operator >= (const Date & lhs, const Date & rhs);

#endif

```

date.h

G.2.3 The Class Dice

Changes from the first-edition code include making accessor functions `const` and moving the random number generator from *dice.h* to *dice.cpp*.

Program G.3 dice.h

```

#ifndef _DICE_H
#define _DICE_H

// class for simulating a die (object "rolled" to generate
// a random number)
//
// Dice(int sides) - constructor, sides specifies number of "sides"
// for the die, e.g., 2 is a coin, 6 is a 'regular' die
//
// int Roll() - returns the random "roll" of the die, a uniformly
// distributed random number between 1 and # sides
//
// int NumSides() - access function, returns # of sides
//
// int NumRolls() - access function, returns # of times Roll called
// for an instance of the class

class Dice
{
public:
    Dice(int sides); // constructor
    int Roll(); // return the random roll
    int NumSides() const; // how many sides this die has
    int NumRolls() const; // # times this die rolled

private:

```

772

Appendix G How to: Understand and Use Tapestry Classes

```

    int myRollCount;          // # times die rolled
    int mySides;             // # sides on die
};

#endif /* _DICE_H not defined */

```

dice.h

G.2.4 The Class RandGen

Program G.4 randgen.h

```

#ifndef _RANDGEN_H
#define _RANDGEN_H

#include <limits.h>                // for INT_MAX

// designed for implementation independent randomization
// if all system dependent calls included in this class, then
// other classes can make use of this class in independent manner
// all random numbers are uniformly distributed in given range
//
// RandGen() - constructor sets seed of random # generator
//             once per program, not per class/object
//
// RandInt(int max)
// RandInt(int low,int max) - return random integer in range [0..max)
//                           when one parameter used, [low..max] when
//                           two parameters used
//
// examples:  rnd.RandInt(6) is random integer [0..5] or [0..6)
//             rnd.RandInt(3,10) is random integer [3..10]
//             rnd.RandInt() is random integer [0..INT_MAX)
//
// RandReal() - returns random double in range [0..1)
// RandReal(double low, double max) - random double in range [low..max)

class RandGen
{
public:
    RandGen();                    // set seed for all instances
    int RandInt(int max = INT_MAX); // returns int in [0..max)
    int RandInt(int low, int max); // returns int in [low..max]
    double RandReal();           // returns double in [0..1)
    double RandReal(double low, double max); // range [low..max)

    static void SetSeed(int seed); // static (per class) seed set
private:
    static int ourInitialized;     // for 'per-class' initialization
};

#endif

```

randgen.h

G.2.5 The Class CTimer

Program G.5 ctimer.h

```

#ifndef _CTIMER_H
#define _CTIMER_H

// a class that can be used to "time" parts of programs
// or as a general timer
//
// operations are:
//
//   Start() : starts the timer
//   Stop()  : stops the timer
//   ElapsedTime() : returns the elapsed time between
//                 start and the last stop
//   CumulativeTime() : returns cumulative total of all
//                 "laps" (timed intervals), i.e., sum of
//                 calls to ElapsedTime
//   Reset()  : resets cumulative time to 0
//                 so "removes" history of timer
//
//
class CTimer{
public:
    CTimer();                // constructor
    void Reset();           // reset timer to 0
    void Start();          // begin timing
    void Stop();           // stop timing
    double ElapsedTime();  // between last start/stop
    double CumulativeTime(); // total of all times since reset
private:
    long myStartTime,myEndTime;
    double myElapsed;        // time since start and last stop
    double myCumulative;     // cumulative of all "lap" times
};

#endif // _CTIMER_H not defined

```

ctimer.h

G.2.6 The Class wordStreamIterator

Changes from the first edition include renaming the iterator functions.

Program G.6 worditer.h

```

#ifndef _WORDSTREAMITERATOR_H
#define _WORDSTREAMITERATOR_H

```

774

Appendix G How to: Understand and Use Tapestry Classes

```

// Owen Astrachan 7/3/95, modified 4/9/99
//
// class WordStreamIterator
//
// void Open(string name)
//     - initializes iterator to file specified by name
//
// void Init(), void Next(), bool HasMore()
//     - "standard" iterating functions (see below)
//
// usage: call Init(), before accessing Current()
//     call Next() to move to the next word in the stream
//     call Current() to access the current word
//     call HasMore() to determine if Current() is valid
//
// string Current()
//     - returns current string (see below)
//
// WordStreamIterator iter;
// iter.Open("testfile.dat");
// for(iter.Init(); iter.HasMore(); iter.Next())
//     cout << iter.Current() << endl;
//
//
#include <string>
#include <fstream>
using namespace std;

class WordStreamIterator
{
public:
    WordStreamIterator();
    void Open(const string & name); // bind stream to specific text file
    void Init(); // initialize iterator
    string Current(); // returns current word
    bool HasMore(); // true if more words
    void Next(); // advance to next word

private:
    string myWord; // the current word
    bool myMore; // true if more words
    ifstream myInput; // the stream to read from
};

#endif

```

worditer.h

G.2.7 The Class StringSet

Program G.7 stringset.h

```
#ifndef _STRINGSET_H
#define _STRINGSET_H

#include <string>
#include "tvector.h"
using namespace std;

class StringSet
{
public:
    StringSet();
    StringSet(int isize); // initialize size - for efficiency

    // accessors
    bool contains(const string& s) const;
    int size() const;

    // mutators
    void insert(const string& s);
    void erase (const string& s);
    void clear();

    friend class StringSetIterator;
private:
    int myCount; // # of entries stored in myList
    tvector<string> myList; // storage for each string

    int search(const string & key) const; // returns index in myList of key
};

class StringSetIterator
{
public:
    StringSetIterator(const StringSet& s);

    void Init() const;
    bool HasMore() const;
    void Next() const;
    string Current() const;
private:
    const StringSet& mySet;
    mutable int myIndex;
};

#endif
```

stringset.h

776

Appendix G How to: Understand and Use Tapestry Classes

G.2.8 The String Functions in `strutils.h`

Program G.8 `strutils.h`

```
#ifndef _STRUTILS_H
#define _STRUTILS_H

#include <iostream>
#include <string>
using namespace std;

void ToLower(string & s);
// postcondition: all alphabetic characters in s changed to lower case
//                (only upper case letters changed)

void ToUpper(string & s);
// postcondition: all alphabetic characters in s changed to upper case
//                (only upper case letters changed)

void StripPunc(string & s);
// postcondition: s has no leading/trailing punctuation

void StripWhite(string & s);
// postcondition: s has no leading/trailing white space

string LowerString(const string & s);
// postcondition: return lowercase equivalent of s

string UpperString(const string & s);
// postcondition: return uppercase equivalent of s

int atoi(const string & s);           // returns int equivalent
double atof(const string & s);       // returns double equivalent
string itoa(int n);                  // returns string equivalent
string tostring(int n);              // like itoa, convert int to string
string tostring(double d);           // convert double to string

#endif
```

`strutils.h`

G.2.9 The Math Helper Functions in `mathutils.h`

Program G.9 `mathutils.h`

```
#ifndef _MATHUTIL_H
#define _MATHUTIL_H

bool FloatEqual(double lhs, double rhs);
```

```
// post: returns true iff lhs == rhs
//       where == is determined by using relative error, i.e.
//       |lhs-rhs| / min(|lhs|,|rhs|)

// convert degrees to radians and vice-versa

double deg2rad(double deg);
double rad2deg(double rad);

const double PI = 3.1415926535897;

// returns smaller of lhs and rhs, operator < must
// be overloaded for the type T

template <class T>
T min(const T& lhs, const T& rhs)
{
    return lhs < rhs ? lhs : rhs;
}

// returns larger of lhs and rhs, operator < must
// be overloaded for the type T

template <class T>
T max(const T& lhs, const T& rhs)
{
    return lhs < rhs ? rhs : lhs;
}

#endif
```

mathutils.h

G.2.10 The struct Point

Program G.10 point.h

```
#ifndef _POINT_H
#define _POINT_H

#include <string>
using namespace std;

struct Point
{
    Point();
    Point(double px, double py);

    string toString() const;
    double distanceFrom(const Point& p) const;
    double x;
    double y;
};
```

778

Appendix G How to: Understand and Use Tapestry Classes

```
};

bool operator == (const Point& lhs, const Point& rhs);
bool operator != (const Point& lhs, const Point& rhs);
bool operator < (const Point& lhs, const Point& rhs);
bool operator > (const Point& lhs, const Point& rhs);
bool operator <= (const Point& lhs, const Point& rhs);
bool operator >= (const Point& lhs, const Point& rhs);
```

```
ostream& operator << (ostream& os, const Point& p);
```

```
#endif
```

`point.h`

G.2.11 The Classes in `directory.h`

Program G.11 `directory.h`

```
#ifndef _DIRECTORY_H
#define _DIRECTORY_H

//
// author: Owen Astrachan
// date: 9/21/93
//
// modified 11/28/94
// modified 4/5/95
// modified 1/18/96
// modified 5/10/99, ported to 32-bit windows
//
// classes for manipulating directories
// provide a standard interface for directory
// queries from C++ programs that can, in theory, be implemented
// on several platforms
//
// currently supported: Unix, DOS, Windows
//
// the class DirEntry provides directory information
// accessible via methods Name, Size, and IsDir
//
// the class DirStream does I/O on directories
// it supports "standard" (for the Tapestry book)
// iterator methods/member functions
//

// ***** DirEntry member functions:
//
// string Name() - returns name of file
// int Size() - returns size of file (in bytes)
// bool IsDir() - returns false if NOT directory, else true
```

G.2 Header Files for Tapestry Classes

779

```

// string Path() - returns full path to file
// DirEntry() - constructor, directory entry undefined attributes

// ***** DirStream member functions:
//
// DirStream(string name) - constructor (pass name of directory)
// DirStream() - default constructor (use current directory)
// void open(string name) - opens directory stream with given name
// bool fail() - returns true if directory operations has
//               failed, else returns false
// void close() - close stream
//
//
// void Init() - set DirStream so first entry is accessible
// bool HasMore() - returns true if current ok, else false
// void Next() - advance to next entry
// DirEntry Current() - return current directory entry
//                   call only when HasMore = true

#include <string>
using namespace std;

const string DIR_SEPARATOR = "\\"; // platform specific

class DirStream; // need forward reference for friendship
class WIN32_DATA; // defined in .cpp file (avoid parsing huge <windows.h>)

#include "date.h"
#include "clockt.h"

class DirEntry
{
public:
    DirEntry(); // constructor
    ~DirEntry(); // destructor

    string Name() const; // return name (not full path) of file
    string Path() const; // return canonicalized path of file
    long Size() const; // return size (bytes) of file
    bool IsDir() const; // return false if file, true if directory
    Date GetDate() const;
    ClockTime GetTime() const;
    friend DirStream; // class has access to internals

private:
    // this is the private directory entry information
    // it is platform specific, probably should be a 'handle'
    // to a class PrivateDirEntry defined in directory.cc

    DirEntry(WIN32_DATA* dat); // from platform specific constructor

    string myName; // NOT full path, just 'file' name
    string myPath; // full, canonicalized path
    Date myDate; // creation date

```

780

Appendix G How to: Understand and Use Tapestry Classes

```

    ClockTime myTime;           // creation time
    long mySize;                // in bytes
    bool myIsDirectory;        // true if directory, else false
};

class HHandle; // forward, really a HANDLE, but avoid parsing <windows.h>

class DirStream
{
public:
    DirStream(const string & name); // name is path to directory
    DirStream();                   // current directory
    ~DirStream();                  // destructor
    void open(const string & name); // open, bind to file with name
    void close();                  // close the stream
    bool fail();                   // return true if failed, else false

    void Init();                   // standard iterator functions
    void Next();
    bool HasMore();
    DirEntry Current();

    // stuff below is 'esoteric' C++
    //
    // the () method returning void * is what allows
    // the expression: while (dirstream)
    // to work [see Teale, The I/O Stream Handbook]

    operator void *() const
    {
        return myStatus ? (void *) this : (void *) 0;
    };

    // allow the expression if (!dirstream)

    int operator !() const
    {
        return !myStatus;
    }

private:
    HHandle * myStream; // for Windows
    WIN32_DATA * myData; // for Windows
    bool myStatus; // if true, everything ok, else all done
    string myPath; // full path to this directory
    DirEntry myEntry; // cached, current entry
    bool myIsClosed; // already closed myStream?

    void Initialize(const string & s); // private init commonality
    void SetEntry(); // common code (Init/Next)

    // disable assignment and copy

```

```

    DirStream operator = (const DirStream & dir);
    DirStream(const DirStream & dir);
};

#endif /* _DIRECTORY_H not defined */

```

directory.h

G.2.12 The Class CList

Program G.12 clist.h

```

#ifndef _LIST_H
#define _LIST_H

#include <iostream>
#include <string>
using namespace std;

template <class Type> class CListIterator;
template <class Type> class CListPrinter;

// CList is a constant, or immutable list. Once a
// list is created, neither the list nor its contents
// can be changed. This means new lists can safely
// share storage with existing lists since none will be
// changed during program execution
//
// Head(), First()
//     return the first element of a list, error if IsEmpty()
// Tail()
//     returns a list with all but first element
//     list.Tail() is empty if list is empty
// Last()
//     returns last element in list, constant time access
// Size()
//     returns # elements in list, constant time
// IsEmpty()
//     returns true if list is empty, else returns false
// Contains(Type t)
//     returns true iff list contains t
// Find(Type t)
//     returns a (sub)list with Head() == t, or EMPTY if !Contains(t)
// Address()
//     returns a string-sized form of the hex address of the first element
// Printer(), Printer(const string& delimiter)
//     effectively returns a stream manipulator, inserts the list
//     onto a stream with delimiter between elements, the
//     default/no-parameter function inserts newlines between elements
//
//     usage: cout << list.Printer(",") << end;
//
// static ConsCalls() - returns # times cons called

```

782

Appendix G How to: Understand and Use Tapestry Classes

```
// static EMPTY - effectively a constant for the empty list
//
// CListIterator is the standard tapestry iterator, constructed from
// a list

template <class Type>
class CList
{
public:

    CList(); // make an empty list

    // accessors, determine properties of list, get first/last values

    Type Head() const; // abbreviation for First()
    Type First() const; // return copy of first element
    Type Last() const; // return copy of last element
    CList Tail() const;

    bool Contains(const Type & t) const; // true if t in list
    int Size() const; // # of items in list
    bool IsEmpty() const; // true if Size() == 0
    CList<Type> Find(const Type& t) const; // return l with l.Head() == t

    string Address() const;

    CListPrinter<Type> Printer() const;
    CListPrinter<Type> Printer(const string& delimiter) const;

    static

    CList<Type> cons(const Type & s, const CList<Type>& slist);

    static CList<Type> EMPTY;

    static int ConsCalls();

    friend class CListIterator<Type>;

private:
    CList(const Type& t, const CList<Type>& lst); // make a new list

    struct TNode
    {
        // data members
        Type info; // value stored
        TNode * next; // link to next TNode

        // constructors
        TNode()
            : next(0)
        { }
        TNode(const Type & val, TNode * link=0)
            : info(val),
              next(link)
    }
};
```

G.2 Header Files for Tapestry Classes

783

```
    { }
};

TNode * myFirst;      // first node of list
TNode * myLast;      // last node of list
int myCount;          // # of items in list

static int ourConsCount; // # calls of cons
};

template <class Type> inline
CList<Type> cons(const Type& t, const CList<Type>& slist)
{
    return slist.cons(t,slist);
}

template <class Type>
CList<Type> append(const Type& t, const CList<Type>& slist);

template <class Type>
class CListPrinter
{
public:
    CListPrinter(const CList<Type>& list);
    CListPrinter(const CList<Type>& list, const string& delimiter);
    CList<Type> myList;
    string    myDelimiter;
};

template <class Type>
ostream& operator << (ostream& output, const CListPrinter<Type>& p);

template <class Type>
ostream& operator << (ostream& output, const CList<Type>& list);

template <class Type>
class CListIterator
{
public:

    CListIterator(const CList<Type>& list);

    void Init()    const;
    bool HasMore() const;
    void Next()    const;
    Type Current() const;

private:
    typedef CList<Type>::TNode Node;
    Node * myFirst;
    mutable Node * myCurrent;
};
```

784

Appendix G How to: Understand and Use Tapestry Classes

```
#include "clist.cpp"

typedef CList<string> StringList;
typedef CListIterator<string> StringListIterator;

#endif
```

clist.h

G.2.13 The Class Poly

Program G.13 poly.h

```
#ifndef _POLY_H
#define _POLY_H

#include <string>
using namespace std;
#include "clist.h"

// polynomials in 'x' (can be easily modified for
// polys in any variable, and templated for polys of ..)
//
// coefficients are doubles
// Poly() or Poly::ZERO represent 0, otherwise
// Poly(a,b) represents ax^b
//
// polynomials of more than one term are constructed using +=, e.g.,
// Poly a = Poly(5,3) + Poly(4,2) + Poly(3,1) + Poly(2,0)
// then a = 5x^3 + 4x^2 + 3x + 2
//
// Head() returns the leading term, Tail() returns all but Head()
// both return Poly objects [and return non-poly on error]
// IsPoly() returns true if object is a "good" polynomial, e.g.,
// Poly().Tail().IsPoly() == false
//
// accessors include
// leadingCoeff(), degree() for first term
// at(double x) to evaluate a polynomial at x
// toString() - standard helper function
//

class Poly
{
public:
    Poly();
    Poly(double coeff, int exp);

    const Poly& operator += (const Poly& rhs);
    const Poly& operator *= (double c);

    string toString() const;
    double at(double x) const;
```

G.2 Header Files for Tapestry Classes

785

```

int degree() const;
double leadingCoeff() const;

Poly Tail() const;
Poly Head() const;
bool IsPoly() const;

static Poly ZERO;
static int TermsAllocated();

private:
struct Pair // this is the (a,b) in ax^b
{
    double coeff;
    int expo;
    Pair()
        : coeff(0.0),expo(0) { }
    Pair(double c, int e) : coeff(c), expo(e) { }
};
typedef CList<Pair> Polist;
typedef CListIterator<Pair> PolistIterator;
static bool ourInitialized;

Poly(Polist p); // make poly from list of terms, helper
Polist myPoly; // the list of terms
};

Poly operator + (const Poly& lhs, const Poly& rhs);
Poly operator * (double c, const Poly& p);
Poly operator * (const Poly& p, double c);
ostream& operator << (ostream& out, const Poly& p);
#endif

```

poly.h

G.2.14 The Sorting Functions in sortall.h

Program G.14 sortall.h

```

#ifndef _SORTALL_H
#define _SORTALL_H

#include "tvector.h"
#include "comparer.h"

// *****
// prototypes for sort functions and search functions
// author: Owen Astrachan
//
// see also: comparer.h, sortall.cpp
//
// for "plain" sorts, the type being sorted
// must be comparable with < and for Merge and Quick also with <=

```

786

Appendix G How to: Understand and Use Tapestry Classes

```
// for sorts with the Comparer template parameter the type
// for Comparer (see comparer.h) must have a member function
// named compare that takes two const Type arguments: lhs, rhs,
// and which returns -1, 0, or +1 if lhs <, ==, > rhs, respectively
//
// search functions take a Comparer object also
//
// *****

template <class Type>
void InsertSort(tvector<Type> & a, int size);

template <class Type, class Comparer>
void InsertSort(tvector<Type> & a, int size, const Comparer & comp);

template <class Type>
void SelectSort(tvector<Type> & a, int size);

template <class Type, class Comparer>
void SelectSort(tvector<Type> & a, int size, const Comparer & comp);

template <class Type>
void BubbleSort(tvector<Type> & a, int size);

template <class Type>
void MergeSort(tvector<Type> & a, int n);

template <class Type, class Comparer>
void MergeSort(tvector<Type> & a, int n, const Comparer & comp);

template <class Type>
void QuickSort(tvector<Type> & a, int size);

template <class Type, class Compare>
void QuickSort(tvector<Type> & a, int size, const Compare& comp);

template <class Type>
void HeapSort(tvector<Type>& a, int size);

template <class Type>
void Swap(tvector<Type>& v, int j, int k);
// post: v[k] and v[j] swapped

// searching functions

template <class Type>
int bsearch(const tvector<Type>& list, const Type& key);

template <class Type, class Comparer>
int bsearch(const tvector<Type>& list, const Type& key, const Comparer& c);

template <class Type, class Comparer>
int search(const tvector<Type>& list, const Type& key, const Comparer& c);
```

```

template <class Type>
int search(const tvector<Type>& list, const Type& key);

#include "sortall.cpp"

#endif

```

sortall.h

G.2.15 Header Files from Circuit Simulation

Program G.15 wires.h

```

#ifndef _WIRES_H
#define _WIRES_H

#include <iostream>
#include <string>
using namespace std;
#include "tvector.h"

// A wire has current flowing through it.
// When the current changes, a wire notifies
// all the gates listening on the wire that the current
// has changed. The gates act accordingly (see gates.h)
//
// Some of the "gates" are really connectors to other wires
// A connector allows gates to be strung together, but connectors
// are more like solder than real gates. As such, connectors aren't
// part of a CompositeGate. To facilitate finding a wire's
// connectors, a ConnectorIterator can be used to access all a wire's
// connectors.

class Gate;
class Connector;

class Wire
{
public:
    Wire(const string& name="");
    virtual ~Wire();
    virtual bool GetSignal() const; // true/false, on/off
    virtual string toString() const; // for I/O

    virtual void SetSignal(bool signal); // set signal, propagate
    virtual void AddGate(Gate * g); // g monitors this wire
    virtual void RemoveGate(Gate * g); // g stops monitoring
    virtual int Number() const; // which wire is this?

    friend class ConnectorIterator; // access myGates

private:

```

788

Appendix G How to: Understand and Use Tapestry Classes

```

    tvector<Gate *>    myGates;
    bool              mySignal;
    string             myName;
    int                myNumber;
    static int        ourCount;           // class wide, keeps count
};

ostream& operator << (ostream& out, const Wire& w);

// A WireFactory is used to encapsulate wire creation
// If wires are "ordered" from the factory, the factory takes
// care of cleaning up the wires when the factory ceases to exist
// This is a rudimentary factory, there's no facility for clients
// to recycle wires and the factory doesn't clean up the gates
// attached to the wires it destroys
//
//
// MakeWire - creates a new wire
// GetWire - retrieves an already created wire by the wire's number

class WireFactory
{
public:
    WireFactory();
    virtual ~WireFactory();
    virtual Wire * MakeWire(const string& name="wire"); // create anew
    virtual Wire * GetWire(int num) const;           // get by number
private:
    tvector<Wire *> myWires;
};

// standard tapestry iterator for iterating over all
// connectors attached to a wire

class ConnectorIterator
{
public:
    ConnectorIterator(Wire* w);
    void Init();
    bool HasMore();
    void Next();
    Connector * Current();
private:
    Wire *      myWire;
    Connector * myConnector;
    int         myIndex;
};

#endif

```

wires.h

Program G.16 gates.h

```
#ifndef _GATES_H
#define _GATES_H

#include <iostream>
#include <string>
using namespace std;
#include "tvector.h"

class Wire;
class WireFactory;
class Gate
{
public:
    virtual ~Gate() {}
    virtual void Act() = 0;
    virtual string toString() const = 0;
    virtual int    InCount() const = 0;
    virtual int    OutCount() const = 0;
    virtual Wire * InWire(int n) const = 0;
    virtual Wire * OutWire(int n) const = 0;
    virtual Gate * clone() = 0;

    virtual string deepString() const { return toString();}

    static Wire * WireByNumber(int num);

protected:
    static WireFactory * ourWireFactory;
};

ostream& operator << (ostream& out, const Gate& g);

class Connector : public Gate
{
public:
    Connector(Wire * in, Wire * out);
    virtual void Act();
    virtual string toString() const;
    int InCount() const {return 1;}
    int OutCount() const {return 1;}
    Wire * InWire(int n) const {return myIn;}
    Wire * OutWire(int n) const {return myOut;}
    Gate * clone();
private:
    Wire * myIn;
    Wire * myOut;
};

class Inverter : public Gate
{
```

790

Appendix G How to: Understand and Use Tapestry Classes

```

public:
    Inverter(Wire * in, Wire * out, const string& name="");
    Inverter(const string& name="");
    virtual void Act( );
    virtual string toString() const;

    int InCount() const {return 1;}
    int OutCount() const {return 1;}
    Wire * InWire(int n) const {return myIn;}
    Wire * OutWire(int n) const {return myOut;}
    Gate * clone();

    virtual string deepString() const;

private:
    Wire * myIn;
    Wire * myOut;
    string myName;
    int myNumber;
    static int ourCount;
};

class NMGate : public Gate
{
public:

    virtual void Act( ) = 0;
    virtual string toString() const = 0;

    int InCount() const {return myIns.size();}
    int OutCount() const {return myOuts.size();}

    Wire * InWire(int n) const {return myIns[n];}
    Wire * OutWire(int n) const {return myOuts[n];}

    virtual string deepString() const;

protected :

    NMGate(int number=0, const string& name="generic");

    void Init(const tvector<Wire *>& in, const tvector<Wire *>& out);
    tvector<Wire *> myIns;
    tvector<Wire *> myOuts;
    int myNumber;
    string myName;
};

class AndGate : public NMGate
{
public:
    AndGate(Wire * in, Wire * in2, Wire * out, const string& name = "");
    AndGate(const string& name="");
    virtual void Act( );
    virtual string toString() const;

```

G.2 Header Files for Tapestry Classes

791

```
    Gate * clone();
private:
    static int ourCount;
};

class OrGate : public NMGate
{
public:
    OrGate(Wire * in, Wire * in2, Wire * out, const string& name = "");
    OrGate(const string& name = "");
    virtual void Act( );
    virtual string toString() const;
    Gate * clone();
private:
    static int ourCount;
};

class CompositeGate : public NMGate
{
public:
    CompositeGate();
    virtual string toString() const;
    virtual void Act();

    virtual void AddIn(Wire * w);
    virtual void AddOut(Wire * w);
    virtual void AddGate(Gate * g);

    virtual Gate * clone();
    virtual string deepString() const;
    virtual int    CountWires() const;

private:
    tvector<Gate *> myGates;
};

class Probe : public Gate
{
public:
    Probe (Wire * w);
    virtual void Act( );
    virtual string toString() const;

    int InCount() const {return 1;}
    int OutCount() const {return 1;}
    Wire * InWire(int n) const {return myWire;}
    Wire * OutWire(int n) const {return myWire;}
    Gate * clone() {return this;}

protected:
    Wire * myWire;
};
```

792

Appendix G How to: Understand and Use Tapestry Classes

```
class GateTester
{
  public:
    static void Test(Gate * gate);
};

void Connect(Wire * w1, Wire * w2);

#endif
```

gates.h

G.2.16 The Map Class SimpleMap

Program G.17 simplemap.h

```
#ifndef _SIMPLEMAP_H
#define _SIMPLEMAP_H

#include "tvector.h"

// simple map, supports (for map m):
//   m.insert(Key,Value);
//   v = m.getValue(k); // returns default Value() if k not found

template <class Key,class Value> class SimpleMapIterator;

template <class Key, class Value>
class SimpleMap
{
  public:
    SimpleMap()
    { }
    void insert(const Key& k, const Value& v)
    {
      myKeys.push_back(k);
      myValues.push_back(v);
    }
    Value getValue(const Key& key) const
    {
      for(int k=0; k < myKeys.size(); k++)
      {
        if (myKeys[k] == key) return myValues[k];
      }
      return Value();
    }
}

friend class SimpleMapIterator<Key,Value>;

private:
  tvector<Key> myKeys;
  tvector<Value> myValues;
```

```
};

template <class Key,class Value>
class SimpleMapIterator
{
public:
    SimpleMapIterator(const SimpleMap<Key,Value>& map)
        : myMap(map),
          myIndex(-1)
    { }
    void Init()
    { myIndex = 0;
    }
    bool HasMore()
    { return myIndex < myMap.myKeys.size();
    }
    bool Next()
    { myIndex++;
    }
    Key Current()
    { return myMap.myKeys[myIndex];
    }
private:
    const SimpleMap<Key,Value>& myMap;
    int myIndex;
};

#endif
```

simplemap.h

Bibliography

- [AA85] Donald J. Albers and G.L. Alexanderson. *Mathematical People*. Birkhäuser, 1985.
- [ACM87] ACM. *Turing Award Lectures: The First Twenty Years 1966–1985*. ACM Press, 1987.
- [AS96] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. 2nd ed. MIT Press and McGraw-Hill, 1996.
- [Asp90] William Aspray. *Computing Before Computers*. Iowa State University Press, 1990.
- [Aus98] Matthew H. Austern *Generic Programming and the STL*. Addison-Wesley, 1998.
- [Ben86] Jon Bentley. *Programming Pearls*. Addison-Wesley, 1986.
- [Ben88] Jon Bentley. *More Programming Pearls*. Addison-Wesley, 1988.
- [Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.
- [Boo91] Grady Booch. *Object Oriented Design with Applications*. Benjamin Cummings, 1991.
- [Boo94] Grady Booch. *Object Oriented Design and Analysis with Applications*. 2nd ed. Benjamin Cummings, 1994.
- [BRE71] I. Barrodale, F.D. Roberts, and B.L. Ehle. *Elementary Computer Applications in Science Engineering and Business*. John Wiley & Sons Inc., 1971.
- [Coo87] Doug Cooper. *Condensed Pascal*. W.W. Norton, 1987.
- [Dij82] Edsger W. Dijkstra. *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 1982.
- [DR90] Nachum Dershowitz and Edward M. Reingold. Calendrical calculations. *Software-Practice and Experience*, 20(9):899–928, September 1990.
- [(ed91] Allen B. Tucker (ed.). *Computing Curricula 1991 Report of the ACM/IEEE-CS Joint Curriculum Task Force*. ACM Press, 1991.
- [EL94] Susan Epstein and Joanne Luciano, editors. *Grace Hopper Celebration of Women in Computing*. Computing Research Association, 1994. Hopper-Book@cra.org.

- [Emm93] Michele Emmer, editor. *The Visual Mind: Art and Mathematics*. MIT Press, 1993.
- [G95] Denise W. Gürer. Pioneering women in computer science. *Communications of the ACM*, 38(1):45–54, January 1995.
- [Gar95] Simson Garfinkel. *PGP: Pretty Good Privacy*. O'Reilly & Associates, 1995.
- [GHJ95] Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Programming*. Addison-Wesley, 1995.
- [Gol93] Herman H. Goldstine. *The Computer from Pascal to von Neumann*. Princeton University Press, 1993.
- [Gri74] David Gries. On structured programming - a reply to smoliar. *Communications of the ACM*, 17(11):655–657, 1974.
- [GS93] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Springer-Verlag, 1993.
- [Har92] David Harel. *Algorithmics, The Spirit of Computing*. Addison-Wesley, second edition, 1992.
- [Hoa89] C.A.R. Hoare. *Essays in Computing Science*. Prentice-Hall, 1989. (editor) C.B. Jones.
- [Hod83] Andrew Hodges. *Alan Turing: The Enigma*. Simon & Schuster, 1983.
- [Hor92] John Horgan. Claude e. shannon. *IEEE Spectrum*, April 1992.
- [JW89] William Strunk Jr. and E.B. White. *The Elements of Style*. MacMillan Publishing Co., third edition, 1989.
- [Knu97] Donald E. Knuth. *The Art of Computer Programming*, volume 1 Fundamental Algorithms. Addison-Wesley, third edition, 1997.
- [Knu98a] Donald E. Knuth. *The Art of Computer Programming*, volume 2 Seminumerical Algorithms. Addison-Wesley, third edition, 1998.
- [Knu98b] Donald E. Knuth. *The Art of Computer Programming*, volume 3 Sorting and Searching. Addison-Wesley, third edition 1998.
- [KR78] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, 1978.
- [KR96] Samuel N. Kamin and Edward M. Reingold. *Programming with class: A C++ Introduction to Computer Science*. McGraw-Hill, 1996.
- [Mac92] Norman Macrae. *John von Neumann*. Pantheon Books, 1992.

- [McC79] Pamela McCorduck. *Machines Who Think*. W.H. Freeman and Company, 1979.
- [McC93] Steve McConnell. *Code Complete*. Microsoft Press, 1993.
- [MGRS91] Albert R. Meyer, John V. Gutag, Ronald L. Rivest, and Peter Szolovits, editors. *Research Directions in Computer Science: An MIT Perspective*. MIT Press, 1991.
- [Neu95] Peter G. Neumann. *Computer Related Risks*. Addison Wesley, 1995.
- [Pat96] Richard E. Pattis. *Get A-Life: Advice for the Beginning Object-Oriented Programmer*. Turing TarPit Press, 1999.
- [Per87] Alan Perlis. The synthesis of algorithmic systems. In *ACM Turing Award Lectures: The First Twenty Years*. ACM Press, 1987.
- [PL90] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.
- [RDC93] Edward M. Reingold, Nachum Dershowitz, and Stewart M. Clamen. Calendrical calculations, ii: Three historical calendars. *Software-Practice and Experience*, 23(4):383–404, April 1993.
- [Rie96] Arthur Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.
- [Rob95] Eric S. Roberts. Loop exits and structured programming: Reopening the debate. In *Papers of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education*, pages 268–272. ACM Press, March 1995. SIGCSE Bulletin V. 27 N 1.
- [Rob95] Eric S. Roberts. *The Art and Science of C*. Addison-Wesley, 1995.
- [Sla87] Robert Slater. *Portraits in Silicon*. MIT Press, 1987.
- [Str87] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1987.
- [Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.
- [Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition, 1997.
- [Mey92] Scott Meyers. *Effective C++*. Addison Wesley, 1992.
- [Mey96] Scott Meyers. *More Effective C++*. Addison-Wesley, 1996.
- [Wei94] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Benjamin Cummings, 1994.
- [Wil56] M.V. Wilkes. *Automatic Digital Computers*. John Wiley & Sons, Inc., 1956.
- [Wil87] Maurice V. Wilkes. Computers then and now. In *ACM Turing Award Lectures: The First Twenty Years*, pages 197–205. ACM Press, 1987.

824

Appendix G How to: Understand and Use Tapestry Classes

- [Wi195] Maurice V. Wilkes. *Computing Perspectives*. Morgan Kaufmann, 1995.
- [Wir87] Niklaus Wirth. From programming language design to compiler construction. In *ACM Turing Award Lectures: The First Twenty Years*. ACM Press, 1987.