ECE322C – DATA STRUCTURES – UNIQUE SECTION # 15515 COURSE SYLLABUS

Fall 2004 – Version 1.0

INSTRUCTOR Dewayne E Perry; **EMAIL:** perry@ece.utexas.edu
OFFICE: ENS 623 **PHONE:** 471-2050

We also have a TA and grader associated with the teaching team for this course. Their information will be provided separately.

OFFICE HOURS MW 4:00-5:00, and by appointment. The office/laboratory hours for your teaching assistant will be posted on the *EE322C web page* during the second week of classes.

CLASS MEETING SPECIFICS. MW 5-6:30, CPE 2.210

COURSE OBJECTIVE: Prepare students for future subjects in ECE, software engineering and/or systems software

SUBJECTS COVERED: Programming with abstractions; programming in C++; fundamental data structures; templates; algorithm analysis; program design approaches. The topics included in this course are covered roughly in this order: From C to C++, C++ classes, Asymptotic Analysis, Dynamic Arrays, Lists, Stacks, Queues, Trees, Hashing, Priority Queues, Sets, and Maps.

PREREQUISITES. EE312 with a grade of at least *C*. Incoming students are expected to know the basics of computers and computation; and how to program in C using features of the language, such as: variables and operators, builtin data types, execution control structures, pointers, arrays, screen I/O, structs , linked lists, and recursion. The student should know how to use basic programming tools and techniques, such as: a programming language development toolset and symbolic debugging. The incoming student may or may not have been introduced to subjects such as: abstract data types, analysis of algorithms, program design techniques, object-oriented programming, advanced C++ features (e.g. classes, templates, etc.).

TEXT. There is no required textbook for this course. Course notes and examples will be provided electronically, e.g. on the course homepage.

SUGGESTED REFERENCES *EE 312 Lecture Notes*, Terry J. Wagner, Fall, 2002. (These notes are on his EE322C homepage.) *C++ Programming Language, 3rd Edition, B. Stroustrup, Addison Wesley, 2000,* and *Thinking in C++*, Bruce Eckel, Prentice-Hall, 2000 (Second Edition). (Other references may be found on the EE322C homepage.)

ATTENDANCE Attendance is expected. Whether you come to class or not, you are responsible for keeping up with what happens in class. If you miss a class (other than for illness or an emergency), it is not reasonable for you to expect me to repeat just for you the material that was covered in the class that you missed. This applies both to the content of the class as well as to announcements about class policies, events, deadlines, or whatever. You can expect a loss of at least one letter grade if you miss four or more lectures.

DROPPING. I will not sign any drop card after the second week of classes unless it is approved beforehand by Dean Meyer's office. The course grade that I assign in these cases will always be my estimate of your current grade. In particular, your grade must be a 'C' or better in order to receive a 'Q' on your drop application.

COURSE GRADES. Course grades will be based on the following components.

Component	**Date**	**Weight**
Exam 1	September 27	20%
Exam 2	October 27	20%
Exam 3 (final)	December 1	20%
Assignments	(Due dates are stated in each assignment) 6-8 assignments are planned.	40%

The grade you are given, either on an individual exam or assignment or as your final grade, is not the starting point of a negotiation. It is your grade unless a concrete error has been made. Do not come to see me or the TA to ask for a better grade because you want one or you "feel you deserve it". Come only if you can document a specific error in grading or in recording your scores. Errors can certainly be made in grading, especially when many students are involved. But keep in mind that the errors can be made either in your favor or not. So it's possible that if you ask to have a piece of work re-graded your grade will go down rather than up.

Remember that the most important characteristic of any grading scheme is that it be fair to everyone in the class. Keep this in mind if you're thinking of asking, for example, for more partial credit points on a problem. The important thing is not the exact number of points that were taken off for each kind of mistake. The important thing is that that number was the same for everyone. So it can't be changed once the grading is done and the exams or assignments have been returned.

If you have questions or concerns about any of your grades, contact me during office hours or via email.

Final Grades: Final grades will be assigned according to the following standard criteria:

Final Average	Letter Grade
90 - 100	A
80 - 89	B
70 - 79	C

60 - 69	D
0 - 59	F

Final class grades will be calculated to 2 decimal places and rounded to the nearest integer. 89.49 is a B. 89.50 is an A. The line has to be drawn somewhere, and no special allowances will be made for students whose final average falls near, but below the cutoff. Nonacademic explanations for poor class performance will have no bearing on the assignment of grades.

EXAMINATIONS. Exams will cover material from lecture, assignments, and the assigned readings. Exams will be cumulative, although they will be more heavily weighted towards material not yet tested. Programming is a cumulative discipline, so it is important to master earlier topics in order to understand later topics. Exam scores may be curved if the instructor believes it is warranted. Three exams will be given in class. The dates are shown in the table above. If your work or a personal situation forces you to unexpectedly miss exams, you should expect to get a zero on those occasions. If you miss an exam because of illness, you are expected to provide a statement from a doctor stating that, in his/her opinion, it was impossible for you to attend because of illness. A slip showing you visited the UT Health Center or your personal doctor is not sufficient for this. In any other situation, you should contact me beforehand.

PROGRAMMING ASSIGNMENTS. The assigned class work in this course will consist of up to ten programming assignments. Programming is a discipline that you learn by doing, not by listening to a lecturer. Therefore, doing the programming assignments is crucial to performing well in this class. Assignments will be given almost every week. Each assignment will have a clearly stated due date and time. **No late assignments will be accepted**. If you are having considerable difficulty with the early assignments, this is a sign that you may be in over your head - you should come see me immediately. The assignments will require a substantial time commitment over several days (an average of 6 hours per week outside of class should be expected). Be sure to budget sufficient time to complete assignments before the deadline. At the time you submit each assignment for grading, you are **required** to make a backup copy of the source code file on your removable secondary storage device (e.g. a floppy or ZIP disk). This will be necessary in cases where your program gets lost, is corrupted, or if there is some dispute over what was turned in when. If you cannot finish an assignment on time, then submit whatever you have finished before the deadline to receive partial credit.

SUBMITTING PROGRAMMING ASSIGNMENTS. There will be a separate document detailing the process for submitting homework. It will be on the class web site at the time of the first assignment.

PROGRAMMING ASSIGNMENT GRADES. Assignment grading criteria may vary on each assignment. However, in general, programs that do not compile correctly on the Lab configuration will receive no more than 25% of the possible points. Larger point deductions are given for such things as: incorrect results, missing features, bad solution logic, poor style, etc. With regard to programming style, I expect you to follow the C++ coding standards that are found on the course web page. These coding standards will be loosely enforced on the early assignments but rigorously enforced on the latter assignments. In addition the following criteria are important: (i) a block structured design should be evident; (ii) comments and/or appropriate variable names should be used to make your program readable; (iii) appropriate prompts and messages for input and output should be given to the user of your program. Unless stated otherwise, more emphasis will be placed on program clarity than on program length, speed or size. For pair programming assignments, each member will receive the same grade as the other.

PROGRAMMING LAB and SOFTWARE. The computers in the third-floor labs of ENS have all of the software needed for this course. If you plan to do all of your work at home, you will need a *C*/*C*++ compiler, a web browser, an email program, and the program *Adobe Acrobat Reader*. All of the programs are free except the *C*/*C*++ compiler, which can be purchased at the *Campus Computer Store* or directly from Metrowerks. Programs submitted are expected to run on the version of Metrowerks CodeWarrior currently installed in these labs. If your program doesn't run on the CodeWarrior version currently in the labs, you can expect to lose points even if it runs on some other compiler.

GRADE DISPUTES AND CORRECTIONS. If you are dissatisfied with a grade you receive, you must submit your complaint briefly in writing or by email, along with supporting evidence or arguments, to me within one week of the date that I (or the TA) first attempted to return the exam or assignment to you. Complaints about grades received after the one-week deadline will be considered only if there are extraordinary circumstances for missing the deadline (e.g. student hospitalization). No new disputes will be accepted after noon of the day before the course grade sheets must be turned in.

POSTED INFORMATION. The scores for tests and programming assignments will be posted on the *class web page* using a six-digit random number I will give you near the beginning of the semester.

CLASS WEB PAGE. . Course materials (e.g. the syllabus, assignments, etc.) and grades will be available via this semester's web page. This will be the main source of current class information: (i) the daily class announcements, (ii) the programming assignments, (iii) solutions to programming assignments and exams, (iv) course reading materials, (v) TA office hours and so on.

USE OF EMAIL. You cannot expect to get last-minute help on assignments by email. More generally, you cannot expect to get detailed answers to technical questions by email. Students are encouraged to discuss important matters with the teaching team in person, typically after class or during lab or office hours. In email, include your name, and the number of the assignment or exam in question. Please include your name in the "From:" line of the email message, not just your email address. Email accounts are available free to students from the university and commercial sources. Some commercial providers filter your email, and so you may not be receiving appended documents unless you set the permissions to do so (this is particularly true of HOTMAIL accounts). Email is a valuable tool for communicating with the teaching team. But be sure to use it properly, and follow the rules of good email etiquette (e.g. no flaming, spamming, etc.). Although it's easy for you to dash off an email question, it takes time to answer it. Don't ask questions to which you can find the answer somewhere else (e.g. class notes, web page).

OTHER COURSE RELATED POLICIES

ACADEMIC DISHONESTY (cheating): The University and the Department are committed to preserving the reputation of your UT degree. In order to guarantee that every degree means what it says it means, we must enforce a strict policy on academic honesty: Every piece of work that you turn in with your name on it must be yours and yours alone. No co-working is allowed on any test, project, or programming assignment unless explicitly allowed (*). As an honest student, you are responsible for enforcing this policy in three ways:

1. You must not turn in work that is not yours, except as expressly permitted by me
2. You must not enable someone else to turn in work that is not his or hers. Do not share your work with anyone else. Make sure that you adequately protect all your files. Even after you have finished a class, do not share your work or published answers with the students who come after you. They need to do their work on their own.
3. You must not allow someone to openly violate this policy because it diminishes your effort as well as that of your honest classmates.

Students who violate University rules on scholastic dishonesty in assignments or exams are subject to disciplinary penalties, including the possibility of a lowered or 0 grade on an assignment or exam, failure in the course, and/or dismissal from the University. Changing your exam answers after they have been graded, copying answers during exams, or plagiarizing the work of others (in programming assignments) will be considered academic dishonesty and will not be tolerated. Plagiarism detection software may be used on the programs submitted in this class. If cheating is discovered, a report will be made to the Dean of Students recommending a course grade of 'F' for all involved in the incident. (*) In this course when we do pair programming, the pair is treated as one individual with regard to this policy.

LEARNING DISABILITIES. If you have a learning disability that requires special attention, either during class or during an exam, please give me a letter from the Dean of Students describing what needs to be done. You should do this during the first week of classes. (The University of Texas at Austin provides upon request appropriate academic accommodations for qualified students with disabilities. For more information, contact the Office of the Dean of Students at 471-6259, 471-4641.)

RELIGIOUS HOLY DAYS. A student who is absent from an examination or cannot meet an assignment deadline due to the observance of a religious holy day may take the examination on an alternate day, submit the assignment up to 24 hours late without penalty, or be excused from the examination or assignment, if proper notice of the planned absence has been given. Notice must be given at least fourteen days prior to the classes scheduled on dates the student will be absent. For religious holy days that fall within the first two weeks of the semester, notice should be given on the first day of the semester. It must be personally delivered to the instructor and signed and dated by the instructor, or sent certified mail, return receipt requested. Email notification will be accepted if received, but a student submitting such notification must receive email confirmation from the instructor. A student who fails to complete missed work within the time allowed will be subject to the normal academic penalties.

CLASSROOM BEHAVIOR. You have the right to learn in every class you attend. But you also have the responsibility to help assure that every other student shares that right. Specifically:

1. Under normal circumstances, class will start on time and end on time.

2. Come to class on time. Do not leave early. These things are very disruptive. Recognize that the buses and the parking space situation are unpredictable elements and allow for that. If you must come late or leave early (for example because of a doctor's appointment), let the instructor know in advance.
3. Don't be disruptive during class. Don't chat with your neighbors or rustle the newspaper.
4. Don't allow your electronic devices to be disruptive. Turn off your cell phone, beeper, and watch alarm.
5. Don’t leave your mess lying on the classroom floor when you leave – pick it up and throw it in a trash can.

EXTERNAL TUTORING. For those students having considerable difficulties with the course material, individual tutoring is provided by certain organizations not directly affiliated with this course. See the following references:

1. Eta Kappa Nu – they will announce their tutoring schedule shortly after the semester begins
2. The Learning Skills Center, located in Jester A332A, 471-3614, has individual tutors for hire. The cost is about $10 per hour; students receiving financial aid can get 2 hours per week free. See the web page http://www.utexas.edu/student/utlc/tap.html for more information.

DISCLAIMER. I occasionally tell jokes and stories during class as a way of breaking up the technical material that we're covering. A story might be a simple observation about campus life, about something that has happened to me as a student or a professor, or it may even be an aggie joke. These stories may reflect a point of view that is different from your own. Hopefully, most will be interesting or funny. Some, almost certainly, will be politically incorrect. None are intended to be offensive.

COURSE POLICIES CAVEAT. As departmental, college and UT policies change, I reserve the right to alter the effected course policies stated herein during the semester.

C/C++ Differences Highlighted

These notes are an extraction of Dr. Wagner's full set of notes on C and C++ programming *(EE 312 Lecture Notes, Terry J. Wagner, Fall, 2002)* that can be found on his class web site at

http://www.ece.utexas.edu/~tjwagner/EE322C/Notes.html

This extracted set of notes covers the first six chapters of the full set and highlights the main differences between C and C++ in order to help you make the transition to becoming proficient in C++. The subjects range from simple to more complex differences. We will be referring to these notes over the next few weeks in our class.

1. Simple C/C++ Program Differences

Commenting style. *C*++ has an alternate way of commenting your program that is illustrated in the outline in figure 1.1. In particular, anything on the line after '//' is ignored by the *C*++ compiler. In addition, the statement

```
return 0;
```

can be omitted without generating a compiler error (or warning). We will usually do this in all of our *C*++ examples.

```
#include <stdio.h>

int main (void)
/* C++ comments can extend
      over several lines */
{
      ...;          // C++ comment placed after a C++ statement                .
      .
      .
      ...;
}
```

Figure 1.1. The outline of a simple *C*++ program.

Variable declarations. *C*++ allows variables to be declared or defined anywhere within the program as illustrated in the program of figure 1.2. Declaring variables when they are needed usually makes the program more readable.

```
#include <stdio.h>
int main (void)
{
   int m = 10;
   printf("The value of m equals %d\n", m);
   int n = 20;
   printf("The value of n equals %d\n", n);
}

/* CodeWarrior Pro 7 input/output:
The value of m equals 10
The value of n equals 20
*/
```

Figure 1.2. Relaxed declaration rule of *C*++.

C++ allows variables to be defined in a different way as illustrated in the program of figure 1.3. This method becomes quite useful later with *C*++ objects.

```
#include <stdio.h>
int main (void)
{
   int m = 10;
   printf("The value of m equals %d\n", m);
   int n(20);       //This is an alternate method of defining a variable
   printf("The value of n equals %d\n", n);
}

/* CodeWarrior Pro 7 input/output:
The value of m equals 10
The value of n equals 20
*/
```

Figure 1.3. Alternate method of defining variables in *C*++.

If *m* is an *int* variable, it can be cast to a *float* value *z* in two other ways:

```
   z = float(m);
   z = static_cast<float>(m);
```

As we will see after chapter 2, the first method above seems more natural since it looks like a function call. The usefulness of the second method can't really be appreciated without a deeper understanding of casting in *C*++.

New data types. *C*++ has a Boolean type *bool* that takes the values *true* and *false*. However, when it is used in a *printf*() statement, it is converted to an integer and output as an integer. When an *int* type is assigned to a *bool*, *true* becomes 1 and 0 becomes *false*. When a *bool* type is assigned to an *int*, a nonzero *int* value becomes *true* and a zero *int* value becomes *false*. The *bool* type is illustrated in figure 1.4. Notice a *bool* type cannot be input directly from the keyboard with *scanf*(). Instead, one must input an integer value and convert it to a *bool* as described above.

```
#include <stdio.h>
int main (void)
{
   bool b = false;
   printf("b equals %d\n", b);
   int m = b;
   printf("m equals %d\n", m);
   b = true;
   printf("b equals %d\n", b);
   m = b;
   printf("m equals %d\n", m);
   m = 17;
   printf("m equals %d\n", m);
   b = m;
   printf("b equals %d\n", b);
   m = 0;
   printf("m equals %d\n", m);
   b = m;
   printf("b equals %d\n", b);
   printf("Type 0 or 1 to input b:  ");
   scanf("%d", &m);
   b = m;
   printf("b equals %d\n", b);
}

/* CodeWarrior Pro 7 input/output:
b equals 0
m equals 0
b equals 1
m equals 1
m equals 17
b equals 1
m equals 0
b equals 0
Type 0 or 1 to input b:  0
b equals 0
*/
```

Figure 1.4. The type *bool* in *C++*.

In addition to bool there is another new basic data type called **wchar_t**, or wide character. This provides a 16 bit representation for each character which expands the vocabulary greatly. This data type and its related functions will be discussed in more detail later in the course.

Input/output. Input/output in *C*++ is usually handled with the two operators '<<' and '>>', called the ***insertion operator*** and the ***extraction operator***, respectively. These operators, which are also defined in *C* for specific bit operations, have been changed or ***overloaded*** in *C*++ to also have meanings for input/output. Some examples are shown in the program *Output.cp* of figure 1.5.

The first two lines of figure 1.12.5 tell the *C*++ compiler to use the iostream class which contains the definitions of the overloaded operators '<<' and '>>'. These two lines will be in most of our *C*++ programs. The first statement of *main*() inserts a literal string into the output stream with the binary operator '<<' where *cout* is the standard output stream, the one that appears in the console window. (Notice the carriage return at the end of the string.) The third statement inserts four *char* values into *cout*. Notice how the operator '<<' can be used to output values in a left-to-right order. The next-to-last two uses of '<<' show how easy it is to insert different types into *cout* using the same concatenation form used for different values of the same type. The standard output for floating-point values is slightly different for *C*++. Six total decimal digits are used with trailing zeros to the right of the decimal point suppressed. If the floating-point value is very large or very small, an exponential format is used as shown by the last use of '<<'. Finally, notice *endl* can be used in place of the return character '\n'.

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

int main (void) {
   float x;
   int n;
   bool b;
   char c, d, e, f;
   cin >> x >> n >> b >> c >> d >> e >> f;
   cout << "x = " << x << endl;
   cout << "n = " << n << endl;
   cout << "b = " << b << endl;
   cout << "c = " << c << endl;
   cout << "d = " << d << endl;
   cout << "e = " << e << endl;
   cout << "f = " << f << endl;
   x = 1234567.89;
   cout << "x = " << x << endl; }

/*CodeWarrior Pro 7 input/output:
12.3  31  1  wx  yz
x = 12.3
n = 31
b = 1
c = w
d = x
e = y
f = z
x = 1.23457e+06
*/
```

Figure 1.5. The program *Output.cp*.

The extraction operator '>>' will skip over white space until it finds a string of non-white space characters that it will try to interpret as a value of the variable it is looking for. This is illustrated in the program *Input.cp* of figure 1.12.6. Note a *bool* value must be input with a 0 or 1 when the extraction operator '>>' is used for input.

The *iostream* member function *get*() can be used to get the next character in the input stream, including a white-space character. For example, assume the following declaration

```
char x;
```

Then

```
x = cin.get();
```

or

```
cin.get(x);
```

will give *x* the value of the next character in the input stream. (The function *get*() actually returns the integer value of the next character in the buffer, which is converted to the corresponding ASCII character by either of the two calls.) The function *peek*(), another input stream member function, returns the integer value of the next character in the buffer without reading it with

```
x = cin.peek();
```

or

```
cin.peek(x);
```

If the input stream is empty, the program will wait until another line is input from the console.

```
#include <iostream>
using namespace std;
int main (void)
{
   float x, y;
   cout << "Input the value of x followed by the value of y: ";
   cin >> x >> y;
   cout << "The value of x equals " << x << "; the value of y equals " << y << '\n';
   int m, n;
   cout << "Input the value of m followed by the value of n: ";
   cin >> m >> n;
   cout << "The value of m equals " << m << "; the value of n equals " << n << '\n';
}
```

```
/* CodeWarrior Pro 7 input/output:
Input the value of x followed by the value of y: 1.1 2.2
The value of x equals 1.1; the value of y equals 2.2
Input the value of m followed by the value of n: 1 2
The value of m equals 1; the value of n equals 2
*/
```

Figure 1.6. The program *Input.cp*.

Manipulators are data objects that are inserted into the output stream to change the stream. For example, if the manipulator *endl* is put into the output stream *cout*, a carriage return character is put into the output stream and all the data in the stream to that point is output, including the carriage return character. In effect, the manipulator *endl* flushes the output stream buffer. In order to use all the manipulators in the *C*++ standard library, your program should include the directive

```
#include <iomanip>
```

The most commonly used manipulators for floating-point output are:

```
setiosflags()
resetiosflags()
setprecision()
setw()
```

The first two allow the output of a floating-point number to be a fixed or scientific format and to be right- or left-justified, the third allows the number of digits to the right of the decimal point to be set, and the fourth sets the width of the display field for the object that follows. The objects are then displayed in that field with the current justification. (The default justification is right.) Unlike the other manipulators, which remain valid until changed, the *setw*() manipulator must be used before every object output.

The program *Manipulators.cp*, shown in the figure 1.7, illustrates the use of these four manipulators. Notice the standard format uses six *total* digits for a floating-point value where leading zeros are not counted and trailing zeros are suppressed for numbers output in fixed-point form. With very large or very small numbers, the output is automatically changed to scientific notation where six digits are used for the significand. Notice also using the value '6' with *setprecision*() returns the output to the standard format.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main(void)
{
  double x = 33.979999, y = 0.000001336789, z = 100000002.4567;
  cout << "Print all numbers in the standard format\n";
  cout << "x = " << x << "\ny = " << y << "\nz = " << z << endl;
  cout << "Print all numbers in a fixed notation with a precision of 2\n";
  cout << setiosflags(ios::fixed) << setprecision(2)
      << "x = " << x << "\ny = " << y << "\nz = " << z << endl;
  cout << "Print all numbers in a scientific notation with a precision of 3\n";
  cout << resetiosflags(ios::fixed) << setiosflags(ios::scientific) << setprecision(3)
      << "x = " << x << "\ny = " << y << "\nz = " << z << endl;
  cout << "Print all numbers with a precision of 0 in the standard format\n";
  cout << resetiosflags(ios::scientific) << setprecision(0)
      << "x = " << x << "\ny = " << y << "\nz = " << z << endl;
```

```
   cout << "Print all numbers in the standard format\n";
   cout << setprecision(6) << "x = " << x << "\ny = " << y << "\nz = " << z << endl;
   cout << "Print all numbers in the standard format using a field of 15 spaces\n";
   cout << "x = " << setw(15) << x << " y = " << setw(15) << y << " z = " << setw(15) << z << endl;
}


/* CodeWarrior Pro 7 input/output:
Print all numbers in the standard format
x = 33.98
y = 1.33679e-06
z = 1e+08
Print all numbers in a fixed notation with a precision of 2
x = 33.98
y = 0.00
z = 100000002.46
Print all numbers in a scientific notation with a precision of 3
x = 3.398e+01
y = 1.337e-06
z = 1.000e+08
Print all numbers with a precision of 0 in the standard format
x = 3e+01
y = 1e-06
z = 1e+08
Print all numbers in the standard format
x = 33.98
y = 1.33679e-06
z = 1e+08
Print all numbers in the standard format using a field of 15 spaces
x =           33.98 y =     1.33679e-06 z =           1e+08
*/
```

Figure 1.7. The program *Manipulators.cp*.

2. C/C++ Functions

Parameter passing. One of the significant additions to *C*++ allows a programmer to pass parameters by reference to a function without the explicit need to use pointers and the address operator. We start with the program in figure 2.1. The new *C*++ version for passing parameters is shown in figure 2.2. In this case, the symbol '&' placed after the parameter type (e.g., *int*&) indicates that parameter is passed by reference. Notice how much cleaner it is than the older *C* version: '&' is used once in the first line of the function and pointers, the dereference operator and the address operator are never used.

```
#include <stdio.h>

void Sum(int x, int y, int* zPtr);

int main (void) {
   int m = 10, n = 5, p;
   Sum(m, n, &p);
   printf("The value of p equals %d\n", p);
   return 0; }

void Sum(int x, int y, int* zPtr) {
   *zPtr = x + y; }

 /* CodeWarrior Pro 7 input/output:
The value of p equals 15
```

```
*/
```

Figure 2.1. The function *Sum*() with a parameter passed by reference in a *C* program.

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

void Sum(int x, int y, int& z);

int main (void) {
  int m = 10, n = 5, p;
  Sum(m, n, p);
  cout << "The value of p equals " << p << endl; }

void Sum(int x, int y, int& z) {
  z = x + y; }

/* CodeWarrior Pro 7 input/output:
The value of p equals 15
*/
```

Figure 2.2. The function *Sum*() with a parameter passed by reference in a *C++* program.

The ***signature*** of a function is the sequence of parameter types in its parameter list. For example,

```
float F(int x, float y);
int G(int x, float& y);
void H(int& x, float y);
```

all have the same signature *int*, *float*. Notice the function signature does not include its return value or depend on how its parameters are passed.

The phrase 'tonight's the night' can have a dramatically different meaning depending on the context in which it is used. *C++* abstracts this idea to the use of functions by allowing the function identifier to have different meanings depending on the signature of the function.

As an example, the function *Exchange*(),

```
void Exchange (double& x, double& y) {
  double z = x;
  x = y;
  y = z; }
```

exchanges two *double* parameters *x* and *y* but cannot be called with *int* arguments. The programmer, however, may provide two different versions of *Exchange*(), one for *int* parameters, shown below, and the one for *double* parameters shown above.

```
void Exchange (int& x, int& y) {
  int z = x;
  x = y;
  y = z; }
```

The compiler decides between the two versions of *Exchange*() by the type of the arguments the function is called with. This is an example of ***function overloading***, an idea we extend later with function templates. This specific example is illustrated in the program *FunctionOverload.cp* of figure 2.3.

Each version of a function should have a different signature. Thus, the three functions *Sum*() with declarations

```
int Sum(int m, int n);
int Sum(int& m, int& n);
void Sum(int& m, int n);
```

are not allowed since all three have the same signature. In effect, the signature becomes the context in which the function is called. The version picked for the call is the one whose signature matches the arguments in number and type. In cases where it makes sense, *C*++ has rules for picking a function even when an exact match does not occur. We leave a discussion of this to the references.

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

void Exchange (int& x, int& y);
void Exchange (double& x, double& y);

int main(void) {
   int m = 10, n = 20;
   cout << "m equals " << m << endl;
   cout << "n equals " << n << endl;
   Exchange(m, n);
   cout << "m equals " << m << endl;
   cout << "n equals " << n << endl;
   double x = 10.0, y = 20.0;
   cout << "x equals " << x << endl;
   cout << "y equals " << y << endl;
   Exchange(x, y);
   cout << "x equals " << x << endl;
   cout << "y equals " << y << endl;}

void Exchange (int& x, int& y) {
   int z = x;
   x = y;
   y = z; }

void Exchange (double& x, double& y) {
   double z = x;
   x = y;
   y = z; }

/* CodeWarrior Pro 7 input/output:
m equals 10
n equals 20
m equals 20
n equals 10
x equals 10
y equals 20
x equals 20
y equals 10
*/
```

Figure 2.3. The *C*++ program *FunctionOverload.cp*.

Because *C*++ allows function overloading, the function *abs*() has been overloaded to work for all standard variable types, not just integer types as in *C*. It can be used whenever

```
using namespace std;
```

is used.

A parameter called by value in *C*++ can be given a default argument. This is useful when a function is frequently called with the same value since default values don't have to be included in the function call. Consider the function *Sum*() shown below.

```
float Sum (float x, float y) {
   return x + y; }
```

If the declaration of the function is given as

```
float Sum (float x, float y = 3.14);
```

the statement given by

```
float z = Sum (1.0, 2.0);
```

gives the value 3.0 to *z* whereas the statement

```
float z = Sum (1.0);
```

gives the value 4.14 to *z*.

Only parameters at the end of the parameter list can be given default values. For example, if the function parameter list consists of

```
(float x, float y, float z)
```

then *z* can be given a default value, *y* and *z* can be given default values, or *x*, *y* and *z* can be given default values. Note the default values are given only in the declaration of the function, not in its definition. These details are illustrated in the program *DefaultValues.cp* shown in figure 2.4.

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

void Print (int x = 1, int y = 2, int z = 3);      //default values are given only in the function declaration

int main (void) {
   int u = 10, v = 20, w = 30;
   Print();
   Print(u);
   Print(u, v);
   Print(u, v, w); }

void Print (int x, int y, int z) {
   cout << "x = " << x << ", y = " << y << ", z = " << z << endl;}

/* CodeWarrior Pro 7 input/output:
x = 1, y = 2, z = 3
x = 10, y = 2, z = 3
x = 10, y = 20, z = 3
x = 10, y = 20, z = 30
*/
```

Figure 2.4. The program *DefaultValues.cp*.

Default values should be used with care since their use can make a program hard to read. We will use them frequently when we take up *C*++ objects later.

Finally, when the qualifier *inline* is used before a function identifier, the compiler tries to insert the code from the function definition at each place the function is called. The usual result is a longer program that runs faster than the non-inline version because the overhead of calling the function is avoided. (The reader will gain more insight into the overhead of calling a function later when we examine the stack.) The *inline* qualifier should be used only for short, simple functions.

3. The for-statement

In *C++*, the initialization section of a for-loop can be used to define variables that are local to the for-loop. The program *ForLoop.cp* in figure 3.1 illustrates this feature. In this case, a syntax error occurs if a statement uses the *int* variable *i* after the for-statement.

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

int main (void) {
  int sum = 0;
  int n;  cout << "n = "; cin >> n;
  for (int i = 1; i <= n; i++)            // the definition of i is local to the for-loop
     sum += i;
  cout << "sum = " << sum << endl;}


/* CodeWarrior Pro 7 input/output:
n = 10
sum = 55
*/
```

Figure 3.1. The program *ForLoop.cp*.

The declaration of the *C++* function *f*() that returns a *double* value with a *double* parameter *x* passed by constant reference in *C++* looks like

```
double f(const double& x);
```

Even though the parameter *x* is passed by reference, function calls in this case can use explicit values for the argument, say

```
double y = f(3.0);
```

In this case, a temporary *double* variable is created with the value 3.0 and the address of this temporary variable is passed to *f*(). Even though this is somewhat inefficient, it still makes sense in the constant reference situation since *f*() can't change the value 3.0 in the temporary variable.

4. Control and Repetition

In *C++* the keywords *and*, *or* and *not* can be substituted for the '&&', '||' and '!', respectively, as illustrated in the program *MoreLogicOperations.cp* of figure 4.1. Notice how parentheses are used to ensure the Boolean operators are evaluated first.

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

int main (void) {
   int m;
   cout << "m = ";  cin >> m;
   int n;
   cout << "n = ";   cin >> n;
   cout << "(m and n) equals " <<  (m and n) << endl;
   cout << "(m or n) equals " <<  (m or n) << endl;
   cout << "(not m) equals " << (not m) << endl; }

/* CodeWarrior Pro 7 input/output:
m = 0
n = 3
(m and n) equals 0
(m or n) equals 1
(not m) equals 1
*/
```

Figure 4.1. The program *MoreLogicOperators.cp*.

The *C++* statements

```
   float x = 17.3;
   float& y = x;
```

define *y* as a ***reference*** or ***alias*** to *x*. Reference variables must be initialized when they are declared and cannot be assigned afterwards to reference another variable. Reference variables are used mainly to pass parameters by reference. In this section, we examine a second use of reference variables.

Suppose we wanted to write a function *Max*() that returns the maximum of two float variables *x* and *y* with the call

```
   cout << Max(x, y) << endl;
```

The shortest version of *Max*(), which uses '?', is shown below.

```
float Max(const float& x, const float& y) {
   return (x > y ? x : y);}
```

However, suppose we wrote *Max*() so it returned a reference to the maximum of the two float parameters. This can be done as shown below.

```
float& Max(float& x, float& y) {
   return (x > y ? x : y);}
```

When written this way, the call to *Max*() can also be used to change the maximum of the two arguments, as with

```
   Max(u, v) = 17.3;
```

See the program *Reference.cp* in figure 4.2.

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

float& Max(float& x, float& y);

int main () {
   float u = 1.0, v = 2.0;
   Max(u, v) = -1.0;
   cout << "u = " << u << endl;
   cout << "v = " << v << endl;
   Max(u, v) = 3.0;
   cout << "u = " << u << endl;
   cout << "v = " << v << endl; }

float& Max(float& x, float& y) {
   return (x > y ? x : y); }

/* CodeWarrior Pro 7 input/output:
u = 1
v = -1
u = 3
v = -1
*/
```

Figure 4.2. The program *Reference.cp*.

Sentinels are an easy way to do some calculations, but, in other cases, their use seems awkward. For example, suppose we want to calculate the sum of the integers entered on a line from the console window including a '-1'. Suppose also we were willing to assume: (i) the numbers are separated by one or more spaces, (ii) a carriage return is typed immediately after the last digit of the last number and (iii) an empty line consists of just a carriage return. We could use the *peek*() member function of the *iostream* class with *cin* as shown in the program *EndOfLineWhile.cp*. (See figure 4.3). The function *peek*() returns the value of the next character to be read. As long as that character is not a carriage return '\n', we know we can read another number from that line. In particular, the Boolean expression

```
(cin.peek() != '\n')
```

is *true* as long as the next character to be read is not a carriage return.

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

int main (void) {
   int sum = 0;
   while (cin.peek() != '\n') {
      int  nextInteger;
      cin >> nextInteger;
      sum += nextInteger; }
   cout << "sum = " << sum << '\n'; }

/* CodeWarrior Pro 7 input/output:
1 2 3 4 5
sum = 15
*/
```

Figure 4.3. The program *EndOfLineWhile.cp*.

A line consisting of just a carriage return can also be used to designate the termination of the input. For example, assume an *m* by *n* matrix *A* of floating-point numbers will be read in from the console window in the following way. The first line is used for the first row of *A*, the second line is used for the second row of *A*, and so on. On each line, the row components are separated by one or more spaces with a return character coming immediately after the last digit of the last row entry. A blank line, that is, a line with just a return character, terminates the input. The function *InputMatrix*() shown in figure 4.4 does this with the call

```
InputMatrix(m, n, A);
```

The identifier *columnSize* used in the header of *InputMatrix*() is a global integer constant.

```
void InputMatrix(int& m, int& n, float A[][columnSize]) {
  m = 0;
  while (cin.peek() != '\n') {
    n = 0;
    while (cin.peek() != '\n')
      cin >> A[m][n++];
    m++;
    cin.get(); }  // remove the return character from the end of an input row
  cin.get(); }     // remove the return character from empty line
```

Figure 4.4. The function *InputMatrix*().

The while-loop version of *InputMatrix*() can be shortened somewhat with for-loops as shown in figure 4.5.

```
void InputMatrix (int& m, int& n, float A[][10]) {
  for (m = 0; cin.peek() != '\n'; m++) {
    for (n = 0; cin.peek() != '\n'; n++)
      cin >> A[m][n];
    cin.get(); }  // remove the return character from the end of an input row
  cin.get(); }     // remove the return character for the empty line
```

Figure 4.5. The for-loop version of *InputMatrix*().

5. Pointers, Arrays and Strings

The function *Maximum*() is shown below in figure 5.1. It returns a pointer to the maximum of two *float* values passed at the addresses **xPtr* and **yPtr*.

```
#include <stdio.h>

float* Maximum(float* xPtr, float* yPtr);

int main (void) {
  float x = 17.1, y = 3.9;
  printf("The maximum value of x and y is %f\n", *Maximum(&x, &y));
  *Maximum(&x, &y) = 0;
  printf("The value of x is %f\n", x);
  return 0; }

float* Maximum(float* xPtr, float* yPtr) {
  if (*xPtr >= *yPtr) return xPtr;
```

```
   return yPtr;}

/* CodeWarrior Pro 8.2 input/output:
The maximum value of x and y is 17.100000
The value of x is 0.000000
*/
```

Figure 5.1. The *C*-function *Maximum*().

Using *C*++, *Maximum*() can be written to return a reference as shown in figure 5.2. It seems much cleaner than its *C* counterpart.

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

float& Maximum(float& x, float& y);

int main (void) {
   float x = 17.1, y = 3.9;
   cout << "The maximum value of x and y is " << Maximum(x, y) << endl;
   Maximum(x, y) = 0;
   cout << "The value of x is " << x << endl;}

float& Maximum(float& x, float& y) {
   if (x >= y) return x;
   return y;}

/* CodeWarrior Pro 8.2 input/output:
The maximum value of x and y is 17.1
The value of x is 0
*/
```

Figure 5.2. The *C*++-function *Maximum*().

Using *C*-strings is awkward. They can't be assigned or compared without using functions from the *C* string library and their size is always limited to a value that is fixed before the program is run. The *C*++ *string* class removes these difficulties. We will see how this is done later; for now, we just examine its basic use.

The following statement defines a *string* variable (or object) *x*

```
   string  x;
```

which gives *x* the value of the empty string "". If *x* and *y* are two *string* objects, we can assign them with the statement

```
   x = y;
```

We can see if *x* and *y* are equal or not equal with the Boolean expressions

```
   (x == y)

   (x != y)
```

We can compare *x* and *y* with the Boolean expressions

```
   (x > y)

   (x >= y)

   (x < y)

   (x <= y)
```

In this case, (x > y) is *true* if *x* comes after *y* as defined earlier for *C*-strings.

The statement

```
cout << x << endl;
```

will output the string *x* to the console window followed by a carriage return. The statement

```
cin >> x;
```

inputs the first string of non white-space characters as *x*. Because white-space characters are skipped over for the input, you cannot input an empty string for *x* with this statement.

The length of the string *x* is returned from the function *x.length*(). The function *getline*() will extract the complete line from the input stream, discarding the return character, and assign it to the string *x* with the call

```
getline(cin, x);
```

Similarly, the function *ignore*() will discard the next 80 characters, or the characters up to and including a carriage return, from the input stream with the call

```
cin.ignore(80, '\n');
```

These functions and basic operations are illustrated in the program *CPlusPlusStrings.cp* of figure 5.3 where the *C++ string* class is included in the program with the statement

```
#include <string>
```

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

int main () {
  string x, y;
  cin >> x;
  cout << x << endl;
  cout << x.length() << endl;
  cin >> y;
  cout << y << endl;
  cin.ignore(80, '\n');
  getline(cin, x);
  cout << x << endl;
  cout << "(x == y) = " << (x == y) << endl;
  cout << "(x != y) = " << (x != y) << endl;
  cout << "(x > y) = " << (x > y) << endl;
  cout << "(x >= y) = " << (x >= y) << endl;
  cout << "(x < y) = " << (x < y) << endl;
  cout << "(x <= y) = " << (x <= y) << endl; }

/* CodeWarrior Pro 8.2 input/output:
jack
jack
4
jane is here too
jane
jane is here too
jane is here too
(x == y) = 0
(x != y) = 1
(x > y) = 1
(x >= y) = 1
(x < y) = 0
(x <= y) = 0

      jack
jack
4
```

```
   jane
jane
   jane is here too
   jane is here too
(x == y) = 0
(x != y) = 1
(x > y) = 0
(x >= y) = 0
(x < y) = 1
(x <= y) = 1
*/
```

Figure 5.3. The program *CPlusPlusStrings.cp*.

6. User Defined Types.

When you are writing in *C++*, the definitions made with *struct*, *enum* and *union* are automatically variable types. This is illustrated in the program *struct.cp* of figure 6.1.

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

struct Student {
   string name;
   string address;
   int    age; };

int main () {
   Student student, anotherStudent, * studentPtr;
   student.name = "Melissa Ethridge";
   student.address = "1040 Main, Kansas City, KA, 32456";
   student.age = 31;
   anotherStudent = student;
   cout << anotherStudent.name << endl;
   studentPtr = &student;
   cout << studentPtr->name << endl;
   cout << sizeof(student) << endl; }

/* CodeWarrior Pro 8 input/output:
Melissa Ethridge
Melissa Ethridge
12
*/
```

Figure 6.1. The program *struct.cp*.

Furthermore, the programmer can overload the stream operators '<<' and '>>' to work for all three types. We will examine how you do this in detail later. A preview of this is given in figure 6.2.

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

enum Days {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};

ostream& operator<< (ostream& out, const Days& x);
istream& operator>> (istream& in, Days& x);
```

```
int main (void) {
   Days x = Tuesday, y;
   cout << "x = " << x << endl;
   cin >> x >> y;
   cout << "x = " << x << ";  y = " << y << endl;}

ostream& operator<< (ostream& out, const Days& x) {
   if (x == Monday)
      out << "Monday";
   else if (x == Tuesday)
      out << "Tuesday";
   else if (x == Wednesday)
      out << "Wednesday";
   else if (x == Thursday)
      out << "Thursday";
   else if (x == Friday)
      out << "Friday";
   else if (x == Saturday)
      out << "Saturday";
   else if (x == Sunday)
      out << "Sunday";
   return out;}

istream& operator>> (istream& in, Days& x) {
   string s;
   in >> s;
   if (s == "Monday")
      x = Monday;
   else if (s == "Tuesday")
      x = Tuesday;
   else if (s == "Wednesday")
      x = Wednesday;
   else if (s == "Thursday")
      x = Thursday;
   else if (s == "Friday")
      x = Friday;
   else if (s == "Saturday")
      x = Saturday;
   else if (s == "Sunday")
      x = Sunday;
   return in; }

/* CodeWarrior Pro 8.2 input/output:
x = Tuesday
Monday   Friday
x = Monday;   y = Friday
*/
```

Figure 6.2. Adding i/o capabilities to the *enum* type.

In *C++ typedef* creates an alias to an existing type. For example, it can be used to simplify definitions involving pointers. The statement

```
typedef  float*   FloatPtr;
```

eliminates multiple occurrences of '*' so, for example, instead of writing

```
float* aPtr, * bPtr;
```

one can now write

```
FloatPtr aPtr, bPtr;
```

which eliminates the second '*'. The above *typedef* statement also simplifies the syntax for passing a pointer by constant value as illustrated in the program of figure 6.3. (If the two statements that are commented out are uncommented, a syntax error will occur.) It is a good exercise to write this program without the *typedef* statement.

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

typedef float* FloatPtr;

float f(const FloatPtr xPtr);
float f(const FloatPtr xPtr) {
//  float z = 3;
//  xPtr = &z;
   return (*xPtr) * (*xPtr);}

int main(void) {
   FloatPtr xPtr, yPtr;
   xPtr = new float(1.0);
   yPtr = new float(2.0);
   cout << *xPtr << endl;
   cout << *yPtr << endl;
   cout << f(yPtr) << endl; }

/* CodeWarrior Pro 9.0 output:
1
2
4
*/
```

Figure 6.3. A simple use of *typedef*.

Another use of *typedef* is illustrated in the program *typedef.cp* of figure 6.8.4 where *Function* is a pointer-type that points to functions that take a *float* argument and return a *float* value. Since the identifier of a function is interpreted as an address, the operator '&' is not used when *h* is assigned to a function.

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

typedef float (*Function)(float x);

float f(float x);
float g(float x);

int main () {
   Function h = f;
   cout << h(2) << endl;
   h = g;
   cout << h(2) << endl; }

float f(float x) {
   return x * x; }

float g(float x) {
   return x * x * x;}

/* CodeWarrior Pro 8 input/output:
4
8
*/
```

Figure 6.4. The program *typedef.cp*.

A final use of *typedef* is shown in the program *rowPointer.cp* of figure 6.5. The statements

```
const int capacity = 10;
typedef float Matrix[capacity][capacity];
typedef float Row[capacity];
```

define the types *Matrix* and *Row* that are used in the program to access the individual elements of the *Matrix* object *A*. As we stated earlier, it is rarely necessary to use this type of access to the components of *A* and we will not pursue it further in these notes.

```
#include <iostream>
#include <iomanip>
using namespace std;

const int capacity = 10;
typedef float Matrix[capacity][capacity];
typedef float Row[capacity];

int main(void) {
  Matrix A;
  for (Row* row = A; row != A + capacity; row++)
    for (float* ptr = row[0];  ptr != row[0] + capacity; ptr++)
      *ptr = (row - A) * capacity + (ptr - row[0]);

  for (int i = 0; i < capacity; i++) {
    for (int j = 0; j < capacity; j++)
      cout << setw(4) << A[i][j];
    cout << endl; }}

/* CodeWarrior Pro 7 input/output:
   0   1   2   3   4   5   6   7   8   9
  10  11  12  13  14  15  16  17  18  19
  20  21  22  23  24  25  26  27  28  29
  30  31  32  33  34  35  36  37  38  39
  40  41  42  43  44  45  46  47  48  49
  50  51  52  53  54  55  56  57  58  59
  60  61  62  63  64  65  66  67  68  69
  70  71  72  73  74  75  76  77  78  79
  80  81  82  83  84  85  86  87  88  89
  90  91  92  93  94  95  96  97  98  99
*/
```

Figure 6.5. The program *rowPointer.cp*.

In *C++*, the operators *new* and *delete* replace the functions *malloc*() and *free*(). In fact, since only pointers of the same type can be assigned in *C++*, *malloc*() can't be used without a cast since it returns a pointer of type *void*. With *new*, the execution of the statement

```
Node* ptr = new Node;
```

takes memory from the heap for a *Node* variable defined in figure 6.7. The components of **ptr* can be given values using the arrow operator. For example,

```
ptr->name = "terry";
```

puts the string "terry" in the name-component of **ptr*. The statement

```
delete ptr;
```

returns the memory used by *ptr* to the heap. Since repeated uses of *delete* on the same pointer causes unpredictable results, many programmers use

```
delete ptr;
ptr = 0;
```

since using *delete* on a pointer with value 0 has no effect. (The value 0 replaces *NULL* in *C*++.)

In general, one should not mix *new-delete* with *malloc-free*. For example, it is an error to use *free*() with a pointer assigned a value with *new* or *delete* with a pointer assigned a value by *malloc*(). The best thing to do is to avoid *malloc*() and *free*() altogether unless there is a compelling reason to use them.

The statements

```
int n = 100;
double* ptr = new double[n];
```

creates a dynamic array with *n* components where the memory for the array comes from the heap. The statements

```
delete [] ptr;
ptr = 0;
```

returns the memory for the array to the free store. The programmer must be careful to use '[]' in the *delete* statement only with arrays.

Despite the fussiness with the *delete* statement with arrays, using *C*++ with linked lists allows the programmer to avoid the use of handles. For example, the program of figure 6.7 is rewritten in *C*++ in figure 6.6. Notice how *Node** can be treated as a type so *firstPtr* and *lastPtr* can be passed by reference without the use of handles. Even a hard-core *C*-programmer has to admit this is simpler. Finally, notice 0 replaces *NULL* in *C*++.

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

struct Node {
   string name;
   Node* nextPtr;};

Node* RandomInsertionPoint(Node* firstPtr);
void InsertNode(Node*& firstPtr, Node*& lastPtr,  Node* insertionPtr, Node* ptr);
void CreateLinkedList(Node*& firstPtr, Node*& lastPtr);
void OutputList(Node* firstPtr);
Node* FindName(Node* firstPtr, string name);
void SearchNames(Node* firstPtr);
void DeleteNode(Node*& firstPtr, Node*& lastPtr, Node* selectPtr);
void DeleteNames(Node*& firstPtr, Node*& lastPtr);

int main (void) {
   Node* firstPtr = 0, * lastPtr = 0;                 // initialize pointers for an empty linked list
   CreateLinkedList(firstPtr, lastPtr);
   OutputList(firstPtr);
   SearchNames(firstPtr);
   DeleteNames(firstPtr, lastPtr);
   cout << "The program is over" << endl;
   return 0;}

Node* RandomInsertionPoint(Node* firstPtr) {   // picks a random insertion point for a new node
```

```
   for (Node* ptr = firstPtr; ptr != 0; ptr = ptr->nextPtr)
      if (rand() % 2) return ptr;
   return 0; }

void InsertNode(Node*& firstPtr, Node*& lastPtr, Node* insertionPtr, Node* ptr) {
   if (insertionPtr == 0) {                                // insert *ptr after *insertionPtr
      ptr->nextPtr = firstPtr;
      firstPtr = ptr; }
   else {
      ptr->nextPtr = insertionPtr->nextPtr;
      insertionPtr->nextPtr = ptr; }
   if (insertionPtr == lastPtr )
      lastPtr = ptr; }

void CreateLinkedList(Node*& firstPtr, Node*& lastPtr) {
   cout << "Input the names to add to the list:\n";
   while (cin.peek() != '\n') {
      Node* ptr = new Node;          // create new node
      cin >> ptr->name;              // get name for new node
      cin.get();
      InsertNode(firstPtr, lastPtr, RandomInsertionPoint(firstPtr), ptr);}
   cin.get(); }                      // remove return character from empty line

void OutputList(Node* firstPtr) {    // output the strings in the linked list from beginning to end
   cout << "The names in the linked list are:\n";
   for (Node* ptr = firstPtr; ptr != 0; ptr = ptr->nextPtr)
      cout << ptr->name << endl;
   cout << endl; }

Node* FindName(Node* firstPtr, string name) { // returns a pointer to first node containing 'name'
   for (Node* ptr = firstPtr; ptr != 0; ptr = ptr->nextPtr)
      if (name == ptr->name)
         return ptr;
   return 0; }


void SearchNames(Node* firstPtr) {
   cout << "Input the names to find in the list:\n";
   while (cin.peek() != '\n') {
      string s;
      cin >> s;
      cin.get();
      if (FindName(firstPtr, s) != 0)
         cout << s << " is in the list\n";
      else
         cout << s <<" is not in the list\n";}
   cin.get(); }

void DeleteNode(Node*& firstPtr, Node*& lastPtr, Node* selectPtr) {
   Node* previousPtr = 0;
   if (selectPtr == 0) return;       // nothing happens if selectPtr equals 0.
   if (selectPtr == firstPtr)
      firstPtr = firstPtr->nextPtr;
   else {
      previousPtr = firstPtr;
      while (previousPtr->nextPtr != selectPtr)
         previousPtr = previousPtr->nextPtr;
      previousPtr->nextPtr = selectPtr->nextPtr; }
   if (selectPtr == lastPtr)
      lastPtr = previousPtr;
   delete selectPtr;}

void DeleteNames(Node*& firstPtr, Node*& lastPtr) {
   cout << "Input the names to delete from the list:\n";
   while (cin.peek() != '\n') {
      string s;
      cin >> s;
      cin.get();
      DeleteNode(firstPtr, lastPtr, FindName(firstPtr, s));
      OutputList(firstPtr); }}
```

```
/* CodeWarrior Pro 8.3 input/output:
Input the names to add to the list:
bob
ted
carol
alice

The names in the linked list are:
ted
alice
bob
carol

Input the names to find in the list:
ted
ted is in the list
joe
joe is not in the list
carol
carol is in the list

Input the names to delete from the list:
ted
The names in the linked list are:
alice
bob
carol

carol
The names in the linked list are:
alice
bob

pete
The names in the linked list are:
alice
bob


The program is over
*/
```

Figure 6.6. The program *LinkedListOperations.cp*.

A ***doubly linked list*** has two links, called *leftPtr* and *rightPtr*, to go with the illustration in figure 6.7. In particular, *leftPtr* points to the node on the left and *rightPtr* points to the node on the right. The right-most node has '0' for the *rightPtr* value and the left-most node has '0' for the *leftPtr* value. Mimicking what we did for singly linked lists, we will use the following generic *Node* class for our examples.

```
struct Node {
  string name;
  Node* leftPtr, * rightPtr; } ;
```

For this *Node* definition, the name-component is shown in gray in figure 6.7.

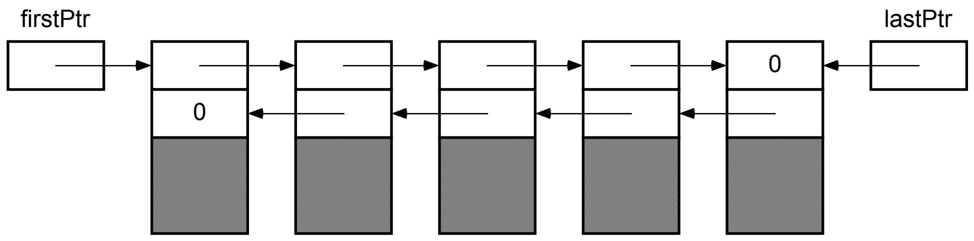

Figure 6.7. A doubly linked list.

The advantage of a doubly linked list over a singly linked list is that deletion is faster. In particular, suppose *selectPtr* points to the node to be deleted. With a singly linked list, we have to find a pointer to the previous node. The time for this calculation depends on how many nodes are to the left of **selectPtr*. The segment below for a doubly linked list avoids this calculation, so the time to delete a node in a doubly linked list is independent of the position of the node in the list.

```
if (firstPtr == selectPtr)
  firstPtr = selectPtr->rightPtr;
else
  selectPtr->leftPtr->rightPtr = selectPtr->rightPtr;
if (lastPtr == selectPtr)
  lastPtr = selectPtr->leftPtr;
else
  selectPtr->rightPtr->leftPtr = selectPtr->leftPtr;
```

The above segment would usually be followed with

```
delete selectPtr;
```

to return the node's memory to the free store.

If *insertionPtr* points to a node in the list and *newPtr* points to a new node not in the list, the segment below will insert the new node to the *right* of the node **insertionPtr*. If *insertionPtr* has the value 0, the new node will be put at the start of the linked list. (Recall this is what we did with a singly linked list.) Nothing is done in this segment to the new node's information components; we assume they've been assigned beforehand.

```
if (insertionPtr == 0) {
  newPtr->leftPtr = 0;
  newPtr->rightPtr = firstPtr;
  if (lastPtr == 0)
    lastPtr = newPtr;
  else
    firstPtr->leftPtr = newPtr;
  firstPtr = newPtr;}
else {
  newPtr->rightPtr = insertionPtr->rightPtr;       // a
  newPtr->leftPtr = insertionPtr;                  // b
  if (insertionPtr->rightPtr != 0)                 // c
    insertionPtr->rightPtr->leftPtr = newPtr;
  else
    lastPtr = newPtr;
  insertionPtr->rightPtr = newPtr; }                    // d
```

The above statements labeled 'a', 'b', 'c' and 'd' are illustrated in figure 6.8.

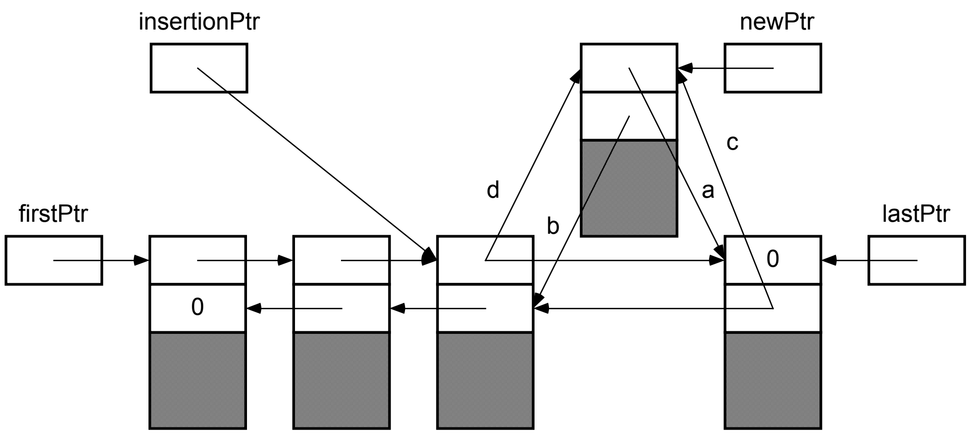

Figure 6.8. Insertion in a doubly linked list.

Finally, the program of figure 6.6 is repeated in figure 6.9 for doubly linked lists with the *Node* definition used for the above discussion.

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

struct Node {
    string name;
    Node* leftPtr, * rightPtr; };

void InsertNode(Node*& firstPtr, Node*& lastPtr, Node* insertionPtr, Node* ptr);
Node* InsertionPoint(Node* firstPtr);
void CreateList(Node*& firstPtr, Node*& lastPtr);
void OutputList(Node* firstPtr);
bool FindName(Node* firstPtr, string name);
void SearchNames(Node* firstPtr);
void DeleteNode(Node*& firstPtr, Node*& lastPtr, string name);
void ModifyList(Node*& firstPtr, Node*& lastPtr);

int main () {
    Node* firstPtr = 0, * lastPtr = 0;              // initialize for an empty linked list
    CreateList(firstPtr, lastPtr);
    OutputList(firstPtr);
    SearchNames(firstPtr);
    ModifyList(firstPtr, lastPtr);
    cout << "The program is over" << endl; }

void InsertNode(Node*& firstPtr, Node*& lastPtr, Node* insertionPtr, Node* ptr) {
    if (insertionPtr == 0) {
        ptr->leftPtr = 0;
        ptr->rightPtr = firstPtr;
        if (lastPtr == 0)
            lastPtr = ptr;
        else
            firstPtr->leftPtr = ptr;
        firstPtr = ptr; }
    else {
        ptr->rightPtr = insertionPtr->rightPtr;
        ptr->leftPtr = insertionPtr;
        if (insertionPtr->rightPtr != 0)
            insertionPtr->rightPtr->leftPtr = ptr;
        else
            lastPtr = ptr;
        insertionPtr->rightPtr = ptr; }}

Node* InsertionPoint(Node* firstPtr) {
    for (Node* ptr = firstPtr; ptr != 0; ptr = ptr->rightPtr)
        if (rand() % 2) return ptr;
    return firstPtr; }

void CreateList(Node*& firstPtr, Node*& lastPtr) {
    cout << "Input the names to be added to the list:" << endl;
    while (cin.peek() != '\n') {
        Node* ptr = new Node;       // create new node
        cin >> ptr->name;           // put data in node
        cin.get();                  // remove return character
        Node* insertionPtr = InsertionPoint(firstPtr);
        InsertNode(firstPtr, lastPtr, insertionPtr, ptr); }    // insert node to the right of insertionPtr
    cin.get();}

 void OutputList(Node* firstPtr) {
    cout << "The names in the linked list are:" << endl;
    for (Node* ptr = firstPtr; ptr != 0; ptr = ptr->rightPtr)
        cout << ptr->name << endl;
    cout << endl; }

bool FindName(Node* firstPtr, string name) {
    for (Node* ptr = firstPtr; ptr != 0; ptr = ptr->rightPtr)
        if (name == ptr->name)
            return true;
    return false; }

void SearchNames(Node* firstPtr) {
    cout << "Input the names to find to find in the list:" << endl;
    while (cin.peek() != '\n') {
        string s;  cin >> s;
        cin.get();
        if (FindName(firstPtr, s))
            cout << s << " is in the list" << endl;
        else
            cout << s << " is not in the list" << endl; }
    cin.get(); }

void DeleteNode(Node*& firstPtr, Node*& lastPtr, string name) {
    Node* selectPtr = 0;
    for (Node* ptr = firstPtr; ptr != 0; ptr = ptr->rightPtr)
        if (ptr->name == name)
            selectPtr = ptr;
    if (selectPtr) {
        if (firstPtr == selectPtr)
            firstPtr = selectPtr->rightPtr;
        else
            selectPtr->leftPtr->rightPtr = selectPtr->rightPtr;
        if (lastPtr == selectPtr)
            lastPtr = selectPtr->leftPtr;
        else
            selectPtr->rightPtr->leftPtr = selectPtr->leftPtr;
        delete selectPtr; }}

void ModifyList(Node*& firstPtr, Node*& lastPtr) {
    cout << "Input the names to be deleted from list: " << endl;
```

```
  while (cin.peek() != '\n') {
    string name;                    // name = "" each time through loop
    cin >> name;
    cin.get();
    DeleteNode(firstPtr, lastPtr, name);
    OutputList(firstPtr); }}


/* CodeWarrior Pro 8.3 input/output:
Input the names to be added to the list:
bob
ted
carol
alice
joe

The names in the linked list are:
bob
joe
alice
ted
carol

Input the names to find to find in the list:
joe
joe is in the list
bill
bill is not in the list
carol
carol is in the list

Input the names to be deleted from list:
bob
The names in the linked list are:
joe
alice
ted
carol

carol
The names in the linked list are:
joe
alice
ted


The program is over
*/
```

Figure 6.9. Basic list operations with a doubly linked list.

C++ Language Coding Style Guidelines

EE322C – Version 1.1

1 Introduction

Writing code that is easy for others to read and modify is an important part of programming. Learning to program does not just mean getting the program to work. The program should be clear, as simple as possible, easy to read and understand. It should be decomposed as the problem itself is decomposed and solved. In the "real world" and on the later assignments you simply will not be able to write all the code at once. You will have to code in pieces and if you do not use good style you will spend a large amount of time asking yourself, "what was I trying to do here?"

This coding style guide is a simplified version of one that has been used with good success both in industrial practice and for other college computing courses.

A style guide is a set of mandatory requirements for layout and formatting. Uniform style makes it easier for you to read code from your instructor and classmates. It is also easier for your instructor and your grader to grasp the essence of your programs quickly. A style guide makes you a more productive programmer because it *reduces gratuitous choice.* If you don't have to make choices about trivial matters, you can spend your energy on the solution of real problems.

In these guidelines, several constructs are plainly outlawed. That doesn't mean that programmers using them are evil or incompetent. It does mean that the constructs are not essential and can be expressed just as well or even better with other language constructs.

If you already have programming experience, in C++ or another language, you may be initially uncomfortable at giving up some fond habits. However, it is a sign of professionalism to set aside personal preferences in minor matters and to compromise for the benefit of the community.

These guidelines are necessarily somewhat dull. They also may mention features that you may not yet have seen in class. Here are the most important style highlights:

- Tabs are set every four spaces for indented blocks of code.
- Variable and function names are lowercase, with occasional uppercase letters or underscore characters in the middle for run together words.
- Class names start with an Uppercase letter
- Constant names are UPPERCASE, with an occasional UNDER_SCORE.
- There are spaces after keywords and surrounding binary operators.
- Braces must line up horizontally or vertically.
- No magic numbers may be used.
- Every function must have a block header comment explaining its purpose and use.
- Use single line comments to document key logic structures within functions
- At most 50 lines of non comment code may be used per function.

2 Source Files

Each C++ program is a collection of one or more source files. The executable program is obtained by compiling and linking these files. Organize the material in each file as follows. The overall ordering of things in the main source file are:

File comment
System includes

User includes
prototypes for functions in this file
global variable declarations
main function
other functions

The overall ordering of things in a non-class header file are:
#ifndef ... (Macro guard)
#define ...
File comment
System includes
User includes
Constants
Structures
Prototypes
#endif

Each file should include those header files that are NECESSARY for it to compile. Include header files based upon what is in that file. Do not base it upon what will be in the file after #includes are included.

The comment explaining the purpose of this file should be in the block comment format. Between `/* .. */`, and clearly ientifying the `author` and `version` information.

```
/* This file contains the class to manipulate widgets.
  Solves EE322C homework assignment #3
  author Harry Hacker
  version 1.01 1997-02-15 */
```

Never #include a .cpp file (or any file that contains functions definitions).

Inline functions should go in a separate file that is included at the bottom of the header file containing their prototypes. Preferred extensions for this file are ".ipp", ".inl", or ".inline". Do NOT put them in a file with extension ".cpp"

Template functions should go in a separate file that is included at the bottom of the header file containing their prototypes. Preferred extensions for this file are ".ipp", ".tpp", or ".template"

3 Classes

The overall ordering of things in a class header file is:
#ifndef ... (Macro guard)
#define ...
File comment
Function comments
System includes
User includes
Class definition
Non-member function prototypes
#endif

Each class should be preceded by a class comment explaining the purpose of the class.

Within each subsection, first list all public features, then all protected features, then all private features.

For a good OOD, you should make all data members and helper/utility functions private. The interface functions have to be public.

Statements granting friendship should go above the public section immediately after the opening curly brace.

Data in the member initializer lists should be in the order that they are declared inside the class.

You should code a constructor, destructor, copy constructor, and assignment operator for any class that contains dynamic memory.

If a struct/class is merely a helper struct/class, you should encapsulate it inside the other class.

Functions belonging to a templated class should go in separate file that is included immediately before the #endif in your class header file. Again the preferred extensions are: .tpp, .tem, or .template

All const variables are normally private. (However, instance variables of a `private` nested class may be public.) Functions and final variables can be either public or private, as appropriate.

4 Functions

Function names should indicate what they do or what they return. Each function shall have a comment explaining what it does. At a minimum, comments for functions will list pre and post conditions. The things that must be true when calling the function and the things that will be true when the function is done - assuming the preconditions were met. The pre and post conditions should be stated in precise terms and in terms of the public interface of the class whenever possible. Each function (except for `main`) will start with a block comment.

```
/*  This function converts calendar date into Julian day.
Note: This algorithm is from Press et al., Numerical Recipes    in C, 2nd ed.,
Cambridge University Press, 1992
param1 is the day of the date to be converted
param2 is the month of the date to be converted
param3 is the year of the date to be converted
the function returns the Julian day number that begins at noon of the given
calendar date. */

int dat2jul(int day, int month, int year)
{     . . .
}
```

Functions must have at most 50 lines of code. The function signature, comments, blank lines, and lines containing only braces are not included in this count. This rule forces you to break up complex computations into separate functions. Leave a blank line after every function.

The return type of the main function should be int.

Leave a blank line after every function definition.

Function macros should not be used.

Put names and data types for all formal parameters in the function prototypes.

Calculation functions should not have input or output statements in them.

Argument variables that are to be modified by the function called should be passed by reference instead of using the C method which used the address and dereferencing operators.

Single line comments (// comment) are used within a function (or other component) to describe the logic, or to explain the meaning of a statement or a following section of code.

5 Variables and Constants

Define each variable just before it is used for the first time. For variables that need to be accessible outside of a block as well as inside, define them right before the block (e.g. xold if used both inside and outside of the while loop).

```
{double xold = function_call ( );
 bool more = false;
 while (more)
 { double xnew = (xold + a / xold) / 2; // OK
 . . .
 }    . . .
}
```

Do not define all variables at the beginning of a block:

```
{double xold; // Don't
 double xnew;
 bool more;    . . .
}
```

Do not define two variables on the same line:

```
int dimes = 0, nickels = 0; // Don't
```

Instead, use two separate definitions:

```
int dimes = 0; // OK
int nickels = 0;
```

In C++, constants must be defined with the keyword const. Do not use the preprocessor command #define to define a constant. If the constant is used by multiple functions, declare it as `static`. It is a good idea to define static constants as `private` if no other class has an interest in them.

Do not use *magic numbers!* A magic number is a numeric constant embedded in code, without a constant definition. Any number except -1, 0, 1, and 2 is considered magic. Imagine trying to figure out what is meant by statements like:

```
if (p.getX() < 300) // Don't
while (i ! = 1776)  // Don't
```

Use `const` variables instead:

```
const double WINDOW WIDTH = 300; . . .
if (p.getX() < WINDOW_WIDTH) // OK
```

Even the most reasonable cosmic constant is going to change one day. You think there are 365 days per year? Your customers on Mars are going to be pretty unhappy about your silly prejudice. Make a constant

```
const int DAYS PER YEAR = 365;
```

so that you can easily produce a Martian version without trying to find all the 365s, 364s, 366s, 367s, and so on, in your code.

Pointer variable declarations should be done using the following style: double* p;

Dynamically allocated memory must always be deallocated.

Address arrays using array notation instead of pointer arithmetic. For example: Arr[i] instead of *(Arr + i)

6 Control Flow

Booleans: Boolean variables are useful in simplifying control flow constructs and they add readability to a program, but it helps to use them correctly.

6.1—The `if` Statement

Always use proper indentation to show corresponding if and else clauses, especially when using nested if .. else logic for multi-way alternatives. Avoid the "`if` ... `if` ... `else`" trap. The code

```
if ( ... )    if ( ... ) ...; else ...;
```

will not do what the indentation level suggests, and it can take hours to find such a bug. Always use an extra pair of { ... } when dealing with "`if` ... `if` ... `else`":

```
if ( ... )
{  if ( ... ) ...;
} // {...} are necessary
else ...;

if ( ... )
{  if ( ... ) ...;
   else ...;
}
// {...} not necessary, but they keep you out of trouble
```

6.2—The `for` Statement

Use `for` loops only when a variable runs from somewhere to somewhere with some constant increment/decrement:

```
for (int i = 0; i < a.length; i++)    cout << (a[i]) <<endl;
```

Do not use the `for` loop for weird constructs such as

```
for (a = a / 2; count < ITERATIONS; cout << (xnew))
// Don't do this
```

Make such a loop into a `while` loop. That way, the sequence of instructions is much clearer.

```
a = a / 2;
while (count < ITERATIONS) // OK
{  . . .    cout << xnew; }
```

6.3—Exceptions

Do not tag a function with an overly general exception specification:

```
Widget getline(cin, mystring) throws (Exception); // Bad
```

Instead, specifically declare any checked exceptions that your function may throw:

```
Widget getline(cin, mystring) throws (IOException, MalformedWidgetException);
// Good
```

Do not "squelch" exceptions:

```
try { double price = getnumber (cin); }
catch (Exception e) {} // Bad
```

Beginners often make this mistake "to keep the compiler happy". If the current function is not appropriate for handling the exception, simply use a `throws` specification and let one of its callers handle it.

7 Lexical Issues

7.1—Naming Conventions

Names should represent the problem or program concepts that they stand for. Avoid misleading names. Names must be reasonably long and descriptive. Use `first_player` instead of `fp`. No drppng f vwls. Local variables that are fairly routine can be short (`ch`, `i`) as long as they are really just boring holders for an input character, a loop counter, and so on. Also, do not use `ctr`, `c`, `cntr`, `cnt`, `c2` for variables in your function. Surely these variables all have specific purposes and can be named to remind the reader of them (for example, `current`, `next`, `previous`, `result`, `...`).

Class names shall start with an uppercase letter. All other letters shall be lowercase except for the beginning of run together words which may use a capital letter.

Variable names will be meaningful. The name of variable shall help describe its purpose in the program and / or what it is an abstraction of from the real world. Of course you can go to far:

numberOfMembersOnTheOlympicTeam - too long
n - too short
num_team_members - just right

And of course some short variable names are used so much they have a clear meaning even though they are very short

i, j, and k – usually used as loop counters
n, len, length - length of array or String
x, y - Cartesian coordinates

Variable and function names shall be lower case. No $ characters will be included in variable and function names.

Constants shall be declared and used for all values in your program that model real world constants or for numbers that are not to change in the program. The only exceptions are when using the numbers -1, 0, 1, and 2. Constant names shall be meaningful and help describe what the constant represents. The name shall be all capital letters with underscore characters used to separate internal words. A good constant name would be DAYS_PER_YEAR. A bad constant name would be THREE_HUNDRED_SIXTY_FIVE.

The following rules specify when to use upper- and lowercase letters in identifier names.

- All variable and function names and all data fields of classes are in lowercase (maybe with an occasional underscore in the middle); for example, `first_player`.
- All constants are in uppercase (maybe with an occasional UNDER_SCORE); for example, `CLOCK_RADIUS`.
- All class names start with uppercase and are followed by lowercase letters (maybe with an occasional underscore); for example, `Bank_teller`.

7.2—Indentation and White Space

Proper indentation is used to show the logic/structure in your program. Use tab stops every four columns. That means you may need to change the tab stop setting in your editor!

Statements in the same program block shall be lined up vertically. To ensure this is correct no matter what editor is being used to view your code, you should use either all tabs or all spaces. Mixing tabs and spaces may cause the code to look good in your editor, but it may look bad in another editor if the tabs are set differently.

Code shall be indented to show the code block it belongs to, and that block's level of nesting within the program logic. For example other than the class header and class braces, all code in a class shall be indented one tab. So the header for the main function is indented one tab, as well as the braces for the main function. The program statements and code inside the main function shall be indented two tabs. Blocks of code inside of functions shall be indented 3 tabs, etc. The end of each block shall be marked so that it can be easily matched to the beginning of that block. The longer the block and the more it is nested, the more important this is. This practice will also help you visually find mismatched braces. For example:

```
int main (… )
{ . . . .
   while (i < n)
   { cout << (a[i]) << endl;
     . . .
   } // end of while loop
   return 0;
} // end of main
```

Use blank lines freely to separate parts of a function that are logically distinct.

There shall be a space after keywords. Use a blank space around every binary operator:

```
x1 = (-b - sqrt(b * b - 4 * a * c)) / (2 * a);  // Good
x1=(-b- sqrt(b*b-4*a*c))/(2*a);//Bad
```

Leave a blank space after (and not before) each comma or semicolon. Do not leave a space before or after a parenthesis or bracket in an expression. Leave spaces around the `( . . . )` part of an `if`, `while`, `for`, or `catch` statement.

```
if (x == 0) y = 0;  f(a, b[i]);
```

Every line must fit on 80 columns. If you must break a statement, add an indentation level for the continuation:

```
a[n] = ..................................................    +
       .................;
```

Start the indented line with an operator (if possible).

If the condition in an `if` or `while` statement must be broken, be sure to brace the body in, *even if it consists of only one statement:*

```
if ( .......................................................
     && .................    || .......... ) {      . . . }
```

If it weren't for the braces, it would be hard to separate the continuation of the condition visually from the statement to be executed.

7.3— Use of Braces to Identify Blocks

Braces must be lined up vertically or horizontally. This includes braces for a class, a function, and braces that contain a code block for if/else statements. (The only time braces may be lined horizontally are for functions that contain a single line of code).

Opening and closing braces must line up, either horizontally or vertically:

```
while (i < n) { cout << (a[i]) << endl; i++; }
while (i < n)
{ cout << (a[i]) << endl;
  i++;
}
```

Some programmers don't line up vertical braces but place the { behind the key word:

```
while (i < n) { // DON'T DO THIS
   // body of loop
   i++;
}
```

Doing so makes it hard to check that the braces match.

7.4—Unstable Layout

Some programmers take great pride in lining up certain columns in their code:

```
First record = other.first record;
Last_record  = other.last_record;
cutoff       = other.cutoff;
```

This is undeniably neat, but the layout is not *stable* under change. A new variable name that is longer than the preallotted number of columns requires that you move *all* entries around:

```
// e.g. when you add something like
marginal_fudge_factor = other.marginal fudge factor;
```

This is just the kind of trap that makes you decide to use a short variable name like `mff` instead.

7.5— Use of Comments

The comments should not contradict what the code actually does. Don’t comment bad code, redesign it.

Do not use `//` comments for comments that extend for more than two lines. You don't want to have to move the `//` around when you edit the comment.

```
 // comment — don't do this
 // more comment
 // more comment
```

Use `/*` ... `*/` comments instead. When using `/*` ... `*/` comments, don't "beautify" them with additional asterisks:

```
    /* comment—don't do this  * more comment  * more comment  */
```

It looks neat, but it is a major disincentive to update the comment. Some people have text editors that lay out comments. But even if you do, you don't know whether the next person who maintains your code has such an editor.

Instead, format long comments like this:

```
/*    comment
more comment
more comment
*/
```

or this:

```
 /* comment
more comment
more comment */
```

These comments are easier to maintain as your program changes. If you have to choose between pretty but unmaintained comments and ugly comments that are up to date, truth wins over beauty.

REFERENCES:

For further reading on programming style issues see:

(1) Brian W. Kernighan and P. J. Plauger, "The Elements of Programming Style", McGraw-Hill, 1978.

(2) Brian W. Kernighan and Rob Pike, "The Practice of Programming", Chapter 1 (Style), Addison-Wesley, 1999.