A BRIEF GUIDE TO C AND C++ FOR FORTRAN PROGRAMMERS

Clopper Almon

Feb 1999 Version

Basic Syntax

1. C and C++ are free-form; anywhere that a blank may appear, any number of blanks or new line characters may appear. This fact is used to give programs indentation which makes them easy to read. The indentation, however, has no effect on the operation of the program. They are case-sensitive: x is not the same as X.

2. Every expression statement ends with a ";" . Examples:

   ```
   x = 1;
   printf("Hello, World!");
   ```

   Usually expression statements are assignments (like the first example), or function calls (like the second).

3. A single character is represented thus: 'a'. This is one byte.

4. A zero-terminated string is enclosed in " ". For example: "Smith" or "a" . This latter is two bytes: a0 . C provides a number of functions for working with such strings. Four commonly used ones ares:

   strcpy(a,b) string copy: copies string b to string a.
   strlen(a) string length: returns the length of the string, the number of bytes in the string, not counting the 0 at the end.
   strcmp(a,b) string compare: returns 0 if a and b are the same string.
   strcat(a,b) string concatenation: tacks string b onto the end of string a, making one string.

5. Anywhere that one statement is called for, a group of statements enclosed in { } can be used. A ";" is not required after the "}" except after the struct or class keywords.

6. The following logical operators are recognized:

   && and || or
   < less than > greater than <= less than or equal >= greater than or equal = = equal ! not != not equal

7. Subscripts are shown by [ ].

8. "If" statements have the form:

   if(logical expression) statement

   For example:

   ```
   if(a == b){
       x = z;
   ```

```
        x = 0;
        }
```

To illustrate point 1, this code could also be written:

```
if(a==b){x = z; x=0;}
```

"If ... else" constructions have the form

if(logical expression) statement **else** statement

For example:

```
if(a==b) x = a;
else x = 0;
```

This structure can be extended with any number of "else if" constructions. For example:

```
if(a==b) x = a;
else if (a < b)  x = a*b;
else x = a+b;
```

The final "else" may or may not be present.

9. i++ is the same as i = i+1; i-- is the same as i = i - 1;
x += y is the same as x = x + y
x -= y is the same as x = x - y
x *= y is the same as x = x*y
x /= y is the same as x = x/y.

10. Looping may be done by **while**, **for**, or **do...until** statements. The format of the **for** statement is:

for(initial; while; increment) statement

For example:

```
for(i=1; i <=n; i++){
        x[i] = y[i];
        y[i] = 0;
        }
```

For **while**, the format is

while(logical expression) statement

For example, the same thing that was done with the **for** statement above could be done with a **while** loop as follows:

```
i = 0;
while( i <= n){
        x[i] = y[i];
        y[i] = 0;
        i++;
        }
```

The last two statements could be compressed to

```
y[i++] = 0;
```

since the incrementing of i is done after the use of i as a subscript. This sort or coding, however,

is not recommended because it tends to be hard to read.

11. A comment may be shown by enclosing the comment in /* ... */. For example:
    /* This kind of comment can extend over many lines or be at the beginning of a line */
    A comment extending to the end of a line may be shown by just a // followed by the comment. For example:
    i++; // This is the same as i = i+1.

12. All data must be declared before using; for example:

```
short i, rows[10];    // i is a single, two-byte integer,
                      // rows[0], ..., rows[9] are ten two-byte integers.
long j,cols[10];      // j is a single, four-byte integer,
                      // cols[0], ..., cols[9] are ten four-byte integers.
float x, x[20];  // x is a single, four-byte floating point number.
                      // x[0], ..., x[19] are 20, four-byte floating point numbers.
double a[20][20];   // a[0][0] ... a[19][19] is an array of 8-byte floating point numbers.
char c, name[40];   // c is a single, one-byte character, name is an array of 40 characters
FILE *fp;             // fp is a "file pointer", which is the way C refers to files.
```

13. Labels, comparable to statement numbers in Fortran, and the **goto** keyword are used in this way:

```
top: x = y;
....
goto top;
```

14. The statements **break** and **continue** can only occur inside loops. The **break** statement breaks out of the present loop. The **continue** statement jumps to the next iteration of the loop. Note that C's **continue** is radically different from the Fortran CONTINUE.

15. When calling a function, C passes a copy of the value of an argument whereas Fortran passes the pointer to the argument. In a Fortran program, it is therefore very dangerous to modify the value of, say, an integer that was passed to it. In C, that modification causes no problem. It is passing arguments by value rather than by reference that makes C recursive, that is, that give it the possibility for a function to call itself. (G uses this feature to simplify greatly the programing of the evaluation of the right side of f commands.) If it is desired to pass the pointer to something, then the argument should be the pointer.

Here is a simple program for practice in reading. You will find it in the rolodex.cpp file. Although it uses many of the basic C control statements, it is fairly close to Fortran or Basic in its logic. In the following sections, we will expand it and make it more C-like and finally C++ like.

```
/* The Rolodex Program -- First Version*********************************\
This program works like a Rolodex file.  It asks the user "Whom do you want?" and looks up
the answer in a file of names and addresses.  If it finds the name, it displays the line
in the file which begins with that name. It begins by asking the user to supply the name
of the data file.  Each line in the data file should begin with a name followed by a comma.
\*********************************************************************/

#include <stdio.h>     /*  The #include directs the compiler to include the "header" file
                       stdio.h which defines for the compiler the "standard input-output"
                       functions, including printf, which is used below.  The <> tells the
                       compiler to look in its own "include" directory to find this file.*/

#include <string.h>    /* Similarly, for string.h, which is needed for all the string
                       functions. If the compiler complains that it has no prototype for
                       the function you have used, look up the function in the Library
                       Reference manual.  It will show what what header file you need.*/
void main()       // The name of the main program is always main.
{
     FILE *fp;     //fp will be a "file pointer", C's identification for a file
     char filename[40],name[40],person[120],him[40],found;
     int go, i;

     query: printf("Filename:"); //Display the message on the screen.
     gets(filename);   //Get a zero-terminated string from the user.
     if((fp = fopen(filename,"rt")) == 0){
          /*Attempt  to  open  a  text  file  by  that  name  for  reading.  The  "rt"  is
          "reading-text". The alternative to t is b for binary.  The alternative to r is w
          or w+ for writing or writing and reading. If the attempt is successful, the value
          of the file pointer will not be 0.  Therefore the following statement is reached
          only if the attempt failed.*/

          printf("Cannot open %s.\n",filename);//Error message
          goto query;
          }
     go = 1;
     while(go == 1){ //Start an infinite loop.  Only a "break" will get us out.
          printf("Whom do you want?");
          gets(name);
          if(strcmp(name,"q") == 0) break;
          fseek(fp,0,0);  //Position the fp file to the beginning.
          found = 'n';
          while(fgets(person,120,fp) != NULL){
               /*The fgets() gets a line from the fp file.  It becomes a zero-terminated
               string in "person".  A maximum of 119 bytes will be read, but if there are
               fewer bytes in the line, only as many will be read as are found.  Now put the
               name, (the first 40 characters or until a comma is encountered) into "him".
               */
               for(i=0;i<40;i++){
                    if(person[i] != ',') him[i] = person[i];
                    else break;
                    }
               him[i] = '\0'; //Make "him" a properly terminated string.
               if(strcmp(name,him)==0){
                    printf("%s\n",person);
                    found = 'y';
                    break;
                    }
               }
          if(found == 'y') continue;
          printf("%s is not in the Rolodex.\n",name);
          }
     }
```

Exercises

1. Compile, link, and run rolodex. If you machine has be set up properly, these commands will do

the compiling and linking.

```
cp rolodex
m  rolodex
```

Steal a look at rolodex.dat so you will know whom you can find. Then start the program with

```
rolodex
```

When asked "Filename:", reply "rolodex.dat".

2. If there are two people by the same last name, this program will find only the first. Make the program ask "Is this the right one?" and if the answer is no, make it keep looking. Use the getch() function. Look it up in the Library Reference.

3. Make the program accept either a comma or a space as terminating the name in the input file.

Pointers and Dynamic Space Allocation

Our rolodex.cpp has to read the whole file every time it wants to find someone. A smarter program would read it all once, extract the names, put them in an index kept in RAM, and record with each the location in the file of the first byte of that line. Then when the user asks for someone, the program just looks through the index and, if it finds the name in question, positions the file to the beginning of that line, reads it and displays it.

Now some names are long like Krollpfeiffer while others are short like Ma. In our index we would like to use just as many characters as necessary to store the name, no more, no less. To do so, we must allocate space for storing the name after we know how long it is. That is, we must allocate the space dynamically, after the program is running. This we will do with the "new" command of C++. (In C, the same thing was done a little less elegantly by the malloc function, which still works in C++.)

The other new idea in this section is that of pointers. There are two aspects of any number or letter used in a program: (1) where it is stored, its address and (b) the value that is stored there, its content. In the code

```
int x;
x = 2;
```

the second line makes the content of x equal to 2. Often it is enough to deal only with contents, but sometimes, as we shall see, it is convenient to know the address of x. In C, the address of any item is given by putting a & in front of it. Thus &x is the address of x; in C jargon, &x is the pointer to x. Conversely, if name is a pointer, then *name is the content of what it points to. In particular, *&x is the content of x. Now when an array is declared,
for example, by

```
char name[40],
```

name itself is a pointer to the first byte in this array. Thus, name is exactly the same as &name[0]. We can say that name is a pointer to a character. An almost equivalent alternative to the above

declaration would be

```
char *name;
name = new char[40];
```

The first line declares name to be a pointer to a character. The notation is intended to be mnemonic; put a * in front of name and you have a character. The second line grabs 40 bytes from "heap" memory, assigns them to this program, and sets name to point to them.[1]

Now what we need is not just space for one name but space for a large number. We need an array of names. Let us use maxrolo as a maximum number of names in our rolodex and set it equal to 1000. Then we need something like this near the beginning of our program:

```
const int maxrolo = 1000;
char **names;
names = new char*[maxrolo];
```

In the first line, the "const" protects the value of maxrolo from inadvertent change in the program. The second line says that names is to be a pointer to pointers to characters. The third line grabs enough space for 1000 pointers to characters and sets names to point to the beginning of this space. Then as we read through our rolodex data file the first time and have the name from the kth line in the array "him", we just do

```
len = strlen(him);
names[k] = new char[len+1];
strcpy(names[k],him);
```

The middle line grabs space for len+1 characters and sets names[k] to point to the beginning of it. We need len+1, not len, to allow space for the zero at the end of the string.

With the names taken care of, we can quickly attend to the remaining matter, the position in the file of the beginning of the corresponding line. The ftell function tells us where a file is at any time. We store these positions in an array of unsigned long (four-byte) integers. We will also put this array on the heap, so the program is something like this

```
unsigned long *positions;
...
positions = new unsigned long[maxrolo];
...
```

Just before the kth read of the fp file, we do

---

[1] In DOS, the "heap" is what remains of the 640K after your program has been loaded. The only difference between the effect of these two lines and the "char name[40];" declaration is where the memory is allocated. If the declaration occurs within a program, the memory is allocated on the "stack", which is where there are stored values of local variables and information about where functions should return control when they finish. The stack is much more limited than the heap, so if you get the message "stack overflow!" when you try to execute your program, you should go over to the second method. In the Borland compilers, you can also put "extern unsigned _stklen = 65536U;" above the main() statement. This line sets the stack length to its maximum size.

positions[k] = ftell(fp);

When we are looking for a name and have found that it is the kth one, we need to put the file back where it was just before reading that kth line. The statement is just

fseek(fp,positions[k],0);

The final 0 argument means to position from the beginning of the file.

We now have all the elements we need. Here is the new version of the program modified to use pointers and dynamic allocation.

```
/* The Rolodex Program with an index*/
#include <stdio.h>  // for printf()
#include <string.h> // for strcmp()
#include <conio.h>  // for getch()
void main()
{
     FILE *fp;
     char filename[40],name[40],person[120],him[40],found;
     int go, i, n, k, len;
     char **names,c;
     unsigned long *psn;
     const int maxrolo = 1000;

     psn = new unsigned long[maxrolo];
     names = new char*[maxrolo];

     query: printf("Filename:");
     gets(filename);
     if((fp = fopen(filename,"rt")) == 0){
          printf("Cannot open %s.\n",filename);
          goto query;
          }
     // Make the index
     k = 0;
     while(k < maxrolo){
          psn[k] = ftell(fp);
          if(fgets(person,120,fp) == NULL) break;
          for(i=0;i<40;i++){
               if(person[i] != ',' && person[i] != ' ') him[i] = person[i];
               else break;
               }
          him[i] = '\0';
          len = strlen(him);
          names[k]= new char[len+1];
          strcpy(names[k],him);
          k++;
          }
     n = k;
     go = 1;
     while(go == 1){
          printf("\nWhom do you want?");
          gets(name);
          if(strcmp(name,"q") == 0) break;
          search:
          for(k= 0; k < n; k++){
               if(strcmp(name,names[k]) == 0){
                    fseek(fp,psn[k],0);
                    fgets(person,120,fp);
                    printf("%s\n",person);
                    printf("Is this the right one? (y or n):");
                    c = getch();
                    printf("%c\n",c);
                    if(c == 'n') continue;
                    break;
```

```
                }
            }

        if(k == n) {
            printf("%s is not in the Rolodex.\n",name);
            continue;
            }
        }
    }
```

Structures

The arrays which we have used so far, and which are characteristic of Fortran programs, are collections of similar items. We have seen:

```
char filename[40];          // an array of characters
char **names;               // an array of pointers to characters
unsigned long *positions;   // an array of unsigned long integers.
```

We have not used but you can readily imagine how to use

```
float x[40];                // an array of 4-byte floating point numbers
int years[30];              // an array of 2-byte integers
double **a;                 // a matrix of 8-byte floating point numbers.
```

It frequently happens, however, that we would like to have unlike elements grouped together. Thus, we may have a data bank with various sorts of information on individuals. On each individual, we might have

name	a character string
date of birth	an array of three integers
income	a floating point number

and so on. C provides a device for grouping together such diverse pieces of information. It is known as a structure. Structures are useful for two reasons. First, one can pass all information about an individual to a function (or subroutine in Fortran terms) by just passing a pointer to the structure. In Fortran, if you want a function to work on a matrix, you have to pass to the function the pointer to the rectangular array and the pointer to the number of rows and the pointer to the number of columns. (Yes, you used pointers in Fortran and never knew it.) In C, you just put these three elements into a structure and pass the pointer to the structure. Thus the calls to functions are much simplified. Second, the main thing that C++ did was to generalize the C structure just slightly. The whole power of "object oriented programming" is achieved through these structures.

We will now re-write rolodey.cpp into rolodez.cpp to use a structure. We need a very simple one. For each line in the data file, we just put together the items we need for the indexing. We will call our structure a "line". Here is the code that defines its contents.

```
struct line{
    char *name;
    unsigned long position;
    };
```

Note the ; after the }. This is the one place that it is necessary. Once "line" has been defined in this way, it is just as much a data type as int, float, char, and so on. We put the definition of the structure

above the main() line so the compiler knows what a "line" is when it encounters the following declaration.

```
line *Lines;
Lines = new line[maxrolo];
```

Here Lines is an array of pointers to line structures. The "new" statement grabs enough space for maxrolo (1000) of these pointers and sets Lines to point to the beginning of that space. Note that at this point, no space has been grabbed for the content of the name in each line, only for the pointer to that content. Now as we read the rolodex data file, we fill in the position and name for each line. Note how the elements of the structure are referred to by the pointer to the structure followed by a "." followed by the name of the element. The part of the program which reads the file becomes the following.

```
k = 0;
while(k < maxrolo){
    Lines[k].position = ftell(fp);
    if(fgets(person,120,fp) == NULL) break;
    for(i=0;i<40;i++){
        if(person[i] != ',' && person[i] != ' ') him[i] = person[i];
        else break;
        }
    him[i] = '\0';
    len = strlen(him);
    Lines[k].name = new char[len+1];
    strcpy(Lines[k].name,him);
    k++;
    }
```

The part of the code that looks up the name the user has given now has these two lines:

```
if(strcmp(name,Lines[k].name) == 0){
    fseek(fp,Lines[k].position,0);
```

In this simple example, I would not claim that the use of the structure offers much advantage. The point has been, rather, to show the mechanics of using a structure in an example that is simple enough to do without it. The full code for this third stage of the rolodex program, rolodez.cpp, follows.

```
/* The Rolodex Program with a structure*/
#include <stdio.h>  // for printf()
#include <string.h> // for strcmp()
#include <conio.h>  // for getch()

struct line {
     char *name;
     unsigned long  position;
     };  // note the ; here.

void main()
{
     FILE *fp;
     char filename[40],name[40],person[120],him[40];
     int go, i, n, k, len;
     char c;
     const int maxrolo = 1000;
     line *Lines;

     Lines = new line[maxrolo];

     query: printf("Filename:");
     gets(filename);
     if((fp = fopen(filename,"rt")) == 0){
          printf("Cannot open %s.\n",filename);
          goto query;
          }

     // Make index
     k = 0;
     while(k < MAXROLO){
          Lines[k].position = ftell(fp);
          if(fgets(person,120,fp) == NULL) break;
          for(i=0;i<40;i++){
               if(person[i] != ',' && person[i] != ' ') him[i] = person[i];
               else break;
               }
          him[i] = '\0';
          len = strlen(him);
          Lines[k].name = new char[len+1];
          strcpy(Lines[k].name,him);
          k++;
          }
     n = k;
     go = 1;
     while(go == 1){
          printf("\nWhom do you want?");
          gets(name);
          if(strcmp(name,"q") == 0) break;
          search:
          for(k= 0; k < n; k++){
               if(strcmp(name,Lines[k].name) == 0){
                    fseek(fp,Lines[k].position,0);
                    fgets(person,120,fp);
                    printf("%s\n",person);
                    printf("Is this the right one? (y or n):");
                    c = getch();
                    printf("%c\n",c);
                    if(c == 'n') continue;
                    break;
                    }
               }
          if(k == n) {
               printf("%s is not in the Rolodex.\n",name);
               continue;
               }
```

```
        }
}
```

Classes

The most significant advances of C++ over C lie in the expansion of the structures concept. They are

- to allow functions to be part of the structure.
- to allow operators such as +, -, *, and = to be overloaded so that they apply to structures for which the programmer has defined them.
- to make it possible to derive one structure from another so that the derived structure has all the elements of the original plus some of its own.
- to make it possible to restrict access to some elements of a structure.

The first of these will be illustrated with the rolodex example. The others are abundantly illustrated in the Beginners' Understandable Matrix Package, BUMP, whose study should follow that of this brief introduction.

Structures with these characteristics are often called classes, and C++ has introduced the rather redundant keyword **class** for such a structure. The difference between a structure and a class lies in the last of the four points and indeed only in the *default* accessibility of its elements. In a structure, all elements are by default accessible from any part of the program, although the programmer can explicitly restrict access; in a class, they are all restricted to "private" by default but the programmer can explicitly make them accessible. The purpose of the final point, in case it seems rather strange, is to facilitate the division of labor where a number of programmers are working on one project. The programmer working on a particular structure commits to giving it certain functions that programmers working on other parts can depend upon. The internal working of the structure, however, she is free to change anyway she likes without inconveniencing her colleagues. It is obviously rather difficult to demonstrate the usefulness of the feature in small programs.

Instances of a class are called *objects.* The rolodex program with an object, rolodaze.cpp, augments the structure definition at the top as follows:

```
struct line {
    char *name;
    unsigned long  position;
    void load(char *who, unsigned long psn);
    char check(char *who);
    };  // note the ; here.
```

Here, load is a function which will allocate space for the name, copy the name to that space, and store the position number. Here is its code.

```
void line :: load(char *who, unsigned long psn){
    int len;

    position = psn;
    len = strlen(who);
    name = new char[len+1];
    strcpy(name,who);
    }
```

Note the way it is identified in the first line as a member function of the line structure. Also note that, as a member of the line structure, it is on a first-name basis with the other elements of the structure. It can refer to "position" and "name" without having to precede these names with a reference to the structure.

The check function is used to compare the name requested by the user with the name in this line and, if a match is found, to read the line from the rolodex data file and display it. It returns 'y' if it finds a match and 'n' otherwise. Here is the code.

```
char line :: check(char *who){
    char person[120];
    if(strcmp(who,name) == 0){
        fseek(fp,position,0);
        fgets(person,120,fp);
        printf("%s\n",person);
        return('y');
        }
return('n');
}
```

With this work pushed into the structure, the main program is briefer although the total length is greater. Here is rolodaze.cpp.

```
/* The Rolodex Program with an Object */
#include <stdio.h>  // for printf()
#include <string.h> // for strcmp()
#include <conio.h>  // for getch()

struct line {
     char *name;
     unsigned long  position;
     void load(char *who, unsigned long psn);
     char check(char *who);
     };  // note the ; here.
FILE *fp;  // This has been moved outside any program to make fp global, accessible from
anywhere.
void main()
{
     char filename[40],name[40],person[120],him[40],found;
     int go, i, n, k;
     char c;
     line *Lines;
     unsigned long psn;
     const int maxrolo = 1000;

     Lines = new line[maxrolo];

     query: printf("Filename:");
     gets(filename);
     if((fp = fopen(filename,"rt")) == 0){
          printf("Cannot open %s.\n",filename);
          goto query;
          }

     // Make the index
     k = 0;
     while(k < maxrolo){
          psn = ftell(fp);
          if(fgets(person,120,fp) == NULL) break;
          for(i=0;i<40;i++){
               if(person[i] != ',' && person[i] != ' ') him[i] = person[i];
               else break;
               }
          him[i] = '\0';
          Lines[k].load(him,psn);
          k++;
          }
     n = k;
     // Loop, asking whom the user wants to see.
     go = 1;
     while(go == 1){
          printf("\nWhom do you want?");
          gets(name);
          if(strcmp(name,"q") == 0) break;
          for(k= 0; k < n; k++){
               if(Lines[k].check(name) == 'y'){
                    printf("Is this the right one? (y or n):");
                    c = getch();
                    printf("%c\n",c);
                    if(c == 'n') continue;
                    break;
                    }
               }
          if(k == n) {
               printf("%s is not in the Rolodex.\n",name);
               continue;
               }
          }
     }
```

```
void line :: load(char *who, unsigned long psn){
     int len;

     position = psn;
     len = strlen(who);
     name = new char[len+1];
     strcpy(name,who);
     }

char line :: check(char *who){
     char person[120];
     if(strcmp(who,name) == 0){
          fseek(fp,position,0);
          fgets(person,120,fp);
          printf("%s\n",person);
          return('y');
          }
     return('n');
     }
```

Constructors and Destructors

The objects in rolodaze.cpp were still simple enough that we did not have to construct or destroy them. Let us now make a larger object which we shall call *rolo*, an instance of a class called *Rolodex*. With this object, we can reduce the main program to

```
char filename[40],name[40];
query: printf("Filename:");
gets(filename);
Rolodex rolo(filename);

// Loop, asking whom the user wants to see.
while(1){
     printf("\nWhom do you want?");
     gets(name);
     if(strcmp(name,"q") == 0) break;
     rolo.find(name);
     }
}
```

Note that in the first line in bold print we declare that rolo is an object of type Rolodex. In the second, we call the "find" function of this object. We want a Rolodex object to construct itself when declared, that is, we want it to open the file whose name was passed to it, read this file, and construct a rolodex index such as we have been using. Obviously, so complicated a constructor has to be specially written. Likewise, when we are through with an object, it is important to be able to destroy it, that is, to free up the RAM it is occupying so that it can be used again, if need be, later in the program. The definition of the Rolodex structure must show that it has these various functions. Here is that definition of both the line and the Rolodex structures.

```
struct line {
```

```
  private:
      char *name;
      unsigned long  position;
  public:
      line(){name=0;position= 0;}
      void load(char *who, unsigned long psn);
      char check(char *who, FILE *fp);
      ~line();
      };

struct Rolodex{
   private:
      FILE *fp;
      line *Lines;
      int maxrolo,nlines;
   public:
      Rolodex(char *filename);
      ~Rolodex();
      void find(char *name);
      };
```

In the Rolodex function, the three functions we have mentioned appear below the line "public:". That line makes those functions accessible from anywhere in the program, whereas the elements defined to be "private" are accessible only to these three member functions and other functions explicitly declared to be "friends" of the structure. This structure has no friends; in BUMP we will see structures with lots of friends. A constructor always has the same name as the structure, so Rolodex(char *filename) is the constructor. A structure may have several constructors if they are distinguishable by the number and type of arguments which they have. The destructor always has as a name the name of the structure preceded by a ~. Thus, ~Rolodex() is the destructor. There is never more than one destructor and it has no arguments. Constructors and destructors cannot have return values, so their names are not preceded by a return type in the structure definition. Because one and the same main program might now conceivably have several Rolodex objects, we have put fp, maxrolo, and nlines (the number of lines in the data file) into the Rolodex structure. Since the constructor cannot return a value to let the calling program know if it had trouble, a global variable, RoloOpen, has been introduced to allow it to communicate with the program which calls it. Here is the new part of the Rolodex constructor. Putting in the now-familiar code for making the index has been left as an exercise.

```
Rolodex::Rolodex(char *filename)
{
	int k,i;
	char person[120], him[40];
	unsigned long psn;

	maxrolo = 1000;
	Lines = new line[maxrolo];
	if((fp = fopen(filename,"rt")) == 0){
		printf("Cannot open %s.\n",filename);
		RoloOpen = 'n'; /*  This round-about communication is necessary
					because a constructor cannot return a value. */
		return;
		}

	RoloOpen = 'y';
	// Make the index
	k = 0;
	while(k < maxrolo){
		/* exercise */
		}
	nlines = k;
	}
```

Now we have to deal with the destructor. Here it is. The necessary points are noted in the comments.

```
Rolodex::~Rolodex(){
	delete [] Lines; // the [] is required to show that Lines is an array.
	fclose(fp);	/*	Since fp was opened in the constructor, it must be
				closed in the destructor. */
	}
```

When the program hits the line "delete [] Lines" it will recall that is has maxrolo "line" objects in Lines, and will call the destructor for line that many times in inverse order. That is, it will delete line maxrolo-1 first and line 0 last. The destructor for a line object is

```
line::~line(){
	if(name != 0)
		delete [] name;
	}
```

Note the test on the value of name . Unfortunately, it is very destructive to "delete" something which has not been assigned with a "new" or something which has already been deleted. Look back now

at the definition of the line structure. Its constructor was so short and simple that it could be written in the definition without cluttering it up. It was just

```
line(){name=0;position= 0;}
```

Note, however, that it assures us that name is initially equal to 0 so that the test in the destructor will work correctly.

It is clear when a constructor is called. When is the destructor called? In C++ jargon, the answer is When the object goes out of scope. But what does that mean? In the simplest case, which is all we shall deal with, it means that if the object was declared as a local variable in a function, its destructor is called when that function is completed and returns. To check that our destructor is working correctly, we will put the declaration of the Rolodex in a function called sub(). We will check the free core left before we call sub and again after it returns. If we get the same answers both times, we know that our destructors are working correctly.

```
#include <alloc.h>  // for coreleft()

// Function prototypes
void sub(void);

// Global variables
char RoloOpen;

void main()
{
      unsigned long core;

      core = coreleft();
      sub();
      printf("Original core left: %ld\n",core);
      core = coreleft();
      printf("Final core left:    %ld\n", core);
      }

void sub(void){
      char filename[40],name[40];
      query: printf("Filename:");
      gets(filename);
      Rolodex rolo(filename); // This is a declaration, just as is the "char" line.
      if(RoloOpen == 'n') goto query;
      printf("Intermediate core left: %ld\n", coreleft());
```

```
    // Loop, asking whom the user wants to see.
    while(1){
        printf("\nWhom do you want?");
        gets(name);
        if(strcmp(name,"q") == 0) break;
        rolo.find(name);
        }
    }
```

C++ insists upon knowing the format, or prototype, of each function before that function is used. Most of these prototypes have been provided in the structure specification. The function sub(), however, is not part of any of these so it must have its own prototype statement. It differs from the first line of the declaration of the function by ending in a ";". Here it is.

```
    // Function prototypes
    void sub(void);
```

With these components, you should be able to put together the final version of the rolodex program, complete with constructors and destructors. It is called roldover.cpp on the disk, but it would be a good exercise for you to write it from what has been given here.
Once it works properly, try removing the fclose from the Rolodex destructor. Do you get back all your core?

Exercise

Create a Vector structure which has the following definition.

```
struct Vector{
  private:
    int n;              // number of elements
    float *v;   // the elements
 public:
    Vector (char *filename);   // Read the vector from the named file text file.
    ~Vector();              // Destructor
    int show(char *title, int FieldWidth = 8, int DecimalPlaces = 2);
        // Display the vector on the screen
    float sum();            // return the sum
    float enorm();           // Return the Euclidian norm (Square root of sum of squares)
    float lnorm();           // Return the l-norm. (sum of absolute values)
    float mnorm();           // Return the m-norm. (max absolute value)
    };
```

You should write all the functions and verify that they work. You may choose any format you like for the text file from which you read the vectors. Perhaps the easiest format for which to program is to put the number of elements on the first line and the elements on the following lines, one per line. You may use the function atof() to convert text strings to floats. Read about atof in the compiler help files.

Overloading Operators and Friends

The Vector structure that you wrote in the last exercise was fine as far as it went, but perhaps it occurred to you that the Vectors were lonely. There was no way for them to interact with one another. It should be possible to add Vectors together or to find the angle between two of them. In this section, we show how to overload the + and = operators so that one can write code such as

```
Vector a("a.vec"),b("b.vec"),c(4);
c = a + b;
c.show("C = A + B");
```

To do so, we will need to expand the definition of the Vector class to the following.

```
struct Vector{
  private:
      int n;              // number of elements
      float *v;   // the elements
      char temp;  // y if the vector has been created by an operator
      void freet(){if(temp == 'y') freeh();}
 public:
      Vector(char *filename);   // Read the vector from the named file text file.
      Vector(int n, char temporary = 'n');
      Vector(Vector& a);        // Copy constructor
      ~Vector();                // Destructor
      void Vector::show(char *title,int FieldWidth=8, int DecimalPlaces=2);
      float sum();              // return the sum
      float enorm();             // Return the Euclidian norm
      float lnorm();             // Return the l-norm. (sum of absolute values)
      float mnorm();              // Return the m-norm. (max absolute value)
      void freeh();             // Free the heap memory.
      Vector& operator = (const Vector& a);
      float&  operator [](const int i);
      friend Vector operator + (const Vector &a, const Vector &b);
      };
```

The new elements have been shown in bold type. The simplest is the "temp" character. Why is it necessary?

Consider the problem of adding three vectors:

d = a + b + c.

The computer will first have to add a and b to get an intermediate result. Then to this intermediate result, it must add c. Then it should throw away the intermediate result. If it does not throw it away, the computer's RAM will soon become clogged with these intermediate results and the program will grind to a halt. Thus, operators like + will need to create temporary vectors to hold the intermediate results. Timely disposal of these intermediate results is the trickiest part of achieving the goals of this section. This "temp" flag will be used by the operators to indicate that the vector is such an intermediate product and can -- and must -- be thrown away when it is no longer needed.

Now let us turn to the first of two new constructors. It just constructs a vector of n elements and sets the temp element to the letter passed by the call. Writing it can be left as a exercise. If one calls the constructor by, say, just "Vector c(4);", the default value of the temp flag, n, will be used. On the other hand, this same constructor handles a declaration like "Vector c(4,'y');" within an operator to create an intermediate Vector with the temp set to 'y'. Writing this constructor is simple enough to be left as an exercise.

The next item to considered is the "friend" function that overloads the operator +. A function declared within the definition of the structure to be a friend can access the private elements of the structure. Here is the code for this function.

```
Vector operator + (const Vector& a, const Vector& b){
    int i;
    if(b.n != a.n){
        printf("Vector dimensions do not match in + operator.\n");
        exit(1);
        }
    Vector Temp(a.n,'y');
    for(i = 0; i < a.n; i++)
        Temp.v[i] = a.v[i] + b.v[i];
    a.freet();
    b.freet();
    return (Temp);
    }
```

After checking that the dimensions match, a vector called Temp is constructed with the right number of elements and marked as temporary. Now the way that the + operator works makes the vector on the left of the + the first argument and the vector on the right the second argument. The &'s in the declaration of the function means that these vectors will be passed to the function by reference, not by copying. The **for** loop adds the two vectors together and puts the sum in Temp. The keyword **const** in the declaration of the function is a compiler assistant. It tells the compiler that the following argument is not changed by the function and allows faster compilation. It should be unnecessary. Unfortunately, the Borland Builder C++ compiler has made **const** mandatory in some contexts, even where the argument is *not* constant. However, changing a variable declared as **const** produces only

a compiler warning and the code works correctly, while omitting it in these contexts produces a compiler error, and nothing works. All of the code in these notes previously worked without **const**.

We have just seen a temporary vector created by an operator. A little thought about how you would use temporary scratch paper if you were adding vectors by hand should soon convince you that no temporary is ever used more than once. Hence, we check to see if a was a temporary, and if so we delete it, and likewise for b. These checks are the "freet" calls. (Freet is short for Free if Temporary.) The code for freet was given "in line" in the definition of the structure. That code used the function freeh(), for free heap memory, which is defined as follows:

```
void Vector::freeh(){
    if(v != 0 ) delete v;
    v = 0;
    }
```

If the pointer to the array in the vector is not zero, this frees the heap memory assigned to it and sets the pointer equal to 0.

Now finally note that operator + returns the Temp vector it has just created. But Temp was created in this function and therefore must be destroyed when the function "goes out of scope" or "returns". Now the compiler cannot both return Temp and destroy it, so what does it do? It first makes a copy using the copy constructor, the one whose argument is just a pointer to another object of the same type. Then it destroys the original of Temp. We can take advantage of this knowledge in writing the copy constructor. If we see that the object being copied is a temporary, the copy constructor can just steal the heap memory that belonged to the object being copied, for we can be sure that the temporary is headed straight for the destructor. This theft prevents the heap from becoming fragmented. If we had allocated new memory on the heap for the copy, it would have been above the memory for the original temporary, so when this latter memory was freed, we would have had a hole in the heap. While such a hole is not necessarily the end of the world, it is certainly better to have a compact heap, for any one call of the "new" command can only allocate one continuous chunk of contiguous memory. If the heap becomes fragmented, "new" may not be able to find enough space all in one piece even though it available in scattered parcels. Here is the code for the copy constructor and for the destructor that keeps the heap compact.

```
// The Copy constructor
Vector::Vector(Vector& a)   {
    int i;

    n = a.n;
    temp = a.temp;

    if(a.temp == 'y'){ // If we are copying a temporary,
        v = a.v;     // steal the temporary's piece of the heap,
        a.v = 0;     // and tell it that it has been robbed.
        }
```

```
        else{
            if((v = new float[n]) == 0){
                printf("Out of memory trying to create vector.\n");
                exit(1);
                }
            for(i = 0; i < n; i++)
                v[i] = a.v[i];
            }
        }

    // The destructor
    Vector::~Vector(){
        if(v != 0 )  // check to see if it has been robbed.
             delete v; //if not, delete it.
        v = 0;
        }
```

The meaning of the operator & in function definitions requires some explanation. In the declaration of the copy constructor,

Vector::Vector(Vector& a),

it simply meant that it was sufficient to pass just a reference to a to the routine; it was not necessary to make a separate copy of a for the purpose. The matter is somewhat more perplexing when we come to write the overloading of the [] and = operators for Vectors. We want to define [ ] so that if b is a Vector, we can write b[i] for the ith element of b on either the right or the left side of an equal. Now note that in a state as simple as

x = x+1;

what the compiler does with the x on the right of the = is totally different from what it does on the left. On the right, it takes data from "x" while on the left, it puts data into "x". An expression which can be used in this way on either side of an = but with a very different meaning on the left side is called an lvalue, meaning something that can be used on the left of an = sign. In the example, the expression x is an lvalue. The expression x+1 is not an lvalue; it can only be used on the right side of the = sign. Clearly, we want the [] function to return an lvalue value so that we can use b[i] on either side of an = sign.

We get this lvalue by a & in the declaration of the function that overloads [] thus:

```
float& Vector::operator [] (const int i)
```

What is returned? NOT a pointer to a float, but an "lvalue" for the float. Of course, what is returned must itself be an lvalue. The return statement in this function is

return (v[i]);

and v[i] is certainly an lvalue. But it requires the & in the return type of the function to preserve the

lvalue character of what is returned. The complete code for the [] operator, which also checks that the subscript is in range, is as follows.

```
float& Vector::operator [](const int i)  {
    if(v==0){
        printf("Error: Reference to a deleted vector.\n");
        exit(1);
        }
    if ( i < 0 || i >= n){
        printf("Illegal vector index %d.\n",i);
        exit(1);
        }
    return(v[i]);
    }
```

The code for the = operator also uses the & to indicate that what is returned is an lvalue, in this case the lvalue of the Vector on the left of the = sign. The overloading of the = operator can only be a member function, not a friend. In it, the Vector a, the argument, is the vector on the right side of the = sign. The Vector on the left is the present Vector. Here is the code.

```
Vector& Vector::operator = (const Vector& a){
    int i;
    int s = (n < a.n) ? n : a.n;
    for(i = 0; i < s; i++){
        v[i] = a.v[i];
        }
    a.freet();
    return(*this);
    }
```

The code clearly copies the a Vector into the present vector and frees a if it is a temporary. The word **this** is a keyword in C++ and is a pointer to the present instance of the structure. We do not, however, want to return a pointer to the present vector but a reference to it. Thus we return ***this**, not **this**. The & in the return type of the function ensures that we get an lvalue, which will make the compiler happy. If the explanation of why the [] and = operators are written the way they are seems to you a little strained, I fully agree. The fat books on C give next to no explanation but only an example. The **const** keyword in the declaration will cause a compiler warning, but if omitted, there is an a compiler error. The code seems to work fine. Surely requiring the **const** is a bug in the compiler.

Exercise

Expand the vector structure so that x - y, x*y, a*x, and x/a are defined, where x and y are Vectors and a is a float. For x*y, use the "dot" or "inner" product definition. In your test program, try such expressions as a*(x -y) or (w-x)*(y - z). They should work correctly.