```
/*   File: blkcopy.c
     The program uses block I/O to copy a file.
*/
#include <stdio.h>
main()
{    signed char buf[100];
     const void *ptr = (void *) buf;
     FILE *input, *output;
     size_t n;

     printf("***File Copy - Block I/O***\n\n");
     printf("Input File: ");
     gets(buf);
     input = fopen(buf, "r");
     if (!input) {
          printf("Unable to open input file\n");
          exit(0);
     }
     printf("Output File: ");
     gets(buf);
     output = fopen(buf, "w");
     if (!output) {
          printf("Unable to open output file\n");
          exit(0);
     }
     while ((n = fread(ptr, sizeof(char), 100, input)) == 100)
          fwrite(ptr, sizeof(char), 100, output);
     fwrite(ptr, sizeof(char), n, output);
     close(input);
     close(output);
}
```

Figure 13.1: Copying a File Using Block I/O

```
/*   File: bincopy.c
     This program copies a binary file. Standard files are not allowed.
*/
#include <stdio.h>

main()
{    int c;
     char s[25];
     FILE *input, *output;

     printf("***Binary File Copy - Character I/O***\n\n");
     printf("Input File: ");
     gets(s);
     input = fopen(s, "rb");

     if (!input) {
          printf("Unable to open input file\n");
          exit(0);
     }

     printf("Output File: ");
     gets(s);
     output = fopen(s, "wb");

     if (!output) {
          printf("Unable to open output file\n");
          exit(0);
     }

     while ((c = fgetc(input)) != EOF)
          fputc(c, output);
     close(input);
     close(output);
}
```

Figure 13.2: File Copy Program for Binary Files

13.2 Tell and Seek

The file functions described in the last section, as well as the formatted file I/O functions, read and write data sequentially from the *current position* in the file. This position in the stream is one of the pieces of information stored in the `FILE` data structure which we get a pointer to when the file is opened. The standard library provides a function to determine the current position, `ftell()`. The position returned from `ftell()` is in terms of the number of bytes offset from the beginning of the file. The library also provides a function which allows a program to move the current position elsewhere in stream: `fseek()`.

ftell *Prototype:* `long ftell(FILE *stream);` *in:* <stdio.h>

Returns: the current file pointer in `stream` measured in bytes from the beginning. If there is an error, it returns -1L.

fseek *Prototype:* `int fseek(FILE *stream, long offset, int base);` *in:* <stdio.h>

Returns: 0 if successful; nonzero on failure. (DOS returns zero even if pointer is not repositioned.)

Description: The function, `fseek`, sets the file pointer for `stream` to a new position that is `offset` bytes from the location specified by `base`. For binary files, `offset` is the number of bytes to offset. For text files, `offset` must be zero or a value returned by `ftell()`. The value of `base` must be 0 to indicate the beginning of the file, 1 for the current position, or 2 for the end of file. `stdio.h` defines three constants for the base position: `SEEK_SET` is 0, `SEEK_CUR` is 1, and `SEEK_END is 2`.

Let us illustrate the use of the above functions in a simple program. Suppose we wish to write a program that opens a file and begins to read items from the file. Suppose at some point, the program requires the size of the file. Then, after the size is determined, the program should resume reading items from where it left off.

The program will: read and print a number of strings from a file; then, call a function `filesize()` that returns the size of the file; and finally, resume reading and printing strings from the file. The program driver is shown in Figure 13.3.

Let us now write `filesize()`, which returns the size of the file, `fp`. It determines the size of the file by moving the current position to the end of the file and finding the offset of the pointer in bytes. However, before the current position is moved to the end of the file, the function must save the position where it left off. This saved position is restored by `filesize()` before it returns to the calling program. The function is shown in Figure 13.4.

The function first saves the current position in `savepos`; then moves the pointer to the end of the file. It uses `0L` for offset since the offset must be long. Next, the function uses `ftell()` to find the offset in bytes of the current position, which is now at the end of the file. The value of `end` is, therefore, the size of the file. Finally, the saved position is restored and the file size is returned. Here is a sample session:

```
***File Seek - File Size***

File Name:  payin.dat
```

```
/*   File: seek.c
     This program illustrates the use of fseek() to reposition
     a file pointer. The program reads a specified number of strings
     from a file and prints them. Then, the program calls on filesize()
     to print out the size of the file. After that the program resumes
     reading and printing strings from the file.
*/

#include <stdio.h>
#define MAX 81
long int filesize(FILE *fp);

main()
{    int m, n = 0;
     FILE *fp;
     char s[MAX];

     printf("***File Seek - File Size***\n\n");
     printf("File Name: ");
     gets(s);
     fp = fopen(s, "r");

     if (!fp)
          exit(0);
     printf("Number of lines in first printing: ");
     scanf("%d", &m);

     while (fgets(s, MAX, fp)) {   /* read strings */
          fputs(s, stdout);        /* print the strings */
          n++;
          if (n == m)    /* if m string are printed, print file size */
               printf("Size of file = %ld\n", filesize(fp));
     }

     fclose(fp);
}
```

Figure 13.3: Driver for Program Illustrating `ftell()` and `fseek()`

```
/*   File: seek.c - continued */
/*   Returns the size of the file stream fp.*/
long int filesize(FILE *fp)
{    long int savepos, end;

     savepos = ftell(fp);     /* save the file pointer position */
     fseek(fp, 0L, SEEK_END); /* move to the end of file */
     end = ftell(fp);         /* find the file pointer position */

     fseek(fp, savepos, SEEK_SET); /* return to the saved position */
     return end;                   /* return file size */
}
```

Figure 13.4: Code for `filesize()`

```
Number of lines in first printing:  3
/* File:  payin.dat */
ID Last First Middle Hours Rate
--------------------------------------------------------------------
Size of file = 238
5 Jones Mike David 40 10
7 Johnson Charles Ewing 50 12
12 Smythe Marie Jeanne 35 10
```

In the sample session, the first three lines of `payin.dat` are printed and then the size of the file is printed as 238 bytes. Finally, the rest of the file `payin.dat` is printed.

A few comments on the use of `fseek()`:

For text files, the offset value passed to `fseek()` can be either 0 or a value returned by `ftell()`.

When `fseek()` is used for binary files, the offset must be in terms of actual bytes.

13.3 A Small Data Base Example

A data base is a collection of a large set of data. We have seen several examples of data bases in previous chapters, such as our payroll data and the list of address labels discussed in Chapter 12. In our programs working with these data bases we have simply read data from files, possibly performed some calculations, and printed reports. However, to be a useful data base program, it should also perform other management and maintenance operations on the data. Such operations include editing the information stored in the data base to incorporate changes, saving the current information in the data base, loading an existing data base, searching the data base for an item, printing a report based on the data base, and so forth. Programs that manage data bases can become quite elaborate, and such a program to manage a large and complex data base is called a *Data Base Management System (DBMS).*

```
/*   File: lbldb.h
     This file contains structure tags for labels. Label has two
     members, name and address, each of which is a structure type.
*/
struct name_recd {
     char last[15];
     char first[15];
     char middle[15];
};

struct addr_recd {
     char street[25];
     char city[15];
     char state[15];
     long zip;
};

struct label {
     struct name_recd name;
     struct addr_recd address;
};

typedef struct label label;
```

Figure 13.5: Data Structure Definitions for Label Data Base Program

In this section we will implement a rather small data base system that maintains our data base for address labels. We will assume that there are separate lists of labels for different groups of people; therefore, it should be possible to save one list in a file named by the user as well as to load a list from any of these files. The data in a list of labels is mostly fixed; however, it should be possible to make additions and/or changes. It should also be possible to sort and search a list of labels.

In our skeleton data base system, we will not implement sorting and searching operations (we have already implemented a sort function for labels in Section 12.3), instead, our purpose here is to illustrate some of the other operations to see the overall structure of a DBMS. We will implement operations to add new labels, print a list of labels, as well as loading and saving lists in files. The program driver will be menu driven. The user selects one of the items in the menu, and the program carries out an appropriate task. The data structures we will use include the `label` structure and a type, `label`, defined in the file `lbldb.h` shown in Figure 13.5. The program driver is shown in Figure 13.6.

The list of labels is stored in the array, `lbllist[]`, and `n` stores the actual number of labels, initially zero. A new list is read by the function `load()` which returns the number of labels loaded. A list can be edited by `edit()` which updates the value of `n`. Both `edit()` and `load()` must not exceed the maximum size of the array. The functions `print()` and `save()` write `n` labels from the

```
/*   File: lbldb.c
     Header Files: lbldb.h
     This program initiates a data base for labels. It allows the
     user to edit labels, i.e., add new labels, save labels in a
     file, load a previously saved file, and print labels.
*/

#define MAX 100
#include <stdio.h>
#include <ctype.h>
#include "lbldb.h"       /* declarations for the structures */

main()
{    char s[25];
     label lbllist[MAX];
     int n = 0;

     printf("***Labels - Data Base***\n");
     printf("\nCommand: E)dit, L)oad, P)rint, S)ave, Q)uit\n");
     while (gets(s)) {

          switch(toupper(*s)) {
               case 'E': n = edit(lbllist, n, MAX); break;
               case 'L': n = load(lbllist, MAX); break;
               case 'P': print(lbllist, n); break;
               case 'S': save(lbllist, n); break;
               case 'Q': exit(0);
               default: printf("Invalid command - retype\n");
          }
          printf("\nCommand: E)dit, L)oad, P)rint, S)ave, Q)uit\n");
     }
}
```

Figure 13.6: Driver for Label Data Base Program

current list.

Figure 13.7 shows a partial implementation of `edit()`, allowing only the addition of new labels. It does not implement operations for deletion or change of a label. The `edit()` function presents a sub-menu and calls the appropriate function to perform the task selected by the user. We have included program "stubs" for the functions `del_label()` and `change_label()` which are not yet implemented. The `add_label()` function calls on `readlbl()` to read one label. If a label is read by `readlbl()` it returns `TRUE`; otherwise, it returns `FALSE`. The loop that reads labels terminates when either the maximum limit is reached or `readlbl()` returns `FALSE`. Each time `readlbl()` is called, `n` is updated, and the updated value of `n` is returned by `add_label()`. In turn, `edit()` returns this value of `n` to the main driver.

The function `readlbl()` first reads the last name, as shown in Figure 13.8. If the user enters an empty string, no new label is read and the function returns `FALSE`; otherwise, the remaining informations for a label is read and the function returns `TRUE`.

The `print()` function calls on `printlabel()` to print a single label data to the standard output. The functions are shown in Figure 13.9.

Finally, we are ready to write functions `load()` and `save()`. We will use `fread()` and `fwrite()` to read or write a number of structure items directly from or to a binary file. This method of storing the data base is much more efficient that reading ASCII data, field by field for each label. The code is shown in Figure 13.10. The function `load()` opens an input file, and uses `fread()` to read the maximum possible (`lim`) items of the size of a `label` from the input file. The buffer pointer passed to `fread()` is the pointer to the array of `labels`, `lbllist`. Finally, `load()` closes the input file and returns `n`, the number of items read. Similarly, `save()` opens the output file, and saves `n` items of `label` size from the buffer to the output file. It then closes the output file and returns `n`. If it is unable to open the specified output file, it returns 0. A sample session is shown below.

```
***Labels - Data Base***

Command:  E)dit, L)oad, P)rint, S)ave, Q)uit
l
Input File:  lbl.db

Command:  E)dit, L)oad, P)rint, S)ave, Q)uit
p

Label Data:

James Edward Jones
25 Dole St
Honolulu Hi 96822

Jane Mary Darcy
23 University Ave
Honolulu Hi 96826

Helen Gill Douglas
```

```
/*   File: lbldb.c - continued */
/*   Edits labels: adding labels has been implemented so far. */
int edit(label lbllist[], int n, int lim)
{    char s[80];

     printf("A)dd, D)elete, C)hange\n");
     gets(s);

     switch(toupper(*s)) {
          case 'A': n = add_label(lbllist, n, lim);
                    break;
          case 'D': del_label();
                    break;
          case 'C': change_label();
                    break;
          default: ;
     }

     return n;
}

/*   Adds new labels to lbllist[] which has n labels. The maximum
     number of labels is lim.
*/
int add_label(label lbllist[], int n, int lim)
{
     while (n < lim && readlbl(&lbllist[n++]))
          ;

     if (n == lim)
          printf("Maximum number of labels reached\n");
     else --n;      /* EOF encountered for last value of n */

     return n;
}

void del_label(void)
{
   printf("Delete Label not yet implemented\n");
}

void change_label(void)
{
   printf("Change Label not yet implemented\n");
}
```

Figure 13.7: Partial Code for Editing the Data Base

```
/*   File: lbldb.c - continued */
/*   Includes and defines included at the head of the file. */
#define FALSE 0
#define TRUE 1

/*   This routine reads the label data until the name is a blank. */
int readlbl(struct label * pptr)
{    int x;
     char s[25];

     printf("Enter Last Name, RETURN to quit: ");
     gets(s);
     if (!*s)
          return FALSE;
     else strcpy(pptr->name.last, s);
     printf("Enter First and Middle Name: ");
     x = scanf("%s %s%*c",pptr->name.first, pptr->name.middle);
     printf("Enter Street Address: ");
     gets(pptr->address.street);
     printf("Enter City State Zip: ");
     scanf("%s %s %ld%*c",pptr->address.city, pptr->address.state,
               &(pptr->address.zip));
     return TRUE;
}
```

Figure 13.8: Code for `readllbl()`

```
/*   File: lbldb.c - continued */
/*   Prints n labels stored in lbllist[]. */
void print(label lbllist[], int n)
{    int i;

     printf("\nLabel Data:\n");
     for (i = 0; i < n; i++)
          printlabel(&lbllist[i]);
}

/* This routine prints the label data. */
void printlabel(struct label * pptr)
{
     printf("\n%s %s %s\n%s\n%s %s %ld\n",
               pptr->name.first,
               pptr->name.middle,
               pptr->name.last,
               pptr->address.street,
               pptr->address.city,
               pptr->address.state,
               pptr->address.zip);
}
```

Figure 13.9: Code for `print()` and `printlabel()`

```
/*   File: lbldb.c - continued */
/*   Loads a maximum of lim labels from a file into lbllist[].
     Returns the number n of labels actually read.
*/
int load(label lbllist[], int lim)
{    char s[25];
     FILE *infp;
     int n;

     printf("Input File: ");
     gets(s);
     infp = fopen(s, "r");
     if (!infp)
          return 0;
     n = fread(lbllist, sizeof(label), lim, infp);
     fclose(infp);
     return n;
}

/*   Saves n labels from lbllist[] to a file. */
int save(label lbllist[], int n)
{    char s[25];
     FILE *outfp;

     printf("Output File: ");
     gets(s);
     outfp = fopen(s, "w");
     if (!outfp)
          return 0;
     fwrite(lbllist, sizeof(label), n, outfp);
     fclose(outfp);
     return n;
}
```

Figure 13.10: Code for `load()` and `save`

```
123 Kailani Ave
Kailua Hi 96812

Command:  E)dit, L)oad, P)rint, S)ave, Q)uit
e
A)dd, D)elete, C)hange
a

Enter Last Name, RETURN to quit:  Springer
Enter First and Middle Name:  John Karl
Enter Street Address:  Coconut Ave
Enter City State Zip:  Honolulu Hi 96826
Enter Last Name, RETURN to quit:

Command:  E)dit, L)oad, P)rint, S)ave, Q)uit
s
Output File:  lbl.db

Command:  E)dit, L)oad, P)rint, S)ave, Q)uit
q
```

The session starts with the menu menu. We select the menu item Load to load a previously saved list of labels in the file, `lbl.db`. After this file is loaded, we select Print to print the labels. Next, we select Edit and Add to add one new label. Then we select Save to save the revised list to the file `lbl.db`. Finally, we select Quit to exit the program.

13.4 Operating System Interface

As stated at the beginning of the chapter, all of our programs so far have had minimal interaction with the environment in which they are running, i.e. the operating system, and in particular the shell. One area where we could make use of operating system support is in specifying files to be used in execution of the program. In our previous examples we have either redirected the input or output when running the program (and read or written to the standard input or output in the program code), or prompted the user explicitly for the file names once the program has begun executing. However, this is not the only (nor most convenient) way to specify files to a program. It should also be possible to pass arguments to a program when it is executed. An executable program is invoked by a command to the host operating system consisting of the name of the program. However, the entire command may also include any arguments that are to be passed to the program. For example, the C compiler does not prompt us for the file names to be compiled; instead we simply type the command:

```
cc filename.c
```

The entire command is called the *command line* and may include additional information to the program such as *options* and file names. The C compiler (and most, if not all, other commands) is also simply a C program.

There must be a way this additional information can be passed to an executing program. There is. The command line arguments are passed to the formal parameters of the function `main()`. We have always defined the function `main()` with no formal parameters; however, in reality it does have such parameters. The formal parameters of `main()` are: an integer, `argc`, and an array of pointers, `argv[]`. The full prototype for `main()` is:

```
int main(int argc, char * argv[]);
```

Each word typed by the user on the command line is considered an argument (including the program name). The parameter `argc` receives the integer count of the number of arguments on the command line, and each word of the line is stored in a string pointed to by the elements of `argv[]`. So, if the command line consists of just the program name, `argc` is 1 and `argv[0]` points to a string containing the program name. If there are other arguments on the command line, `argv[1]` points to a string containing the first argument after the program name, `argv[2]` to the next following one, and so forth. In addition, `main()` has an integer return value passing information back to the environment specified by the `return` or `exit` statement terminating `main()`. Recall, we have always used `exit` in the form:

```
exit(0);
```

A common convention in Unix is that a program terminates with a zero return value if it terminates normally, and with non-zero if it terminates abnormally.

Figure 13.11 shows a program that prints the values of `argc` and each of the strings pointed to by the array `argv[]`. The program then uses the first argument passed on the command line as a "source" file name, and the second as a "destination" file name and copies the source file to the destination. The program returns zero to the environment to indicate normal termination.

Sample Session with a command line:

```
filecopy filecopy.c xyz

     ***Command Line Arguments - File Copy***

     Number of arguments, argc = 3
     The arguments are the following strings
     C:\BK\BOOK\CH9\FILECOPY.EXE
     filecopy.c
     xyz
```

The number of arguments in the command line is 3, and each of the strings pointed to by the array `argv[]` is then printed. The first argument is the complete path for the program name as interpreted by the host environment. The program then opens the files and copies the file `filecopy.c` to `xyz`.

In addition to receiving information from the operating system, a program can also call on the shell to execute commands available in the host environment. This is very simple to do with C using the library function `system()`. Its prototype is:

```
int system(const char *cmmdstr);
```

```
/*   File: filecopy.c
     This program shows the use of command line arguments. argc is the number
     of words in the command line. The first word is the program name, the next
     is the first argument, and so on. The program copies one file to another.
     The command line to copy file1 to file2 is:

          filecopy file1 file2
*/

#include <stdio.h>
main(int argc, char *argv[])
{    int i, c;
     FILE *fin, *fout;

     printf("***Command Line Arguments - File Copy***\n\n");
     printf("Number of arguments, argc = %d\n", argc);
     printf("The arguments are the following strings\n");

     /* argv[0] is the program name, */
     /* argv[1] is the first argument after the program name, etc. */
     for (i = 0; i < argc; i++)
          printf("%s\n", argv[i]);

     fin = fopen(argv[1], "r");
     fout = fopen(argv[2], "w");

     if( !fin || !fout ) exit(1);
     while ((c = fgetc(fin)) != EOF)
          fputc(c, fout);
     exit(0);
}
```

Figure 13.11: File Copy Program Using Command Line Arguments

The function executes the command given by the string `cmmdstr`. it returns 0 if successful, and returns -1 upon failure. Examples include:

```
system("date");
system("time");
system("clear");
```

The first prints the current date, the second prints the current time maintained by the system, and the third clears the screen.

13.5 Summary

In this chapter we have looked at alternate file I/O functions, `fread()` and `fwrite()` which perform *block* I/O; transferring blocks of data directly between memory and data files. This form of I/O is more efficient than *formatted* I/O which converts information between its internal binary representation and the corresponding ASCII representation of the information as strings for the actual I/O. It should be remembered that files used for block I/O have information stored in **binary** and are therefore NOT readable by other programs which do not know the format of the data.

We also saw library routines for controlling the "current position" in the file stream for I/O; namely `ftell()` and `fseek()`. These operations can be performed on either text or binary files.

Finally, we discussed the interactions a program can perform with its environment — the operating system or shell. These include receiving information from the shell in the form of command line arguments which are passed to `main()` as arguments, and the `system()` function which can call on the environment to perform some command.

13.6 Problems

1. Write a program that copies one file to another with file names supplied by the command line.

2. Modify the program in Problem 8 in Chapter 12 to add load and store operations to the student data base program using block I/O.

3. Modify the program in Problem 9 in Chapter 12 to add load and store operations to the club data base program using block I/O.

4. Modify the program in Problem 10 in Chapter 12 to add load and store operations to the library data base program using block I/O.

5. Write a program that serves as a dictionary and thesaurus. A dictionary keeps a meaning for each word. A meaning may be one or more lines of text. A thesaurus keeps a set of synonyms for each word. Assume that the maximum number of entries in the dictionary is 500; there are no more than two lines for a meaning; and there are no more than three synonyms for each word. Allow the user to ask for synonyms, meanings, spell check a text file with repacement of words or add word entries to dictionary. Use files to load and save the dictionary.

Chapter 14

Storage Class and Scope

In previous chapters we have discussed the declaration of variables within functions and described how memory space is allocated by the compiler for these variables as a program executes. How (and where) this memory is allocated, as well as how long it is allocated is determined by what is called the *storage class* for the variable. In addition we have discussed where within the code the variable name is "visible", i.e. where it can be accessed by name. This is called the *scope* of the variable. The variables we have seen so far have all been of storage class *automatic*, i.e. they are allocated when the function is called, and deallocated when it returns, with *local* scope, i.e. visible only within the body of the function. The C language provides several other storage classes together with their scope for controlling memory allocation. In this chapter we will discuss in more detail the concepts of memory allocation and present the other storage classes available in C, viz. *automatic*, *external*, *register*, and *static*. We will also see that functions, as well as variables, have storage class and scope. We next discuss *dynamic* allocation of memory, where a program can determine how much additional memory it needs as it executes. Finally, we introduce *function pointers*, i.e. pointer variables which can hold pointers to functions rather than data. We will see how these pointers are created, stored, passed as parameters, and accessed.

14.1 Storage Classes

Every C variable has a storage class and a scope. The storage class determines the part of memory where storage is allocated for an object and how long the storage allocation continues to exist. It also determines the scope which specifies the part of the program over which a variable name is visible, i.e. the variable is accessible by name. The are four storage classes in C are automatic, register, external, and static.

14.1.1 Automatic Variables

We have already discussed automatic variables. They are declared at the start of a block. Memory is allocated automatically upon entry to a block and freed automatically upon exit from the block. The scope of automatic variables is local to the block in which they are declared, including any blocks nested within that block. For these reasons, they are also called *local variables*. No block outside the defining block may have direct access to automatic variables, i.e. by name. Of course, they may be accessed indirectly by other blocks and/or functions using pointers.

```
/* File: reg.c */
main()
{    register float a = 0;
     auto int bb = 1;
     auto char cc = 'w';

     /* rest of the program */
}
```

Figure 14.1: Code fragment illustrating `register` and `auto` declarations

o

Automatic variables may be specified upon declaration to be of storage class `auto`. However, it is not required; by default, storage class within a block is `auto`. Automatic variables declared with initializers are initialized each time the block in which they are declared is entered.

14.1.2 Register Variables

Register variables are a special case of automatic variables. Automatic variables are allocated storage in the memory of the computer; however, for most computers, accessing data in memory is considerably slower than processing in the CPU. These computers often have small amounts of storage within the CPU itself where data can be stored and accessed quickly. These storage cells are called *registers*.

Normally, the compiler determines what data is to be stored in the registers of the CPU at what times. However, the C language provides the storage class `register` so that the programmer can "suggest" to the compiler that particular automatic variables should be allocated to CPU registers, if possible. Thus, `register` variables provide a certain control over efficiency of program execution. Variables which are used repeatedly or whose access times are critical, may be declared to be of storage class `register`.

Register variables behave in every other way just like automatic variables. They are allocated storage upon entry to a block; and the storage is freed when the block is exited. The scope of register variables is local to the block in which they are declared. Rules for initializations for register variables are the same as for automatic variables.

Figure 14.1 shows a code fragment for a `main()` function that uses `register` as well as `auto` storage class. The class specifier simply precedes the type specifier in the declaration. Here, the variable, `a`, should be allocated to a CPU register by the compiler, while `bb` and `cc` will be allocated storage in memory. Note, the use of the `auto` class specifier is optional.

As stated above, the `register` class designation is merely a suggestion to the compiler. Not all implementations will allocate storage in registers for these variables, depending on the number of registers available for the particular computer, or the use of these registers by the compiler. They may be treated just like automatic variables and provided storage in memory.

Finally, even the availability of register storage does not guarantee faster execution of the program. For example, if too many register variables are declared, or there are not enough registers available to store all of them, values in some registers would have to be moved to temporary storage