in memory in order to clear those registers for other variables. Thus, much time may be wasted in moving data back and forth between registers and memory locations. In addition, the use of registers for variable storage may interfere with other uses of registers by the compiler, such as storage of temporary values in expression evaluation. In the end, use of register variables could actually result in slower execution. Register variables should only be used if you have a detailed knowledge of the architecture and compiler for the computer you are using. It is best to check the appropriate manuals if you should need to use register variables.

14.1.3 External Variables

All variables we have seen so far have had limited scope (the block in which they are declared) and limited lifetimes (as for automatic variables). However, in some applications it may be useful to have data which is accessible from within any block and/or which remains in existence for the entire execution of the program. Such variables are called *global variables*, and the C language provides storage classes which can meet these requirements; namely, the external and static classes.

External variables may be declared outside any function block in a source code file the same way any other variable is declared; by specifying its type and name. No storage class specifier is used — the position of the declaration within the file indicates external storage class. Memory for such variables is allocated when the program begins execution, and remains allocated until the program terminates. Fo rmost C implementations, every byte of memory allocated for an external variable is initialized to zero.

The scope of external variables is global, i.e. the entire source code in the file following the declarations. All functions following the declaration may access the external variable by using its name. However, if a local variable having the same name is declared within a function, references to the name access the local variable cell. Figure 14.2 shows an example of external variables and their scope. The comments in the code indicate which variable is accessed in each reference to the name. The situation is shown graphically in Figure 14.3. Executing the program produces the following sample session:

```
***Scope of External Variables***

a1 = 2
a1 = a, b1 = 77
a1 = 2
a1 = 13, b1 = 19.200001
a1 = 13
```

External variables may be initialized in declarations just as automatic variables; however, the initializers must be constant expressions. The initialization is done only once at compile time, i.e. when memory is allocated for the variables variables.

In general, it is a good programming practice to avoid use of external variables as they destroy the concept of a function as a "black box". The black box concept is essential to the development of a modular program with *independent* modules. With an external variable, any function in the program can access and alter the variable, thus making debugging more difficult as well. This is not to say that external variables should *never* be used. There may be occasions when the use of an external variable significantly simplifies the implementation of an algorithm. Suffice it to say that external variables should be used rarely and with caution.

```
/*   File: glb.c
     This program clarifies the scope of external variables.
*/
#include <stdio.h>
void next(void);
void next1(void);

int a1 = 1;         /* external variable:  global scope */
                    /* scope:  main(), next(), next1() */

main()
{
     printf("***Scope of External Variables***\n\n");
     a1 = 2;                  /* external var */
     printf("a1 = %d\n", a1); /* a1 = 2 */
     next();
     printf("a1 = %d\n", a1); /* a1 = 2 */
     next1();
     printf("a1 = %d\n", a1); /* a1 = 13 */
}

int b1 = 0;         /* external variable */
                    /* scope: global to next, next1 */
                    /* main() cannot access b1 */

void next(void)
{    char a1;       /* auto var: scope local to next()  */
                    /* next() cannot access external a1 */
     a1 = 'a';      /* local auto var */
     b1 = 77;       /* external var */
     printf("a1 = %c, b1 = %d\n", a1, b1);   /* a1 = a, b1 = 77 */
}

void next1(void)
{    float b1;      /* auto var: scope local to next1()  */
                    /* next1() cannot access external b1 */
     b1 = 19.2;     /* auto var */
     a1 = 13;       /* external var */
     printf("a1 = %d, b1 = %f\n", a1, b1);   /* a1 = 13 */
                                             /* b1 = 19.2 */
}
```

Figure 14.2: Example of external variable scope

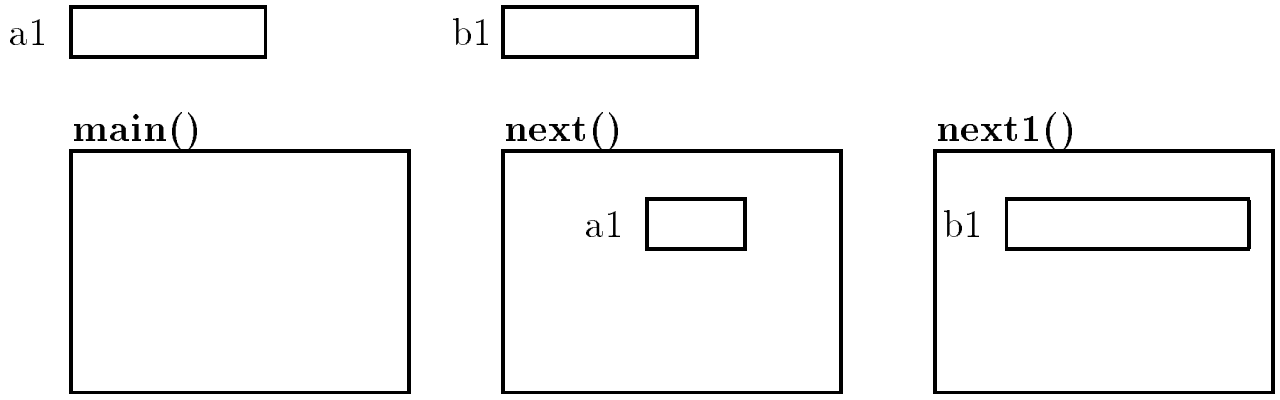

Figure 14.3: Storage allocation for global variables

14.1.4 Variable Definition vs Declaration

Up until now, we have been using the term *declaration* rather loosely when referring to variables. In this section, we will "tighten" the definition of this term. So far when we have "declared" a variable, we have meant that we have told the compiler *about* the variable; i.e. its type and its name, as well as allocated a memory cell for the variable (either locally or globally). This latter action of the compiler, allocation of storage, is more properly called the *definition* of the variable. The stricter definition of *declaration* is simply to describe information "about" the variable.

So far, we have used declarations to *declare* variable names and types as well as to *define* memory for them. Most of the time these two actions occur at the same time, that is, most declarations are definitions; however, this may not always be the case.

We have already seen an analogous case illustrating the difference between *declaring* and *defining* with functions. The prototype statement for a function *declares* it, i.e. tells the compiler "about" the function — its name, return type, and number and type of its parameters. A similar statement, the function header, followed by the body of the function, *defines* the function — giving the details of the steps to perform the function operation.

For automatic and register variables, there is no difference between definition and declaration. The process of declaring an automatic or a register variable defines the variable name and allocates appropriate memory. However, for external variables, these two operations may occur independently. This is important because memory for a variable must be allocated only once, to ensure that access to the variable always refers to the same cell. Thus, all variables must be defined once and only once. If an external variable is to be used in a file other than the one in which it is *defined*, a mechanism is needed to "connect" such a use with the uniquely defined external variable cell allocated for it. This process of connecting the references of the same external variable in different files, is called *resolving the references*.

As we saw in the previous section, external variables may be defined and declared with a declaration statement outside any function, with no storage class specifier. Such a declaration allocates memory for the variable. A declaration statement may also be used to simply *declare* a variable name with the `extern` storage class specifier at the beginning of the declaration. Such a declaration specifies that the variable is *defined* elsewhere, i.e. memory for this variable is allocated in another file. Thus, access to an external variable in a file other than the one in which

it is *defined* is possible if it is *declared* with the keyword `extern`; no new memory is allocated. Such a declaration tells the compiler that the variable is defined elsewhere, and the code is compiled with the external variable left unresolved. The reference to the external variable is resolved during the linking process.

Here are some examples of *declarations* of external variables that are not *definitions*:

```
extern char stack[10];
extern int stkptr;
```

These declarations tell the compiler that the variables `stack[]` and `stkptr` are defined elsewhere, usually in some other file. If the keyword `extern` were omitted, the variables would be considered to be new ones and memory would be allocated for them. Remember, access to the same external variable defined in another file is possible only if the keyword `extern` is used in the declaration. Figure 14.4 shows an example of a source program that references the same external variable in different files. The files are assumed to be compiled separately and linked together to create a load module. A sample run is shown below.

```
***Declaration vs Definition***

a1 = 2
a1 = 13
```

14.1.5 An Example: Lexical Scanner

To illustrate the use of external storage class variables, let us now consider an example in which a good program design is facilitated by the use of an external variable. The task is to find the next *token* in an input stream of characters. A *token* is a useful chunk of characters in the input stream, e.g. an operator, an identifier, an integer, a floating point number, etc. Tokens are also called *symbols*. A function that finds the next token in an input stream and identifies its type is called a *lexical scanner*. For our example, we will write a simple lexical scanner, `get_token()`, to find the next token and its type until an end of file is reached.

We will assume that the only valid tokens in the input stream to be identified by the program are either integers or operators. Further, we assume that integers can have no more than five digit characters and the operator can have no more than a single character. The operators allowed are `+, -, *, /`. If an integer type token exceeds the size limit, an oversize type is to be identified. White space characters between tokens are to be ignored. Any other character is an invalid character which is to be identified as an illegal type of token. Finally, the end of file is to be signaled by an end_of_text type of token.

We assume that `get_token()` determines the next token in the input stream and its type. We use a file `symdef.h` for all the defines. The function prototype for `get_token()` is included in `symtok.h`. The function takes two arguments: a string for the token, and the maximum size of the token. The function returns the type of the token, a symbolic constant with an integer value. The files `symdef.h` and `symtok.h` are shown in Figure 14.5. The logic for the driver is straightforward and the implementation is in the file called `symbol.c` shown in Figure 14.6. A loop is executed as long as there is a new token, and for each iteration, a token and its type are printed. When the end of file is reached, the token type returned by `get_token()` is `EOT`, the loop is terminated and

```
/*   File: ext.c
     This example shows reference to an external variable
     in more than one file. The program is organized in
     three files. The external variable a1 is defined in ext.c,
     and it is declared as extern in FILE3.C.
*/
#include <stdio.h>
void next(void);
void next1(void);
int a1 = 1;              /* definition of external a1 */

main()
{
     printf("***Declaration vs Definition***\n\n");
     a1 = 2;
     printf("a1 = %d\n",a1);       /* a1 = 2 */
     next();                       /* No change in external a1 */
     next1();                      /* external a1 changed to 13 */
     printf("a1 = %d\n", a1);      /* a1 = 13 */
}

/*   File: FILE2.C */
int b1 = 0;         /* definition of external b1 */
void next(void)
{    char a1;       /* auto a1 defined */

     a1 = 'a';      /* only local a1 is visible */
     b1 = 77;       /* external b1 is accessed */
}

/*   File: FILE3.C */
extern int a1;      /* declaration of external a1 */
void next1(void)
{    float b1;      /* auto b1 defined */

     b1 = 19.2;     /* only local b1 is visible */
     a1 = 13;       /* external a1 is accessed */
}
```

Figure 14.4: Example of the use of `extern` declarations

```
/*   File: symdef.h */
     /* Token Types */
     #define INT 0       /* integer */
     #define OPR 1       /* operator */
     #define ILG 2       /* illegal */
     #define EOT 3       /* end of text */
     #define OVR 4       /* oversize */

     #define LIM 5       /* token size limit */

/* File: symtok.h */
int get_token(char * token, int lim);
```

Figure 14.5: Header files for Lexical Scanner

```
/*   File: symbol.c
     Other Source Files: symtok.c, symio.c
     Header Files: symdef.h, symtok.h, symio.h
     This program reads an input stream and determines the tokens in the
     input stream. The primary token types are integer and operator. If
     the integer type token exceeds a specified limit, the token is of
     type oversize. Leading white space is skipped over. All other
     characters are considered to be illegal type tokens. Finally, EOF is
     returned as a special token type to terminate the program.
*/
#include <stdio.h>
#include "symdef.h"
#include "symtok.h"

main()
{    int type;
     char symbol[LIM + 1];

     printf("***Tokens and Types***\n\n");
     printf("Types: integers(0), operators(1), illegal(2),\n");
     printf("       end of text(4), and oversize integers(5)\n");
     printf("Type input text, EOF to quit\n");
     while ((type = get_token(symbol, LIM)) != EOT)
          printf("Token = %8s Type = %8d\n", symbol, type);
}
```

Figure 14.6: Driver for Lexical Scanner

the program ends. The size limit on a token is defined by LIM. The string, symbol, has a size of LIM plus one to accommodate the terminating NULL character.

Here is our logic for get_token(). The function scans the input stream, skipping over any leading white space. The first non-white character determines the type of token to build. For example, if the first non-white character is a digit character, the function builds a token of type INT. The integer type token is built using a loop. As long as the input character is a digit character and the token size limit is not exceeded, the input character is appended to the token string. If the token size limit is exceeded, the type is identified as OVR and the digit is discarded. The process of discarding digits continues until a non-digit character is read. The token string is terminated with a NULL, and the token type is returned. Otherwise, the building of an integer token is terminated when a non-digit character is read. The non-digit character read must somehow be returned to the input stream, so that it is available in building the next token. For example, if the next character is an operator, +, that character must be used in building the next token. If this non-digit character were discarded, it would be lost. Thus, the extra character that was read must be placed back into the input stream to be available once again for building the next token.

We will assume that the desired I/O actions are performed using an "effective input stream". We will write two functions, getchr() and ungetchr(c) for I/O with the effective stream. The function getchr() correctly reads a character from the effective input stream, and ungetchr(c) puts a character, c, back into the effective input stream. Assuming these functions, the algorithm for building an integer type token is simple:

```
if (isdigit(c)) {   /* if c is a digit, */
     type = INT;          /* type is integer */
     while (isdigit(c)) {  /* repeat as long as c is a digit: */
          if (i < lim)        /* if the size limit is not exceeded, */
               s[i++] = c;    /* append the digit char; */
          else type = OVR;    /* otherwise, we have an oversize token */
          c = getchr();  /* get the next input char */
     }
     s[i] = NULL;         /* append the NULL */
     ungetchr(c);         /* put back the extra char read */
}
```

The prototypes for the functions getchr() and ungetchr() are:

```
/* File: symio.h */
int getchr(void);
void ungetchr(int c);
```

Assuming these functions are available in the source file, symio.c, we can implement the function get_token() in Figure 14.7. Finally, we are ready to write the functions getchr() and ungetchr() in a separate file. We will use a buffer to simulate the effective input stream so that when a character is to be returned to the input stream, it is placed in the buffer. When a character is to be read, the buffer is examined first. If there is a character in the buffer, that character is taken as the next input character. If the buffer is empty, a new character is read from standard input using getchar(). Thus, the one character buffer serves as an adjunct to the input stream; getchr() gets the next character either from the buffer or from the standard input, depending on

```
/*   File: symtok.c */
#include <stdio.h>
#include "symdef.h"
#include "symio.h"
#include <ctype.h>
#define TRUE 1
#define FALSE 0

/*   Gets the next token s with a size limit of lim,
     and returns the token type.
*/
int get_token(char s[], int lim)
{    int i, c, type;

     i = 0;                   /* initialize string index i to zero */
     c = getchr();            /* get the first character */
     while (isspace(c))       /* skip over white space */
          c = getchr();
     if (isdigit(c)) {        /* if c is a digit */
          type = INT;         /* type is INT */
          while (isdigit(c)) {     /* Build an INT token */
               if (i < lim)        /* if size limit not exceeded, */
                    s[i++] = c;    /* add the next char to token; */
               else type = OVR;    /* else, type is OVR */
               c = getchr();       /* get next char */
          }
          ungetchr(c);        /* and put back the extra char read. */
     }
     else if (is_op(c)) {     /* if c is an operator */
          s[i++] = c;         /* build an operator token */
          type = OPR;
     }
     else if (c == EOF)       /* if end of file */
          type = EOT;         /* type is EOT */
     else {
          type = ILG;         /* otherwise, we have an illegal char */
          s[i++] = c;         /* a single char string is built */
     }
     s[i] = NULL;             /* terminate the token string */
     return(type);            /* return token type */
}
/* Checks to see if c is an operator */
int is_op(int c)
{
     if (c == '+' || c == '-' || c == '*' || c == '/')
          return(TRUE);
     return(FALSE);
}
```

```
/*   File: symio.c */
#include <stdio.h>
#include "symdef.h"      /* needed for stdio.h */

int c = NULL;            /* buffer c initialized to zero */
                         /* initialization unnecessary */

/*   Gets the next character either from the buffer if there is
     one, otherwise gets a char from stdin.
*/
int getchr(void)
{    int ch;
     if (c) {            /* if c is not a null char, */
          ch = c;        /* save it temporarily, and  */
          c = NULL;      /* reset c to NULL */
          return ch;     /* return the saved value */
     }
     else
          return getchar();   /* else, return a char from stdin */
}

/*   Puts a char into the buffer for later use */
void ungetchr(int cc)
{
     c = cc;        /* save the char cc in the buffer */
}
```

Figure 14.8: Code for implementing the "effective input stream"

the state of the buffer, while `ungetchr()` saves a character into the buffer for later use. Effectively, `getchr()` gets a character from the input stream, and `ungetchr()` returns a character to the input stream. Both `getchr()` and `ungetchr()` must access the buffer. However, `get_token()` should not be concerned with the details of accessing the input stream. Such details should be *hidden* from the rest of the program. Such information hiding is an important component of modular program design. The above case obviously calls for it; thus, `get_token()` should not be involved with the details of maintaining the buffer.

To achieve this information hiding, we put `getchr()` and `ungetchr()` in a separate file together with the external variable used as a one character buffer which is accessible to both `getchr()` and `ungetchr()`. Figure 14.8 shows the implementation. The external variable for the character buffer used in the file `symio.c` makes it unnecessary for other functions to pass a buffer variable as an argument in function calls to `getchr()` and `ungetchr()`. Separation of these functions and the external variable they use into a distinct file makes for a modular program design. No other function needs access to the external variable defined in the file `symio.c`.

A standard library function, `ungetch()`, is available which returns its argument to the keyboard

buffer. We could have also used `ungetch()` and `getchar()` to handle the above tasks of getting and ungetting characters from the keyboard input stream.

A sample run of the program `symbol.c` is shown below:

```
***Tokens and Types***

Types:  integers(0), operators(1), illegal(2),
end of text(4), and oversize integers(5)
Type input text, EOF to quit
123 * 723456 + 12
Token = 123 Type = 0
Token = * Type = 1
Token = 72345 Type = 4
Token = + Type = 1
Token = 12 Type = 0
45+23*7
Token = 45 Type = 0
Token = + Type = 1
Token = 23 Type = 0
Token = * Type = 1
Token = 7 Type = 0
x = 8;
Token = x Type = 2
Token = = Type = 2
Token = 8 Type = 0
Token = ; Type = 2
^D
```

In the first input line, we use blanks to separate the tokens. We also have an oversize token in this case. In the second input line, no blanks are used to separate the tokens. Finally, the last line includes many illegal characters. In each case, the longest possible token is built.

While we caution against the use of external variables as a rule, there are occasions when the use of external variables results in better programs. The deciding factor should always be better program design that provides modularity and flexibility, and that facilitates debugging.

14.1.6 Static Variables

As we have seen, external variables have global scope across the entire program (provided `extern` declarations are used is files other than where the variable is defined), and a lifetime over the the entire program run. The storage class, static, similarly provides a lifetime over the entire program, however; it provides a way to limit the scope of such variables, Static storage class is declared with the keyword `static` as the class specifier when the variable is defined. These variables are automatically initialized to zero upon memory allocation just as external variables are. Static storage class can be specified for automatic as well as external variables.

Static automatic variables continue to exist even after the block in which they are defined terminates. Thus, the value of a static variable in a function is retained between repeated function

calls to the same function. The scope of static automatic variables is identical to that of automatic variables, i.e. it is local to the block in which it is defined; however, the storage allocated becomes permanent for the duration of the program. Static variables may be initialized in their declarations; however, the initializers must be constant expressions, and initialization is done only once at compile time when memory is allocated for the static variable.

Figure 14.9 shows an example which sums integers, using static variables. Function `sumit()` reads a new integer and keeps a cumulative sum of the previous value of the sum and the new integer read in. The cumulative value of sum is kept in the static variable, `sum`. The driver, `main()` calls `sumit()` five times to sum five integers.

Sample Session:

```
***Static Variables***

Please enter 5 numbers to be summed
Enter a number:  12
The current total is 12
Enter a number:  23
The current total is 35
Enter a number:  34
The current total is 69
Enter a number:  45
The current total is 114
Enter a number:  56
The current total is 170
Program completed
```

While the static variable, `sum`, would be automatically initialized to zero, it is better to do so explicitly. In any case, the initialization is performed only once at the time of memory allocation by the compiler. The variable `sum` retains its value during program execution. Each time the function `sumit()` is called, `sum` is incremented by the next integer read.

Static storage class designation can also be applied to external variables. The only difference is that static external variables can be accessed as external variables only in the file in which they are defined. No other source file can access static external variables that are defined in another file.

```
/* File: xxx.c */
static int count;
static char name[8];
main()
{
     ... /* program body */
}
```

Only the code in the file `xxx.c` can access the external variables count and name. Other files cannot access them, even with `extern` declarations.

We have seen that external variables should be used with care, and access to them should not be available indiscriminately. Defining external variables to be static provides an additional

```
/*   File: static.c */
/*   Program uses a function to sum integers. The function
     uses a static variable to store the cumulative sum.
*/
#include <stdio.h>
#define MAX 5
void sumit(void);

main()
{    int count;

     printf("***Static Variables***\n\n");
     printf("Please enter 5 numbers to be summed\n");
     for (count = 0; count < MAX; count++)
          sumit();
     printf("Program completed\n");
}

/*   Function reads an integer, and keeps cumulative sum of
     integer read and the previous value of a static variable sum.
*/
void sumit(void)
{    static int sum = 0;      /* sum is initialized to zero  */
                              /* at compile time. */
     int num;

     printf("Enter a number: ");
     scanf("%d",&num);
     sum += num;
     printf("The current total is %d\n",sum);
}
```

Figure 14.9: An example of static variables

```
/*   File: symio2.c */
#include <stdio.h>
#include "symdef.h"

static int c = NULL;     /* static external c */

/*   Gets the next character either from the buffer if there is
     one, otherwise gets a char from stdin.
*/
int getchr()
{  int ch;

     if (c) {            /* if c is not a null char, */
          ch = c;        /* save it temporarily, and  */
          c = NULL;      /* reset c to zero */
          return(ch);    /* Return the saved value */
     }
     else
          return(getchar());  /* else, return a char from stdin */
}

/*   Puts a char into the buffer for later use */
void ungetchr(int cc)
{
     c = cc;        /* save the char cc in the buffer */
}
```

Figure 14.10: Revised file `symio.c` using `static` variable

control on which functions can access them. For example, in the `symbol.c` example in the last section, we created a file `symio.c` which contained an external variable. This external variable should be accessible only to the functions in that file. However, there is no way to guarantee that some other file may not access it by declaring it as `extern`. We can ensure that this will not happen by declaring the variable as `static` as shown in Figure 14.10. The static variable `c` would not be accessible to functions defined in any other file, thus preventing an unplanned use of it as an external variable by the code in other files.

14.1.7 Storage Class for Functions

Like variables, functions in C have a storage class and scope. All functions in C are external by default and are accessible to all source files. However, functions may be declared to be of static class, in which case they are accessible only to functions in the file in which they are defined, not to functions in other files. This is another way of hiding information. Information hiding makes these static function names invisible to all other files; thus, these names may be used to define

other functions elsewhere.

Here is an example that uses static variables as well as a static function. The program assigns bins to different part numbers. The array, `index`, represents the bin number where the part number is stored (it is easy to generalize the program to structures). The program is organized in two files, `bins.c` and `binutil.c`. The first file, `bins.c`, contains the driver which reads in the part numbers, and calls a function, `getbin()`, to assign a bin number to each part number. Finally, the driver prints the bins and the corresponding part numbers using the function `printbin()`. Here are the prototypes:

```
/*   File: binutil.h */
void getbin(int bin[], int part, int lim);
void printbin(int bin[], int lim);
```

The function `getbin()` needs three arguments: an array of bins, a part number, and the array size limit. The bin number is just the array index, so `getbin()` assigns one of the bins in the array to the part number, and stores the part number in the array at the corresponding bin number index. The function `printbin()` needs the array of bins and its size as arguments. It prints out each bin number index and the corresponding part number stored at that array index. The driver is shown in Figure 14.11. The program loop reads a part number and if it is not zero, it calls `getbin()` to assign a bin number to the part number. If the part number is zero, the loop terminates, and `printbin()` prints bin numbers and corresponding part numbers.

Let us now implement `getbin()`. Unused array elements of bin should be initialized to some invalid part number, say -1, so that `printbin()` would be able to distinguish the valid elements of the array. The first time `getbin()` is called, it calls `initbin()` which initializes bin to -1. In addition, `getbin()` should assign the next available index to the part number. The functions are shown in Figure 14.12. The function `getbin()` uses a static variable, `first`, initialized to `TRUE`, to determine if the function is being called for the first time. When the function is called the first time, it initializes the array, `bin` and changes first to `FALSE`. A second static variable, `bin_number` is used to remember the next available bin number between function calls. As a bin is assigned to a part number, `bin_number` is incremented. Since it is a static variable, its latest value is available each time the function is called. The function `printbin()` merely prints each array index and the part number stored at that index. Initialization of the array bin is done by a static function `initbin()`. This function is not required anywhere else, and so a static class is declared for it, thus the details of array initialization are hidden from all other functions. A sample run of the program is shown below:

```
***Bin Assignments to Parts***

Type part numbers, enter zero to quit
Enter part number:  1523
Enter part number:  234
Enter part number:  725
Enter part number:  9120
Enter part number:  0
Bin number 0 has part number 1523
Bin number 1 has part number 234
```

```
/*   File: bins.c
     Other Source Files: binutil.c
     Header Files: binutil.h
     This program assigns a unique bin number to each part number. The
     user types the part numbers and the program assigns bin numbers
     to the parts in sequence. A zero part number terminates the program.
     It is also assumed that the user types only new part numbers. No
     check is made to see if a part number is already assigned a bin.
*/
#include <stdio.h>
#include "binutil.h"     /* prototypes for getbin(), printbin() */
#define MAX 100

main()
{    int bin[MAX], part_no;

     printf("***Bin Assignments to Parts***\n\n");
     printf("Type part numbers, enter zero to quit\n");
     do {
          printf("Enter part number: ");
          scanf("%d",&part_no);
          if (part_no)
               getbin(bin, part_no, MAX);
     } while (part_no);
     printbin(bin, MAX);
}
```

Figure 14.11: Driver for bins program

```
/*   File: binutil.c */
#include <stdio.h>
#include "binutil.h"     /* prototypes for getbin(), printbin() */
#define TRUE 1
#define FALSE 0
static void initbin(int bin[], int lim);

/*   Initializes an array bin of size lim. The function
     is declared static since no other file needs it.
*/
static void initbin(int bin[], int lim)
{    int i;

     for (i = 0; i < lim; i++)
          bin[i] = 0;
}

/*   Assigns a bin element to a part number. First time it
     is called, it initializes the array bin[].
*/
void getbin(int bin[], int part, int lim)
{    static int first = TRUE;
     static bin_number = 0;

     if (first) {
          initbin(bin, lim);
          first = FALSE;
     }
     if (bin_number < lim)
          bin[bin_number++] = part;
     else
          printf("Error - out of Part Bins\n");
}

/*   Prints out bin numbers and part numbers. */
void printbin(int bin[], int lim)
{    int i;

     for (i = 0; i < lim && bin[i]; i++)
          printf("Bin number %d has part number %d\n",i,bin[i]);
}
```

Figure 14.12: Code for bin utilities

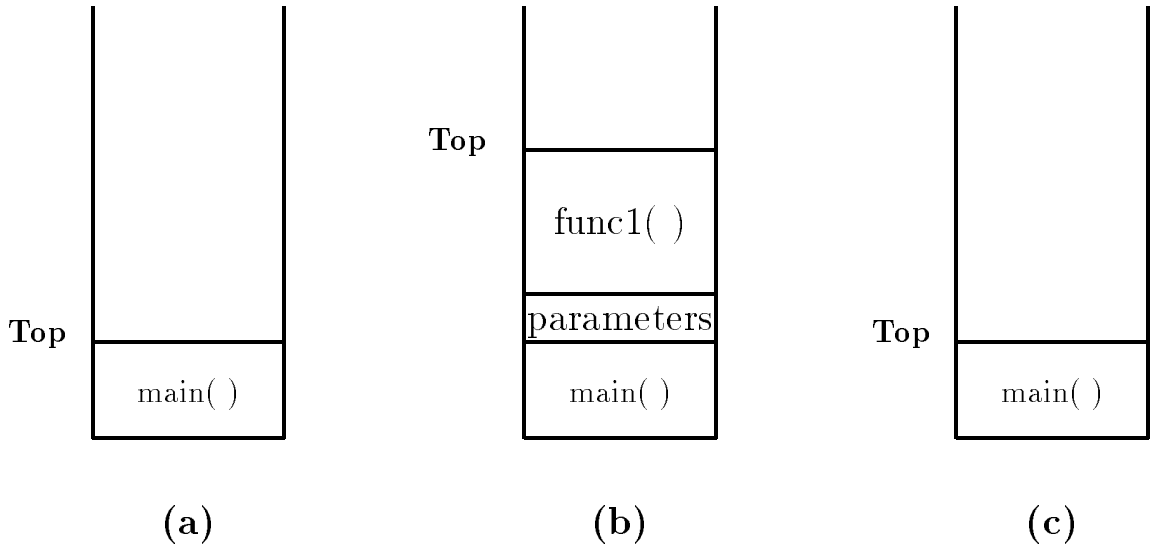

Figure 14.13: Organization of the Stack

```
Bin number 2 has part number 725
Bin number 3 has part number 9120
```

14.1.8 Stack vs Heap Allocation

We conclude our discussion of storage class and scope by breifly describing how the memory of the computer is organized for a running program. When a program is loaded into memory, it is organized into three areas of memory, called *segments*: the *text segment*, *stack segment*, and *heap segment*. The text segment (sometimes also called the code segment) is where the compiled code of the program itself resides. This is the machine language representation of the program steps to be carried out, including all functions making up the program, both user defined and system.

The remaining two areas of system memory is where storage may be allocated by the compiler for data storage. The stack is where memory is allocated for automatic variables within functions. A stack is a *First In First Out* (FIFO) storage device where new storage is allocated and deallocated at only one "end", called the Top of the stack. This can be seen in Figure 14.13. When a program begins executing in the function `main()`, space is allocated on the stack for all variables declared within `main()`, as seen in Figure 14.13(a). If `main()` calls a function, `func1()`, additional storage is allocated for the variables in `func1()` at the top of the stack as shown in Figure 14.13(b). Notice that the parameters passed by `main()` to `func1()` are also stored on the stack. If `func1()` were to call any additional functions, storage would be allocated at the new Top of stack as seen in the figure. When `func1()` returns, storage for its local variables is deallocated, and the Top of the stack returns to to position shown in Figure 14.13(c). If `main()` were to call another function, storage would be allocated for that function at the Top shown in the figure. As can be seen, the memory allocated in the stack area is used and reused during program execution. It should be clear that memory allocated in this area will contain garbage values left over from previous usage.

The heap segment provides more stable storage of data for a program; memory allocated in the heap remains in existence for the duration of a program. Therefore, global variables (storage

class external), and static variables are allocated on the heap. The memory allocated in the heap area, if initialized to zero at program start, remains zero until the program makes use of it. Thus, the heap area need not contain garbage.

14.2 Dynamic Memory Allocation

In the previous section we have described the the storage classes which determined how memory for variables are allocated by the compiler. When a variable is defined in the source program, the type of the variable determines how much memory the compiler allocates. When the program executes, the variable consumes this amount of memory regardless of whether the program actually uses the memory allocated. This is particularly true for arrays. However, in many problems, it is not clear at the outset how much memory the program will actually need. Up to now, we have declared arrays to be "large enough" to hold the maximum number of elements we expect our application to handle. If too much memory is allocated and then not used, there is a waste of memory. If not enough memory is allocated, the program is not able to handle the input data.

We can make our program more flexible if, during execution, it could allocate additional memory when needed and free memory when not needed. Allocation of memory during execution is called *dynamic memory allocation*. C provides library functions to allocate and free memory dynamically during program execution. Dynamic memory is allocated on the heap by the system.

It is important to realize that dynamic memory allocation also has limits. If memory is repeatedly allocated, eventually the system will run out of memory.

14.2.1 Library Functions for Dynamic Allocation

Two standard library functions are available for dynamic allocation. The function `malloc()` allocates memory dynamically, and the function `free()` deallocates the memory previously allocated by `malloc()`. When allocating memory, `malloc()` returns a pointer which is just a byte address. As such, it does not point to an object of a specific type. A pointer type that does not point to a specific data type is said to point to `void` type, i.e. the pointer is of type `void *`. In order to use the memory to access a particular type of object, the void pointer must be cast to an appropriate pointer type. Here are the descriptions for `malloc()`, and `free()`:

malloc — *Prototype:* `void * malloc(unsigned size);` *in:* <`stdlib.h and alloc.h`>

Returns: void pointer to the allocated block of memory if successful, `NULL` otherwise

Description: Returned pointer must be cast to an appropriate type.

free — *Prototype:* `void free(void * ptr);` *in:* <`stdlib.h and alloc.h`>

Returns: none

Description: ptr must be a pointer to previously allocated block of memory

If successful, `malloc()` returns a pointer to the block of memory allocated. Otherwise, it returns a `NULL` pointer. One must always check to see if the pointer returned is `NULL`. If `malloc()` is successful, objects in dynamically allocated memory can be accessed indirectly by dereferencing the pointer, appropriately cast to the type of pointer required.

The size of the memory to be allocated must be specified, in bytes, as an argument to `malloc()`. Since the memory required for different objects is implementation dependent, the best way to specify the size is to use the `sizeof` operator. Recall that the `sizeof` operator returns the size, in bytes, of the operand.

For example, if the program requires memory allocation for an integer, then the size argument to `malloc()` would be `sizeof(int)`. However, in order for the pointer to access an integer object, the pointer returned by `malloc()` must be cast to an `int *`. The code takes the following form:

```
int *ptr;
ptr = (int *)malloc(sizeof(int));
```

Now, if the pointer returned by `malloc()` is not `NULL`, we can make use of it to access the memory indirectly. For example:

```
if (ptr != NULL)
     *ptr = 23;
```

Or, simply,

```
if (ptr)
     *ptr = 23;
printf("Value stored is %d\n", *ptr);
```

Later, memory allocated above may no longer be needed. In which case, it is important to free the memory. Thus:

```
free((void *) ptr);
```

deallocates the previously allocated block of memory pointed to by `ptr`. Or, more simply, we could write:

```
free(ptr);
```

`ptr` is first converted to `void *` in accordance with the function prototype, and then the block of memory pointed to by `ptr` is freed.

It is possible to allocate a block of memory for several elements of the same type by giving the appropriate value as an argument. Suppose, we wish to allocate memory for 100 float numbers. Then, if `fptr` is a `float *`, the following statement does the job:

```
fptr = (float *) malloc(100 * sizeof(float));
```

Pointer `fptr` points to the beginning of the memory block allocated, i.e. to the first object of the block of 100 float objects, `fptr + 1` points to the next float object, and so on. In other words, we have a pointer to an array of float type. The above approach can be used with data of any type including structures. The example in Figure 14.14 allocates memory for a structure, reads data into it, and then prints the data.

Sample Session:

```
***Dynamic Memory Allocation***

Student Name:  James J. Hillary
Student ID: 723
Student Name:  James J. Hillary ID: 723
```

```
/*   File: dynstruct.c
     This program uses dynamic allocation of a block of memory
     for an element of type stdrec structure. It then stores data
     for one student in the memory block, and prints out the data.
*/
#include <stdio.h>
#include <stdlib.h>

struct stdrec {
     char name[20];
     int id;
};

main()
{    struct stdrec * p;

     printf("***Dynamic Memory Allocation***\n\n");
     p = (struct stdrec *)malloc((sizeof(struct stdrec)));
     if (p) {
          printf("Student Name: ");
          gets(p->name);
          printf("Student ID: ");
          scanf("%d%*c", &p->id);
          printf("Student Name: %-10s  ", p->name);
          printf("ID: %4d\n", p->id);
     }
     else
          printf("Out of Memory\n");
}
```

Figure 14.14: Example program using a dynamic structure

14.2.2 Dynamic Arrays

Our next example allocates a block of memory dynamically for a number of elements of structure type. It reads data into the elements and prints the data. Once the returned pointer is cast to an appropriate type, the allocated memory block may be treated as an array of elements, with the returned pointer a pointer to the array. The code is shown in Figure 14.15.

Sample Session:

```
***Dynamic Arrays - Student Records***

Number of students:  2
Student Name:  James J. Hillary
Student ID: 723
Student Name:  John Paul Jones
Student ID: 321
^D
Student Name:  James J. Hillary ID: 723
Student Name:  John Paul Jones ID: 321
```

Dynamic memory allocation can also be performed by the library function `calloc()`, and the allocated memory freed as before by `free()`. All bytes in memory allocated by `calloc()` are cleared to zero, whereas memory allocated by `malloc()` is left unchanged. The description for calloc() is:

calloc — *Prototype:* `void * calloc(unsigned number, unsigned size);` *in:* <`stdlib.h` and `alloc.h`>

Returns: void pointer to the allocated block of memory if successful, `NULL` otherwise

Description: Returned pointer must be cast to an appropriate type.

```
Example:
     void * ptr;          /* pointer to allocated block of memory */
     unsigned number;     /* number of elements to allocate */
     unsigned size;       /* size of memory to allocate in bytes */

     ptr = calloc(number, size);
```

We could have used `calloc()` in the previous program example as follows:

```
p = (struct stdrec *)calloc(n, sizeof(struct stdrec));
```

We could have then used the fact that the allocated memory is set to zero to signal the end of the number of elements in the effective array.

Normally, an array is defined with the range for each dimension specified, and memory is allocated at compile time. As we saw above, a single dimensional array of a desired size can be effectively defined at run time, i.e. during execution, using dynamic allocation. It is equally easy to define multi-dimensional arrays during execution by using dynamic allocation.

We first allocate an appropriate block of memory for the two dimensional array size desired. Since array storage in C is in row major form, we then treat the block as a sequence of rows with

```
/*   File: dynaray.c
     This program shows dynamic allocation of a block of memory
     for elements of the type struct stdrec. This is equivalent
     to allocating memory for an array of the specified size.
     The program reads in the number of students, allocates memory
     for that many structures, gets data for the students, and prints
     out the data.
*/
#include <stdio.h>
#include <stdlib.h>

struct stdrec {
     char name[20];
     int id;
};
void getdata(struct stdrec * p, int n);
void printdata(struct stdrec * p, int n);

main()
{    int n;
     struct stdrec * p;

     printf("***Dynamic Arrays - Student Records***\n\n");
     printf("Number of students: ");
     scanf("%d%*c", &n);
     p = (struct stdrec *)malloc(n * (sizeof(struct stdrec)));
     if (p) {
          getdata(p, n);
          printdata(p, n);
     }
     else
          printf("Out of Memory\n");
}
```

```
/* Gets data for n students */
void getdata(struct stdrec * p, int n)
{    int id, i;

     for (i = 0; i < n; i++) {
          printf("Student Name: ");
          gets(p->name);
          printf("Student ID: ");
          scanf("%d%*c", &p->id);
          p++;
     }
}

/* Prints data for all students */
void printdata(struct stdrec * p, int n)
{    int i;

     for (i = 0; i < n; i++) {
          printf("Student Name: %-10s  ", p->name);
          printf("ID: %4d\n", p->id);
          p++;
     }
}
```

Figure 14.15: Example code for a dynamic array of structures

```
/*   File: dyn2aray.c
     This program shows a dynamic specification of array size for
     a two dimensional array. Appropriate block of memory is then
     allocated. This block is then treated as a two dimensional
     array of the size specified.
*/

#include <stdio.h>
#include <stdlib.h>
void get2data(int * p, int rows, int cols);
void print2data(int * p, int rows, int cols);

main()
{    int cols, rows;
     int  *p;

     printf("***Dynamic Arrays - Two Dimensions***\n\n");
     printf("Type number of rows: ");
     scanf("%d", &rows);
     printf("Type number of columns: ");
     scanf("%d", &cols);
     p = (int *)malloc(rows * cols * sizeof(int));
     get2data(p, rows, cols);
     print2data(p, rows, cols);
}
```

the desired number of columns. The pointer to the allocated block is a pointer to the base type of the array; therefore, it must be incremented to access the next column in a given row. It must also be incremented to move from the last column of a row to the first column of the next row.

Figure 14.16 shows an example that asks the user to specify the number of rows and columns for a two dimensional array. It then dynamically allocates a block of memory to accommodate the array. The block is then treated as a two dimensional array with the specified rows and columns. Data is read into the array, and then the array is printed. A sample output is shown below:

```
***Dynamic Arrays - Two Dimensions***

Type number of rows:  2
Type number of columns:  3
Type a row of integers with 3 columns:  1 2 3
Type a row of integers with 3 columns:  4 5 6
The array is:

               1      2      3
               4      5      6
```

```
/*   Gets data for a two dimensional array pointed to by int *, p,
     with specified rows and cols.
*/
void get2data(int * p, int rows, int cols)
{    int i, j;

     for (i = 0; i < rows; i++) {
          printf("Type a row of integers with %d columns: ", cols);
          for (j = 0; j < cols; j++) {
               scanf("%d", p);
               p++;
          }
     }
}

/*   Prints data in an array pointed to by int *, p, with
     specified rows and cols.
*/
void print2data(int * p, int rows, int cols)
{    int i, j;

     printf("The array is:\n");
     for (i = 0; i < rows; i++) {
          for (j = 0; j < cols; j++) {
               printf("%7d", *p);
               p++;
          }
          printf("\n");
     }
}
```

Figure 14.16: Dynamic allocation for 2D arrays

14.3 Pointers to Functions

We saw earlier that functions have a storage class and scope, similar to variables. In C, it is also possible to define and use function pointers, i.e. pointer variables which point to functions. Function pointers can be declared, assigned values and then used to access the functions they point to. Function pointers are declared as follows:

```
int (*fp)();
double (*fptr)();
```

Here, `fp` is declared as a pointer to a function that returns `int` type, and `fptr` is a pointer to a function that returns `double`. The interpretation is as follows for the first declaration: the dereferenced value of `fp`, i.e. `(*fp)` followed by `()` indicates a function, which returns integer type. The parentheses are essential in the declarations. The declaration without the parentheses:

```
int *fp();
```

declares a function `fp` that returns an integer pointer.

We can assign values to function pointer variables by making use of the fact that, in C, the name of a function, used in an expression by itself, is a pointer to that function. For example, if `isquare()` and `square()` are declared as follows:

```
int isquare(int n);
double square(double x);
```

the names of these functions, `isquare` and `square`, are pointers to those functions. We can assign them to pointer variables:

```
fp = isquare;
fptr = square;
```

The functions can now be accessed, i.e. called, by dereferencing the function pointers:

```
m = (*fp)(n);       /* calls isquare() with n as argument */
y = (*fptr)(x);     /* calls square() with x as argument */
```

Function pointers can be passed as parameters in function calls and can be returned as function values. Use of function pointers as parameters makes for flexible functions and programs. An example will illustrate the approach. Suppose we wish to sum integers in a specified range from x to y. We can easily implement a function to do so:

```
/*   File: sumutil.h */
int sum_int(int x, int y);

/*   File: sumutil.c */
#include <stdio.h>
#include "sumutil.h"
/*   Function sums integers from x to y. */
int sum_int(int x, int y)
{    int i, cumsum = 0;
```

```
     for (i = x; i <= y; i++)
          cumsum += i;
     return cumsum;
}
```

The file `sumutil.h` contains prototypes for all the functions written in `sumutil.c`. Next, suppose we wish to sum squares of integers from x to y. We must write another function to do so:

```
/*   File: sumutil.h - continued */
int sum_squares(int x, int y);
int isquare(int x);

/*   File: sumutil.c - continued */
/*   Function sums squares of integers form x to y. */
int sum_squares(int x, int y)
{    int i, cumsum = 0;

     for (i = x; i <= y; i++)
          cumsum += isquare(i);
     return cumsum;
}

/*   Function returns the square of x. */
int isquare(int x)
{
     return x * x;
}
```

Function `isquare()` returns the integer square of `i`. The constructions of the two functions `sum_int()` and `sum_squares()` are identical. In both cases, we cumulatively add either the integers themselves or squares of the integers. A function `iself()`, which returns the value of the integer argument, can be used in `sum_int()` to make the functions truly identical. Here is a modified function that uses `iself()`:

```
/*   File: sumutil.h - continued */
int sum_integers(int x, int y);
int iself(int x);

/*   File: sumutil.c - continued */
/*   Function sums integers from x to y. */
int sum_integers(int x, int y)
{    int i, cumsum = 0;

     for (i = x; i <= y; i++)
          cumsum += iself(i);
     return cumsum;
```

```
}

/*   Function returns the argument x. */
int iself(int x)
{
     return x;
}
```

The two sum functions, `sum_integers()` and `sum_squares()`, are now identical except for the functions used in the cumulative sum expressions. In one case, we use `iself()`, in the other case, `isquare()`. It is clear that a single more flexible *generic* sum function can be written by passing a function pointer, `fp`, as an argument with a value pointing to the appropriate function to use. The cumulative sum expression would then take the form:

```
cumsum += (*fp)(i);
```

Here is the implementation:

```
/*   File: sumutil.h - continued */
int sum_gen(int (*fp)(), int x, int y);

/*   File: sumutil.c - continued */
/*   Function sums values of *fp applied to integers from x to y. */
int sum_gen(int (*fp)(), int x, int y)
{    int i, cumsum = 0;

     for (i = x; i <= y; i++)
          cumsum += (*fp)(i);
     return cumsum;
}
```

Finally, we can improve the generic sum function by using a pointer to a function that updates the integer using a specified step size:

```
/*   File: sumutil.h - continued */
int sum(int (*fp)(), int x, int (*up)(), int step, int y);

/*   File: sumutil.c - continued */
/*   Function returns the sum of function *fp applied
     to integers from x to y, incremented by *up in step size.
*/
int sum(int (*fp)(), int x, int (*up)(), int step, int y)
{    int i, cumsum = 0;

     for (i = x; i <= y; i = (*up)(i, step))
          cumsum += (*fp)(i);
     return cumsum;
}
```

The function pointed to by `(*up)` takes two arguments, an integer to be updated and the step size. The generic function `sum()` can now be used to sum `(*fp)(i)` applied to integers `i`, which are updated by `(*up)(i, step)`. The pointer variable, `fp` can point to any function that processes an integer and returns an integer. Similarly, `up` can point to any function that returns an updated integer value.

Let us now write a program that reads starting and ending integers as well as step size until `EOF`. For each set of data read, the program first computes and prints the sum of integers using `sum_int`, and sum of squares using `sum_squares()`. These two sums are in steps of one, since that is how the functions are written. Next, the program uses the above generic sum function `sum()` to compute sums of integers and squares in specified step sizes. Figure 14.17 shows the program. The update function used is `iincr()`, which merely returns `x` plus the step size. The program source files, `sums.c` and `sumutil.c`, are compiled separately and linked together. A sample run of the program is shown below:

Sample Session:

```
***Function Pointers - Sums of Integer Function Values***

Type starting, ending limits, and step size, EOF to quit
3 7 1
Sum of integers from 3 to 7 in steps of 1 = 25
Sum of squares from 3 to 7 in steps of 1 = 135
Sum of integers from 3 to 7 in steps of 1 is 25
Sum of squares from 3 to 7 in steps of 1 is 135
3 7 2
Sum of integers from 3 to 7 in steps of 1 = 25
Sum of squares from 3 to 7 in steps of 1 = 135
Sum of integers from 3 to 7 in steps of 2 is 15
Sum of squares from 3 to 7 in steps of 2 is 83
3 7 3
Sum of integers from 3 to 7 in steps of 1 = 25
Sum of squares from 3 to 7 in steps of 1 = 135
Sum of integers from 3 to 7 in steps of 3 is 9
Sum of squares from 3 to 7 in steps of 3 is 45
^D
```

For each set of input data, the output first shows sums of integers and squares in steps of one, and then in specified steps.

14.3.1 Function Pointers as Returned Values

It is also possible for functions to return a function pointer as a value. This ability increases the flexibility of programs. We will use a simple example to implement a function that returns a function pointer. The example is merely illustrative, and it would be easy to write a program to perform the same task without the use of a function pointer. Let us define a type, which is a pointer to a function that returns an integer.

```
typedef int (*PFI)();
```

```
/*   File: sums.c
     Other Source Files: sumutil.c
     Header Files: sumutil.h
     This program illustrates the use of function pointers to define a
     single function sum() that sums powers of integers between specified
     limits. The function is then applied to sum integers and squares.
     Individual functions to sum integers and squares are also implemented.
     The results are printed out for both approaches.
*/

#include <stdio.h>
#include "sumutil.h"

main()
{    int x, y, step, isquare(), iself(), iincr();

     printf("***Function Pointers - Sums of Function Values***\n\n");
     printf("Type starting, ending limits, and step size, EOF to quit\n");
     while (scanf("%d %d %d", &x, &y, &step) != EOF) {
          printf("Sum of integers from %d to %d in steps of 1 = %d\n",
                    x, y, sum_int(x, y));
          printf("Sum of squares from %d to %d in steps of 1 = %d\n",
                    x, y, sum_squares(x, y));
          printf("Sum of integers from %d to %d in steps of %d is %d\n",
                    x, y, step, sum(iself, x, iincr, step, y));
          printf("Sum of squares from %d to %d in steps of %d is %d\n",
                    x, y, step, sum(isquare, x, iincr, step, y));
     }
}

  /*   File: sumutil.h - continued */
  int iincr(int x, int step);

/*   File: sumutil.c - continued */
/*   Increments x by size of step. */
int iincr(int x, int step)
{
     return x + step;
}
```

Figure 14.17: Program illustrating function pointers

We can now use `PFI` as a data type in declaring variables, parameters, and returned values.

Our example program repeatedly reads an integer until `EOF`. If an integer is odd, the program computes its cube; otherwise, the program computes its square. For each integer, we call a function `evenodd()` which returns a function pointer either to `icube()` or to `isquare()` depending on whether the integer is odd or even. The function pointer returned by `evenodd()` and the integer itself are both passed to a function `process()`, which applies the dereferenced function pointer to the integer. The result is then printed. Figure 14.18 shows the program driver. For each integer, the program calls `evenodd()` to get a returned function pointer which is assigned to `fptr`. Then, it calls `process()` to apply `(*fptr)` to `x`. The result is then printed. Let us now write the function `evenodd()` that takes an integer as an argument. If the argument is odd, the function returns a pointer to `icube()`; otherwise, it returns a pointer to `isquare()`. The function `evenodd()`, together with functions `process()` and `icube()` are also shown in Figure 14.18.

When the program files `fptr.c` and `sumutil.c` are compiled and linked, the sample session is:

```
***Function Pointers - Squares and Cubes***

Type integers, EOF to quit
3
Integer = 3, power 2 or 3 = 27
5
Integer = 5, power 2 or 3 = 125
4
Integer = 4, power 2 or 3 = 16
^D
```

Using function pointers as parameters we can write generic functions. By returning function pointers, the called functions can select the functions that must be used in different circumstances. Function pointers help make a program compact as well as intelligent.

14.4 Summary

In this chapter we have discussed the concepts of storage class and scope for variables in a C program. The language provides four storage classes: automatic, register, external, and static. By default, variables declared in functions are of class `auto`, meaning that memory is allocated for them when the block is entered and automatically deallocated when the block is exited. Such variables may be referenced by name only within the block in which they are declared; i.e. they have *local* scope. Register storage class, declared with the class specifier, `register`, are a special case of automatic variables. This class suggests to the compiler that storage for the variable should be allocated in the CPU registers rather than memory. Use of this class should be limited to frequently referenced, time critical variables and only with familiarity with the particular architecture on which the program will be run.

External storage class is used for variables which should remain allocated for the entire execution of a program, and which have *global* scope. In using external variables, the operation of *defining* the variable (allocating memory for it) may be independent of *declaring* the variable (associating a name with the variable). An external variable must be defined exactly once, by

```
/*   File: fptr.c
     Other Source Files: sumutil.c
     Header Files: sumutil.h
     This program illustrates the use of function pointers, both as
     parameters in function calls and as returned values. Program
     reads integers until EOF. As each integer is read, the program
     calls a function evenodd() which returns a function pointer.
     This function pointer is then passed to process() to process
     the integer.

     evenodd() returns a pointer to isquare() if the argument is even,
     and to icube() otherwise. Function process() applies its first
     argument, which is a function pointer, to its second argument, which
     is an integer.
*/

#include <stdio.h>
#include "sumutil.h"
typedef int (*PFI)();
PFI evenodd(int x);
int process(PFI fp, int x);

main()
{    int x, y, z;
     PFI fptr;

     printf("***Function Pointers - Squares and Cubes***\n\n");
     printf("Type integers, EOF to quit\n");
     while (scanf("%d", &x) != EOF) {
          fptr = evenodd(x);
          y = process(fptr, x);
          printf("Integer = %d, power 2 or 3 = %d\n", x, y);
     }
}

/*   Function returns a function pointer. If x is odd, it returns
     a pointer to icube(). Otherwise, it returns a pointer to
     isquare().
*/
PFI evenodd(int x)
{    int isquare(), icube();

     if (x % 2)
          return icube;  /* icube is a pointer to function icube() */
     else
          return isquare; /* isquare is a pointer to isquare() */
}
```

```
/*   Function returns the result of applying the dereferenced function
     pointer fp to x.
*/
int process(PFI fp, int x)
{
     return (*fp)(x);    /* dereferenced fp applied to x */
}

/*   File: sumutil.h - continued */
int icube(int x);

/*   File: sumutil.c - continued */
/*   Function returns the cube of x. */
int icube(int x)
{
     return x * x * x;
}
```

Figure 14.18: Driver illustrating function pointer return values

specifying its type and name outside any function block. A declaration specified as `extern` declares the name of the variable without allocating storage, with the expectation that it has been defined elsewhere.

The storage class, *static*, is used for variables which have local scope, but which remain allocated for the entire program execution. Such variables, while local to a particular function, will retain their values across repeated calls and returns.

We have also seen how memory for variables of different storage classes is allocated in the memory of the computer: automatic variables are allocated and deallocated on the stack, whereas external and static variables are allocated from the heap.

In addition to storage allocated by the compiler, we have seen that additional storage can be allocated *dynamically* (i.e. at run time) using the `malloc()` or `calloc` system functions, and deallocated by the `free()` function. Data stored in dynamically allocated memory is always referenced indirectly.

Finally, we have expanded our discussion of functions; seeing that they have storage class, like variables; and that we can declare and access function indirectly through pointers. Functions are generally external and have global scope. However, we can limit the scope of a function to be within a single source file by declaring it to be of static storage class.

14.5 Problems

1. Modify the functions `getchr()` and `ungetchr()` of Section 14.1.6 so that any number of characters, up to a maximum of 40, can be put back into the input stream. Use these functions in a program that reads characters and puts all vowels back until a newline is read. At that point, the program writes the vowels that were put back in the input stream.

2. Write a program that reads scores from a file, but uses a dynamically allocated array. Assume that the first line of the file has the number of students. Read the value of the number of students, dynamically allocate an array for the scores, read the scores, and print them out.

3. Modify 2 so a student record is a structure. The file lists the number of students in the first line and the number of exams in the second line. Assume that an old weighted average is present as the last column in the file and that the first two columns are student name and an id number. Use dynamic allocation to write a menu-driven grading program that allows all possible options: add student, delete student, change grade, add new exam scores, compute various averages, etc.

4. Write a program that reads and sorts an array of numbers. Use a function to sort the array, but use a pointer to a function to make a comparison of two numbers. If the function returns True, swap the elements; otherwise, the elements are in correct order. Test the program with functions to sort in increasing and in decreasing order.

5. Write a program that reads an array of transliterated strings that represent equivalent strings in some language. The strings are to be sorted according to the alphabet of that language. First order the ASCII characters according to the alphabet of that language. Then use a function that returns True if two characters are ordered in a correct sequence. Use the function to sort the array of strings.

6. Use a structure with two members to represent either a complex number or a rational number. We will call each of them an ordered pair number. Write a generic function to add two ordered pair numbers where the addition is performed by a function a pointer to which is passed as an argument.

7. Repeat 6 to subtract two ordered pair numbers.

8. Repeat 6 to multiply two ordered pair numbers.

9. Repeat 6 to divide two ordered pair numbers.

10. Write a lexical analyzer that finds tokens of the following type in a string:

```
identifier
integer
float
operator
end of string
```

11. Repeat 10 without using an external variable. Use a pointer to a character as an argument to a function `get_token()` which indirectly returns the character read, but unused in a token.

12. Repeat 11, but use a static character variable in `get_token()` instead of indirectly returning the character read but unused. The static character will remain unchanged and may be used for the start of the next token.

Chapter 15

Engineering Programming Examples

In the preceding chapters we have presented the major features of the C language for declaring and accessing data, and controlling program execution flow including both the syntax required by the language and the semantics of the statements. We have also discussed "good" programming style and organization emphasizing the top down design process. In this chapter we make use of these features and techniques to develop several programs for commonly used operations in engineering and scientific computing.

We begin with with operations on matrices, including transforms and sums and products. We next discuss complex numbers together with their representation as a user defined data type and their uses. A program to find solutions to systems of linear algebraic equations is presented next using our complex number functions, followed by another common applications of complex number: the analysis of electrical circuits. We conclude the chapter with a program for numeric integration of arbitrary algebraic functions.

15.1 Matrices

We saw in Chapter 9 that systems of simultaneous linear algebraic equations can be represented and manipulated using two dimensional arrays. For example, a set of n equations in m unknowns:

$$\begin{aligned} a_{0,0} * x_0 + a_{0,1} * x_1 + \cdots + a_{0,m-1} * x_{m-1} &= y_0 \\ a_{1,0} * x_0 + a_{1,1} * x_1 + \cdots + a_{1,m-1} * x_{m-1} &= y_1 \\ &\vdots \\ a_{n-1,0} * x_0 + a_{n-1,1} * x_1 + \cdots + a_{n-1,m-1} * x_{m-1} &= y_{n-1} \end{aligned}$$

Mathematically, such a system can be thought of in terms of a *matrix* equation written in the form:

$$A \times X = Y$$

where A is a matrix, i.e. a two dimensional array of coefficients $a_{i,j}$, X is a vector, i.e. a one dimensional array of elements x_j, and Y is also a vector, y_i. In a matrix representation of algebraic equations, the number of rows corresponds to the number of equations, and the number of columns corresponds to the number of unknowns. (In our case, the values of i range from 0 through $n-1$,

and those of j range from 0 through $m-1$). When the number of rows and columns are equal, the matrix is square, otherwise the matrix is rectangular.

Such a matrix equation may be viewed as a *transformation* of a vector, X, to another vector, Y, by *matrix operator* A. Matrix formalism facilitates combinations of transformations, deriving properties of transformations, as well as finding solutions of equations.

In the next few sections, we will illustrate some useful matrix manipulations and begin to build our own library of utility functions for matrix operations. Many of the functions written can be used in a variety of programs; therefore, we will organize our code in several source files. The file `matutil.c` will contain all the functions we write for matrix manipulations. As usual, the prototypes for these functions are assumed to be in the file `matutil.h`.

In constructing our library, we first implement basic input/output functions for matrices and vectors: the function `readmatrix()` reads the elements of a matrix into a two dimensional array, and the function `printmatrix()` prints the matrix elements. (These functions are similar to the functions `getcoeffs()` and `pr2adbl()` in Chapter 9, except that the right hand side is not included in the matrix array). Vectors are read and printed by functions `readvector()` and `printvector()`. We assume the number of rows and columns are passed as parameters, and that the matrices are arrays of type `double`. The basic I/O functions for matrices and vectors and the requisite header files are straightforward to develop, and are shown in Figure 15.1 These functions are quite simple. The number of rows and columns for the two dimensional arrays are passed as parameters, as are the sizes of the one dimensional arrays. The functions `readmatrix()` and `readvector()` return a cumulative sum of the input values. If desired, these sums may be used by the calling function to detect a matrix or a vector with all zero elements.

15.1.1 Matrix Operations: Transforms

The first operation we will implement is the transformation of a vector X by a matrix A into a vector Y.

$$A \times X = Y$$

In other words, given the values of coefficients and the variables on the left hand side, find the values on the right hand side of the equations. Such linear transformation of a set of values is a common phenomenon in many practical applications such as electronic circuits, mechanical systems, chemical combinations, economic models, interactive relationships, and so forth.

If matrix A has r rows and c columns, the algorithm for the i^{th} equation is:

```
y[i] = a[i][0]*x[0] + a[i][1]*x[1] +...+ a[i][c-1]*x[c-1]
```

This is applied for all the rows from 0 to $r-1$. Translating this algorithm into C code, Figure 15.2 shows the function `mapvector()` that uses A to *map* (i.e. transform) vector X into vector Y.

With these utility functions in had, we can now write a driver program that reads a matrix and then transforms vectors until a zero vector is entered. The code is shown in Figure 15.3. The program declares all array ranges of size `MAX` and uses the function `getrc()` to read the number of rows and columns in the matrix. It then reads and prints the transform matrix of the specified size. Then, the program reads vectors until a zero vector is entered, and for each vector maps it