Chapter 3

Designing Programs Top Down

As program tasks become more complex, it is easier to think about the problem and design the algorithm for the task at hand by breaking the complex task into smaller and simpler *subtasks* and then solve each of the subtasks independently. We do this all the time in everyday life, for example, suppose you need milk for your kid's dinner. A complete algorithm for solving this problem might begin:

```
find the car keys
go to the garage
get in the car
put the key in the ignition
start the car
back the car out of the driveway
...
```

However; when we are worried about feeding the kids, we do not plan our algorithm in such detail. Instead our algorithm might be:

```
drive to the store
buy milk
drive home
```

where each of the steps in this algorithm is a subtask that may involve many steps itself.

We can do the same kind of *modular design* for our programming tasks: begin by thinking at a more abstract level about the major steps to be done, and then for each of these subtasks, design a separate algorithm to solve it. Each program subtask may then be implemented either by a set of statements or by a separate *function*. The advantages of a function are that it hides details of the actual computations from the main body of the code, and it can even be called upon to perform a subtask repeatedly by one or more other functions. In particular, well designed functions can

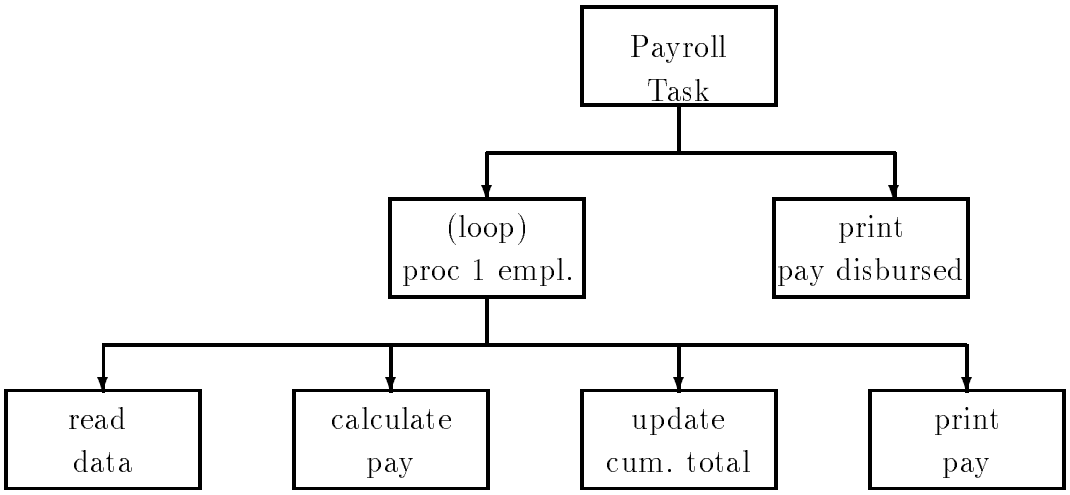

Figure 3.1: Structural Diagram for Payroll Task

be used in a variety of programs. (An example from the above might be driving; it is the same operation in the first and last steps of our algorithm; only the start and destination are different).

In this chapter we will discuss this method of modular design of algorithms and the programs that implement them. We will see how functions may be used in a C program, and how new functions may be defined in the program. As usual, we will look at both the syntax and semantics of this programming construct. Next we will look in more detail at the *macro* facilities provided by the C preprocessor (briefly discussed in Chapter 2) and how these can be used to make programs more readable. The we describe how your programs can interact with the Operating System to perform I/O. Finally we continue our discussion of guidelines for debugging and common errors.

3.1 Designing the Algorithm with Functions

As mentioned above, for complex problems our goal is to divide the task into smaller and simpler tasks during algorithm design. We have seen this technique already in Chapter 1 in our use of a *structural diagram* while developing the algorithm. Figure 3.1 repeats the structural diagram for our payroll task. Here we have divided the payroll task at first into 2 subtasks: processing employees one at a time in a loop, and printing the results. The "processing one employee" subtask is then further divided into four steps: reading data, calculating pay, updating the cumulative total, and printing the pay. In the final implementation of our algorithm, `pay4.c`, we implemented each step using a sequence of statements. The resulting code grew to be rather large, especially for the "calculate pay" step where we had to consider details such as overtime and regular pay. Such details are not important to our understanding of the overall *logic* of the program. However it is to be done, all that we want to do in that step is calculate the pay for one employee as is simply and clearly stated in the algorithm. Calculating pay is an ideal candidate for being implemented as a function.

We will show how to do this shortly, but first it should be pointed out that we have already been using functions to hide the details of tasks in the code we have written. For both the "read data" and "print pay" blocks in the diagram (and the corresponding steps in the algorithm) we have used the built-in library functions, `scanf()` and `printf()`. Many operations are involved in reading the user's typed in data, converting it to its internal representation, and storing it in a variable; however all of this processing is *hidden* by the function `scanf()`. At this point, we do not need to know (and maybe don't care) how it is done, just that it is done correctly.

The important thing here is that top level program logic can use functions without regard to their details. At the next lower level, each function used in the top level program logic can be written in terms of yet lower level functions, and so on. The goal is to arrive at subtasks that are simple to implement with relatively few statements. This approach is called the **top down approach** or **modular programming**. A top down approach is an excellent aid to program development. If the subtasks are simple enough, it also helps produce bug-free reliable programs.

3.1.1 Implementing the Program with Functions

Abstractly, a function can be viewed a a piece of code which, when given sufficient information, performs some subtask and returns the result, a value. Returning to our example, if a function, `calc_pay()`, is used to calculate pay, it will need enough information to perform the computation. In this case the data it needs is the number of hours worked and the rate of pay. As we have stated before, variables, such as `hours_worked` and `rate_of_pay`, defined in a block are only *known*, i.e. can be accessed, within that block. So we cannot give `calc_pay()` direct access to variables defined in other functions, in this case `main()`. However, `calc_pay()` does not need direct access to the variables, it only needs the values to be used for the computation. So we can give a function the values it needs by passing them as **arguments**. We can do this by writing an expression, called a **function call**, giving the name of the function and expressions for the values of the arguments, e.g.:

```
calc_pay(hours_worked, rate_of_pay)
```

The arguments passed are the *values* of `hours_worked` as the first argument, and `rate_of_pay` as the second argument. Given this data we know (or at this point simply believe) that the function does the right thing and returns with a value, the total pay. We say that the function call *evaluates to a value* just as any other expression. The function `calc_pay()` can now be used in `main()` as follows:

```
total_pay = calc_pay(hours_worked, rate_of_pay);
```

In summary, the function `main()` calls `calc_pay()` to perform a task using a set of values. The values are passed as a parenthesized list of data items (which can be any valid expressions) separated by commas. The expressions that appear in such a statement calling the function are called **arguments**. The values of these arguments are received by the called function, `calc_pay()`, which

uses them to perform the desired subtask. Finally, `calc_pay()` returns the value of total pay to the calling function, `main()`, where it is assigned to the variable, `total_pay`.

The value returned by `calc_pay()` will be the total pay calculated using the values of arguments passed to it. Here are a few additional examples of function calls used in an assignment expression:

```
total_pay = calc_pay(30.0, 10.0);  /* calc_pay() returns 300.0, */
                                   /* which is stored in total_pay. */
total_pay = calc_pay(20.0, 10.0);  /* total_pay is assigned 200.0. */
```

A function call is an expression and has a value. Just as we had to declare the data types of variables to the compiler, we must also declare the data type of a function. This declaration also includes the number of arguments the function requires and their types. For example, here is a declaration for `calc_pay()`:

```
float calc_pay(float hours, float rate);
```

The declaration states that `calc_pay()` is a function because the identifier `calc_pay` is followed by a parenthesized list of arguments, that it requires two `float` arguments, and that it is of `float` type, i.e. it returns a `float` value. This declaration statement for a function (notice it is terminated by a semi-colon) is called a **prototype statement** because it gives the *prototype* (or the form) for calls to the function. In general, we will refer to the list of data expected to be passed to a function as specified in the prototype statement as a **parameter list** and an individual data item in this list as a **parameter**. (Sometimes, however, the terms parameter and argument are used interchangeably). The names of the parameters in a prototype statment are optional; but including well chosen names for parameters can make the declaration more meaningful. These parameter names are dummy names which have no relation to the names of arguments in a function call or parameters in the function definition (described in the next section).

Let us implement the top level program logic using the function `calc_pay()` to calculate pay. The code is shown in Figure 3.2 and for simplicity, we have not included calculation of `gross` and `average_pay`.

Figure 3.3 shows the behavior of the function call pictorially. The box labeled **main()** represents the function `main()` in our program and contains memory cells for variables declared in `main()` labeled with their names (e.g. `hours_worked`). The box labeled **calc_pay()** represents the function `calc_pay()`. At this point we do not know anything about the internals of this box such as what variables are declared, and what statements will be executed; but at this point we do not need to know this information. The box shows all of the information we need to know; namely that the function expects two `float` type arguments to be passed and will return a `float` type result. The dashed lines in the figure show that, for the call we have written in `main()`:

```
total_pay = calc_pay(hours_worked, rate_of_pay);
```

```
/*   File: pay5.c
     Programmer: Programmer Name
     Date: Current Date
     The program gets payroll data, calculates pay, and prints out
     the results for a number of people. A separate function is used
     to calculate total pay.
*/
#define   REG_LIMIT      40.0
#define   OT_FACTOR      1.5

main()
{
     /* declarations */
     int id_number;
     float hours_worked, rate_of_pay, total_pay;
     float calc_pay(float hours, float rate);

     /* print title */
     printf("***Pay Calculation***\n");

     /* initialize loop variables */
     printf("\nType ID Number, zero to quit: ");
     scanf("%d", &id_number);

     while (id_number > 0) {
          /* read data into variables */
          printf("Hours Worked: ");
          scanf("%f", &hours_worked);
          printf("Hourly Rate: ");
          scanf("%f", &rate_of_pay);

          /* calculate pay */
          total_pay = calc_pay(hours_worked, rate_of_pay);

          /* print data and results */
          printf("\nID Number = %d\n", id_number);
          printf("Hours Worked = %f, Rate of Pay = $%6.2f\n",
                    hours_worked, rate_of_pay);
          printf("Total Pay = $%10.2f\n", total_pay);

          /* update loop variables */
          printf("\nType ID Number, zero to quit: ");
          scanf("%d", &id_number);
     }
}
```

Figure 3.2: Code for pay5.c driver

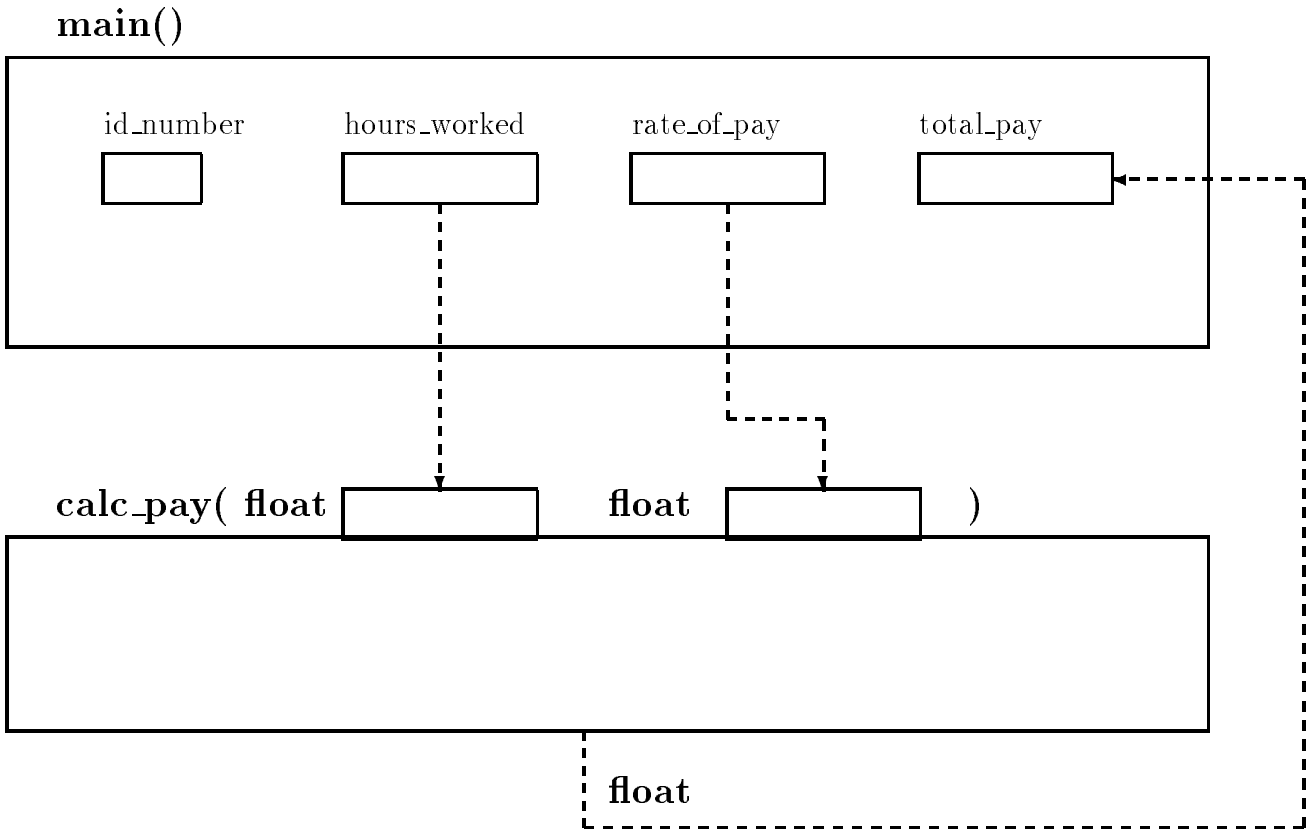

Figure 3.3: Function Call to `calc_pay()`

the first argument, the value of `hours_worked`, is passed to the first parameter of `calc_pay()`, and the second argument, the value of `rate_of_pay`, is passed to the second parameter. The return value from `calc_pay()` is placed in the variable `total_pay` by `main()`.

In summary, the function `main()` represents the overall logic of the program. The details of how pay is actually computed does not change the overall logic. Of course, the program in Figure 3.2 is not yet complete since we have not written the function `calc_pay()`. If an attempt is made to compile the program at this point, there will be a linker error message stating that the function `calc_pay()` cannot be found. Only when the function is written is the program complete and may be compiled and executed.

3.2 Defining Functions

A function is defined by writing the source code for it. Just as for `main()`, defining the function consists of giving a *function header* and a *function body*. The code for `calc_pay()` is shown in Figure 3.4. (It is included in the same source file as the code in Figure 3.2). Let us look at the function header first.

```
/* File: pay5.c - continued */
/* Function calculates and returns total pay */
float calc_pay(float hours, float rate)
{    float regular, overtime, total;

printf("\ndebug:entering calc_pay(): hours = %f, rate = %f\n",
               hours, rate);

     if (hours > REG_LIMIT) {
          regular = REG_LIMIT * rate;
          overtime = OT_FACTOR * rate * (hours - REG_LIMIT);
     }
     else {
          regular = hours * rate;
          overtime = 0;
     }
     total = regular + overtime;
printf("debug:returning from calc_pay(): %f\n", total);
     return total;
}
```

Figure 3.4: Code for `calc_pay()`

```
     float calc_pay(float hours, float rate)
```

The header specifies that the name of the function is `calc_pay`, and that the function returns a `float` value. It also lists the parameters and their types, in this case there are two formal parameters, `hours` and `rate`, each of type `float`. Notice that the function header is very similar to the prototype statement for the function, with two notable exceptions. First, there is no semicolon at the end, indicating that this is the definition of the function, not a declaration. Second, in the function header, the variable names in the parameter list are required, and this list is sometimes called the **formal parameter list**. These formal parameters act as variable declarations for the function with the additional feature that they receive initial values from the arguments when the function is called; the first parameter gets the value of the first argument, the second parameter the value of second argument, and so on. The formal parameters in a function definition behave in the same manner as automatic variables, and their scope is limited to the function itself. The names in this list are the names used within the function body to access these values.

The body of the function is defined, as with `main()`, within brackets, { and } and consists of the variable declarations for the block followed by the executable statements to perform the subtask of the function. In our case, we declare variables `regular`, `overtime`, and `total` which are called **local variables** because their scope is local, i.e. limited to within the function. We then calculate regular pay, overtime pay and total pay as before, but we use the formal parameter names and the names of the local variables in our computations. Finally, since a function can

return only one value, we return only the value of total pay:

```
return total;
```

The above `return` statement returns the *value* of the variable, `total`, to the calling function. In general, a `return` statement can be used to return the value of any expression. When the `return` statement is executed, the program control returns immediately to the calling function where the function call *evaluates* to the returned value.

When a function is first written, it is a good practice to include debug statements in the function definition showing the name of the function entered, the values of the parameters received, and the value returned by the function. When the program is run, these debug statements will produce a trace of all function calls and returns and as such are invaluable for debugging, particularly when a program uses many functions. We have included `printf()` statements for this purpose in the code for `calc_pay()` shown in the figure.

The above function, together with `main()` in the file pay5.c, forms a complete program which may be compiled and executed. A sample session shown below is similar to the one for `pay4.c`. The only change is that `calc_pay()` calculates and returns total pay, whereas in `pay4.c` total pay was calculated in `main()`.

```
***Pay Calculation***

Type ID Number, zero to quit:  123
Hours Worked:  20
Hourly Rate:  7.5

debug:entering calc_pay():  hours = 20.000000, rate = 7.500000
debug:returning from calc_pay():  150.000000

ID Number = 123
Hours Worked = 20.000000, Rate of Pay = $ 7.50
Pay = $ 150.00

Type ID Number, zero to quit:  0
```

The debug printing clearly shows argument values at entry to `calc_pay()` and the returned value. If there are any bugs in a function, such debug printing helps detect and remove them.

3.2.1 Passing Data to and from Functions

As we can see from the above description, and also in Figure 3.5, information is passed to a function as arguments specified in the calling expression. This information is received by the function in the cells reserved for the formal parameters. In our case, the values of `hours_worked`

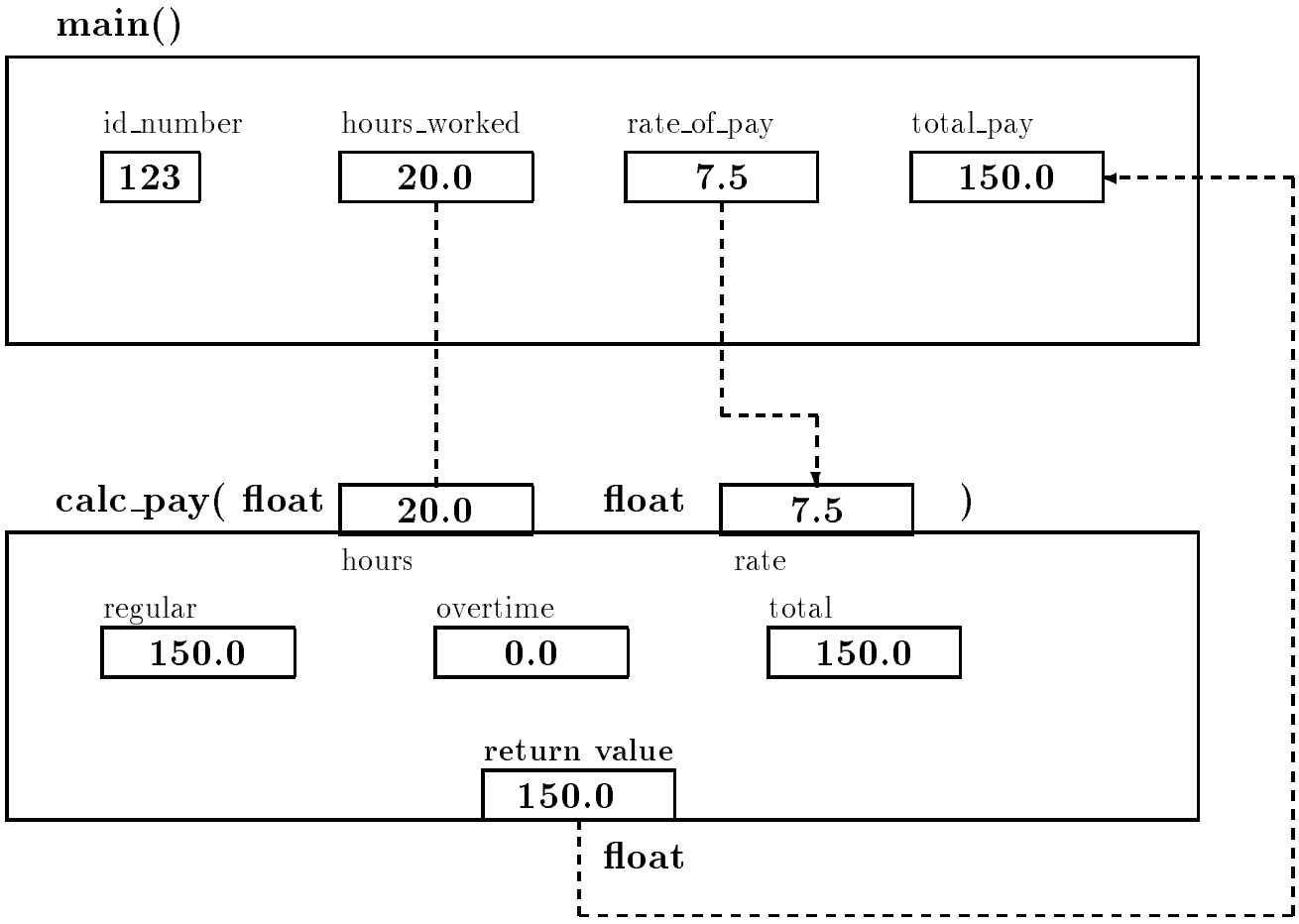

Figure 3.5: Function Call Trace

and `rate_of_pay` (the arguments of the call) are copied to the cells called `hours` and `rate` within the function `calc_pay()`. Remember, these names are only known internally to the function. All that `main()` sees of the function is a *black box* as was shown in Figure 3.3.

The names of the formal parameters are arbitrary. For example, `calc_pay()` may be defined with any names for formal arguments:

```
float calc_pay(float x, float y)
{
     if (x > REG_LIMIT) ...
}
```

or,

```
float calc_pay(float hours_worked, float rate_of_pay)
{
```

```
        if (hours_worked > REG_LIMIT) ...
}
```

As long as the function uses the formal parameters names internally for computations, the function definitions behave the same. In the last case, even though the formal parameters have the same names as variables defined in `main()`, they represent distinctly different variables, as shown in Figure 3.5. In summary, the scope of automatic variables defined in a block is local to that block, i.e. the objects can be directly accessed by name only within that block and in blocks nested within it.

As we stated earlier, the arguments in a function call can be any valid expressions. Only the values of the argument expressions are passed to the called function. For example, these are valid function calls:

```
printf("Pay = %f\n", hours_worked * rate_of_pay);
printf("Pay = %f\n", calc_pay(hours_worked, rate_of_pay));
calc_pay(hours_worked, rate_of_pay * 1.10);
```

The argument in the first `printf()` call is a product expression. The result of evaluating that expression is passed to `printf()`. The second statement uses an argument that is itself a function call. The function call evaluates to a value which is then passed to `printf()`. The second argument in the last statement is an expression whose value is passed to `calc_pay()`.

Information is returned from a function using the `return` statement which can also return the value of any valid expression. The syntax of the `return` statement is:

```
return <expression>;
```

For example, we could have combined the last two statements in the function definition of `calc_pay()`:

```
return regular + overtime;
```

where `calc_pay()` would then return the value of the expression `regular + overtime`.

When writing functions, tools such as shown in Figure 3.5 can be very useful in tracing the behavior of the function. Another way to check a function for bugs is to manually trace its execution with representative values for the formal parameters. Figure 3.6 shows such a trace for `calc_pay()`. Note: the variables `hours` and `rate` (the formal parameters) receive values during the function calls. Other local variables get values as the function is executed.

In our payroll program, the overall logic can be made even more apparent if functions are used to get the input data and to print the results. The driver, i.e. `main()`, can then follow the overall logic and use function call statements to get the data, calculate the pay, and print the results. A function that prints data is simple to write. Writing a function that reads data is somewhat more involved. We will delay writing such functions until Chapter 6.

```
                                        hours  rate   regular overtime  total
float calc_pay(float hours, float rate)  20.0   7.5     ??       ??       ??
{    float regular, overtime, total;

printf("debug:entering calc_pay(): hours = %f, rate = %f\n",
               hours, rate);
     if (hours > REG_LIMIT) {
          regular = REG_LIMIT * rate;
          overtime = OT_FACTOR * rate *
                      (hours - REG_LIMIT);
     }
     else {
          regular = hours * rate;          20.0  7.5     150.0    ??       ??
          overtime = 0;                    20.0  7.5     150.0   0.0       ??
     }
     total = regular + overtime;           20.0  7.5     150.0   0.0     150.0
printf("debug:returning from calc_pay(): %f\n", total);
     return total;
}
```

Figure 3.6: Trace for `calc_pay()`

3.2.2 Call by Value and Local Variables

This section reviews and formalizes several features of variables that we have already encountered. We know that direct access of objects is performed by using variable names in expressions. The use of a variable on the left side of an assignment operator stores a new value in that object; the use of a variable anywhere else retrieves the value of the object. Objects defined in one function are not directly accessible to other functions. A calling function passes values of arguments to a called function. Only the values of these arguments, and NOT the arguments themselves, are available to the called function. The values of the arguments are stored in the parameters, and only the called function has access to these parameters. When called functions have access only to argument values, and not to arguments themselves, the function calls are termed **call by value**. In C, all function calls are call by value. It is impossible for a called function to have direct access to an object defined in the calling function. Let us examine the implications. Consider a program that uses a function to increment the value of an argument.

```
/*   File: incr.c                                  Program Trace
     Program demonstrates call by value.                x
*/
#include <stdio.h>
main()
{    int x;                                             ??
     int incr(int n);
```

```
    printf("***Call by Value***\n");
    x = 7;                                          7
    printf("Original value of x is %d\n", x);       7
    printf("Value of incr(x) is %d\n", incr(x)); 7

    printf("The value of x is %d\n", x);            7
}


/* Function increments n */                         n
int incr(int n)                                     7
{
    n = n + 1;                                      8
    return n;                                       8
}
```

Compiling and executing this programs gives the following sample session:

```
***Call by Value***
Original value of x is 7
Value of incr(x) is 8
The value of x is 7
```

The program trace shows that `x` in `main()` is assigned a value of 7 prior to a function call to `incr()` which increments its parameter to 8 and returns that value. After the function call, the value of `x` in `main()` is still 7, unchanged because only the *value* of `x` is passed to `incr()`. It was the cell, `n`, in `incr()` that was incremented as seen in Figure 3.7

We see that a called function cannot directly change the value of an object defined in the calling function. This is true even if the formal parameter in `incr()` were called `x`. Formal parameters represent new and distinct objects unrelated to any other objects defined elsewhere.

The variables declared at the beginning of a block (e.g. a function body) have all been of a storage class called **automatic**. This means that these variables are automatically created and destroyed each time the function is executed. When the execution of a function begins, the variables declared at the beginning of the function block as well as the formal parameters are created, i.e. memory cells for these variable names are allocated. When the execution of a function is completed (e.g. when a `return` statement is executed), the memory allocated for these variables is freed, i.e. these variables and their values no longer exist.

Automatic variables can be defined at the beginning of any block within the primary function block and exist only in the block in which they are defined. Memory for automatic variables declared in a block is allocated when the block is entered, and freed when the block is exited.

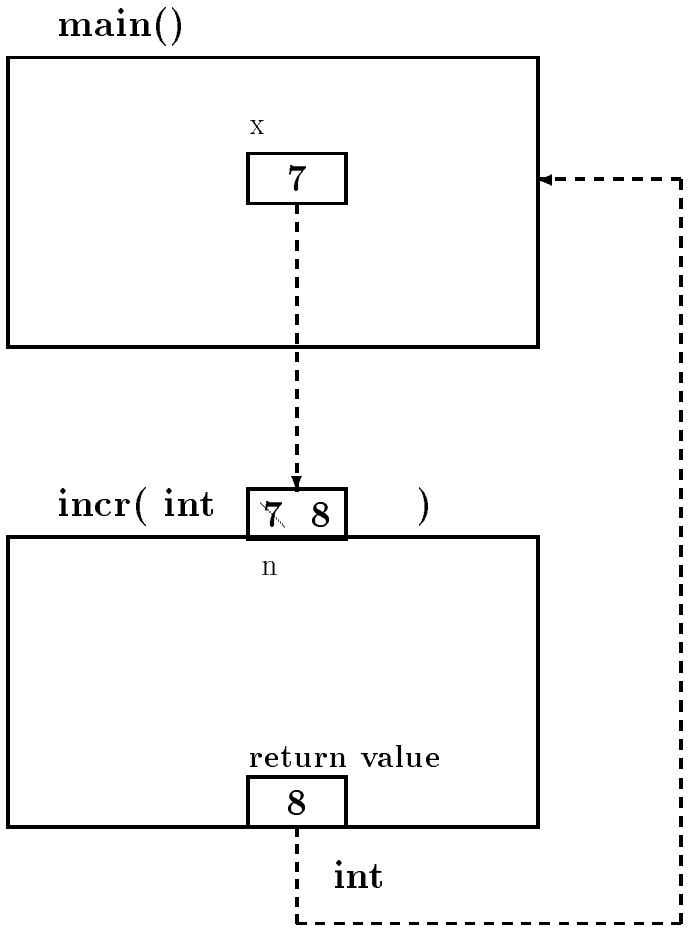

Figure 3.7: Call by value variable allocation

The **scope** of a variable is that part of the program where the variable is visible, i.e. where the variable can be accessed directly by name. The scope of automatic variables is local to the block in which they are defined as well as any blocks nested within it. Automatic variables are frequently referred to as **local variables**, since their scope is *local*.

A variable of automatic storage class can be explicitly defined in a declaration by preceding it with the keyword `auto`. Thus, the following declarations declare automatic variables:

```
auto int x, y;
auto float r;
```

If no storage class is specified in a declaration, automatic storage class is assumed by default. In all of our programs, so far, declarations have been for automatic variables by default. In general, most variables used in programs are automatic, and the default declaration without the keyword `auto` is a standard practice. Other storage classes will be discussed in Chapter 14. Until then, we will use only automatic variables.

As we stated before, a declaration only allocates a memory cell and associates the name with the cell; the value in that cell is, in general, unknown. However, it is possible to specify initial values of automatic variables in the declaration statements. Examples include:

```
int x = 5 * 2;
int y = isquare(2 * x);
float z = 2.8;
```

The first declaration initializes `x` to 10, and the second initializes `y` to the value returned by the function call `isquare(2 * x)`. If the function `isquare()` returns the square of its argument, then `y` in this case, is initialized to 400, i.e. the square of 2 * x. Finally, the last declaration initializes the variable `z` to the value 2.8.

The syntax for a declaration statement with initialization is:

<type_specifier><var_name> [=<init_expr>] [, <var_name> [=<init_expr>]...];

The declaration allocates memory for each <var_name> of a type indicated by <type_specifier>, and initializes the variable to the value of the initializer expression, <init-expr>. The initializer expression can be any C expression including function calls.

Consider the following example in which automatic variables are declared in nested blocks:

```
/*   File: auto.c
     Program shows declarations of automatic variables in nested
     blocks. Scope of automatic variables is the block in which they
     are defined.
*/
main()
{                                   /* outer block */
     auto int x = 10, z = 15;       /* x and z are allocated and initialized */

     printf("***Automatic Variables and Scope***\n\n");
     {                              /* inner block */
          int x = 20, y = 30;       /* new variables x and y are allocated */
                                    /* only the new x can be accessed */
          printf("In the inner block: \n");
          printf("x = %d, y = %d, z = %d\n",
                    x, y, z);       /* new x and y, and z are printed */
     }                              /* new x and y are freed */
     printf("In the outer block:\n");
     printf("x = %d, z = %d\n", x, z); /* only the old x can be */
                                       /* accessed in the outer block.*/
     /* printf("y = %d\n", y);          error: y is not visible here. */
}
```

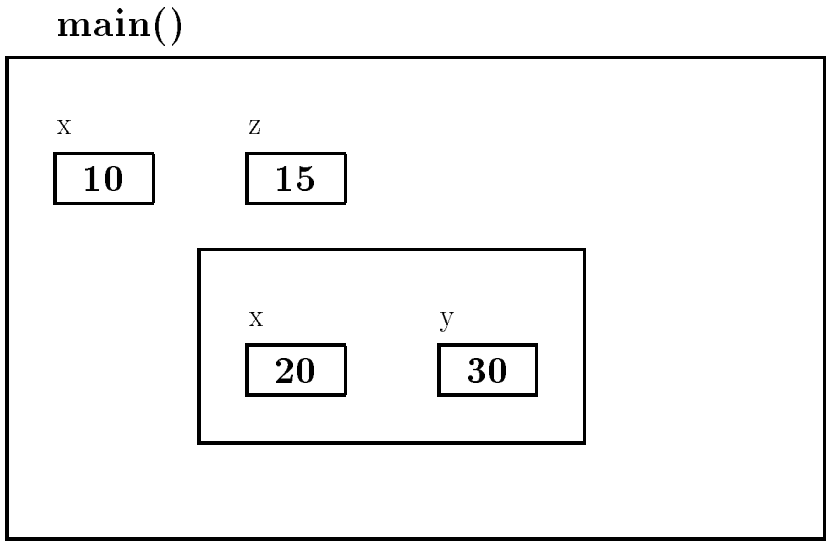

Figure 3.8: Local Variables in Blocks

The program contains an outer block, which is the function body for `main()`, and an inner block. The scope rules say that an inner block can access variables declared within it plus any variables declared in an enclosing block. However, if the same variable name is used in an inner and an outer block, the local variable in the inner block is accessed. The outer block cannot access variables defined in an inner block.

In the example, variables `x` and `z` are declared in the outer block and assigned values. The outer block can access only these variables. Variables `x` and `y` are declared in the inner block and assigned values. The inner block can access the variables `z`, `y`, and that `x` which is defined in the inner block. As shown in a comment, if the outer block tried to access `y`, a compile time error would occur. This behavior can be seen in Figure 3.8. The allocation of storage is shown when the program is executing within the inner block as can be seen by the nested box containing `x` and `y`. When this block is completed, the inner box, and all variables inside, is freed. A sample output of the program shows the results:

```
***Automatic Variables and Scope***

In the inner block:
x = 20, y = 30, z = 15
In the outer block:
x = 10, z = 15
```

It is also possible to qualify an automatic variable as a constant using the keyword `const`. A `const` qualifier allows initialization of a variable but the variable may not be otherwise changed within the program. Here is an example:

```
const int x = 100;
```

In the above case, `x` is initialized to 100 and qualified as a constant. Its value may not be changed elsewhere in the program, e.g. in an assignment statement. Constant qualifiers are used to ensure that certain variable values are not altered by oversight.

Let us consider a somewhat more meaningful example that declares a variable in an inner block. The task is to swap values of two objects, `x` and `y`. We need a temporary variable to save one of the values; otherwise, assigning the value of `y` to `x` would overwrite the original value of `x`. We can declare the temporary value in an inner block.

```
/*   File: swap.c
     This program swaps values of two objects. It defines and uses a
     temporary variable in an inner block.
*/
#include <stdio.h>
main()
{    int x = 10, y = 20;

     printf("***Swap Values***\n\n");
     printf("Original values: x = %d, y = %d\n", x, y);
     {    int temp;

          temp = x;
          x = y;
          y = temp;
     }
     printf("Swapped Values: x = %d, y = %d\n", x, y);
}
```

Here is the output of the program:

```
     ***Swap Values***

     Original values:  x = 10, y = 20
     Swapped Values:  x = 20, y = 10
```

Defining variables in blocks other than a primary function block is not recommended unless there are good reasons for it. In the above example, a temporary variable is declared closest to its use and has no logical role in the rest of the program. When a function uses many variables, declaring variables closest to their use may make it easier to understand the program behavior. For the most part, we will declare all variables at the beginning of primary function blocks.

The formal parameters of a function are also variables that are automatically allocated during a function call, and into which the argument values are passed. Their values, just like those of any other variables, may be changed in the function. The scope of the formal parameters is the body of the function, i.e. the scope is local to the function body.

3.3 Coding Programs for Readability

In the previous sections we have seen how to organize programs modularly, beginning with the algorithm, and carrying that organization into the code using functions. This is a form of *information hiding*, i.e. the details of performing a particular operation are hidden from the more abstract steps of the algorithm. Here we are hiding *ideas* or *abstractions* at the algorithm level. Another form of information hiding at the source code level is described in this section; namely hiding the details of the *syntax* of the language in order to make the source code more readable.

3.3.1 The C Preprocessor

We have already seen that in order for a program to be run, it must be compiled, i.e. translated from the C language to the machine language of the computer being used. This compilation process takes place in several steps; the source code is read from the file, checked for proper syntax, and analyzed for the meaning of the statements in the code. The proper machine language steps to perform these statements can then be generated (and optimized) and then linked with other functions to produce the executable file. At the beginning of this entire process, standard C compilers provide an additional step called the **preprocessor**. The source code is read from the file and given to the preprocessor where it is translated into a modified source code file which is then given to the compiler proper for translation to machine language. The transformations performed by the preprocessor are directed by lines in the original source file called **compiler directives**. All such lines begin with the `#` character as the first non-white space character on the line and are of one of three types of directives: macro definitions, file inclusion, and conditional compilation. Each of these are discussed in the following sections.

3.3.2 Macros

In Chapter 2 we introduced the `define` compiler directive which defines symbolic names for strings of characters. Such a string of characters can be arbitrary, for example a sequence of characters representing a numeric constant. These names can then be used anywhere in the program instead of the string itself. The C preprocessor replaces these symbolic names with the specified strings prior to compiling the program. We have seen examples where using names for arbitrary strings makes it easy to change all occurrences of these names by merely changing the definitions. It also makes for easier reading and debugging of programs by allowing the programmer to use a name which has some meaning rather than some "magic number".

The definition is called a **macro** and the preprocessor performs a **macro expansion** when it substitutes the string for the name. A macro definition takes the form:

#define <symbol_name> <substitution_string>

The macro names follow the same rules as identifiers, however, a common convention observed

by most C programmers is to name macros in all upper case to distinguish them from program variables. No quotation marks are used to delimit the string, nor is the directive terminated by a semi-colon. Instead, the string extends to the end of the line (an escape character, \, can be used to continue the string on the next line). For example, the following are macro definitions:

```
#define     PI          3.14159
#define     SIZE        1000
#define     RSQUARED    radius * radius
#define     AREA        PI * RSQUARED
#define     LONG        This is a very long macro \
definition we continued to the next line
```

When directives such as these appear in the source file, then the macros are said to have been defined. We have defined macros for the symbols `PI`, `SIZE`, `RSQUARED`, `AREA` and `LONG`. With the above definitions, the defined names may be used anywhere in program statements. The preprocesser generates the expanded source code by string replacement, for example:

```
        Original code           Expanded code after preprocessing


circum = 2 * PI * radius;     circum = 2 * 3.14159 * radius;
y = x + SIZE;                 y = x + 1000;
printf("SIZE = ",SIZE);       printf("SIZE = ",1000);
AREA;                         3.14159 * radius * radius;
```

As can be seen, the preprocessor replaces the macro name with the specified replacement string in the entire source file following the definition. The substitution is not made if a macro name, occurs in double quotes as in the format string in the `printf()` statement shown above.

The scope of the macro definition is the entire source file following the definition line. The definitions may be removed at any point in the program by a directive `#undef`, for example:

```
#undef SIZE
```

The above directive makes the preprocessor "forget" the previous definition for SIZE. If desired, a new definition may be specified for SIZE at this point. It is a common practice to put macro definitions at the top of the source file, unless the old definitions are removed at some point in the source file and new definitions are specified:

```
#define SIZE 40          /* SIZE is define to be the string 40 */
...
#undef SIZE              /* SIZE is undefine */
#define SIZE 100         /* SIZE is defined to be 100 */
```

Identical definitions for identifiers may appear in a file without causing any problems; however, two different definitions for an identifier represent an error.

```
#define SIZE 40
#define SIZE 40     /* OK */
#define SIZE 100    /* ERROR */
```

The only way to make a new definition for an identifier is to first undefine it, i.e. remove its first definition.

Macros with Arguments

Macro definitions may also have formal parameters which are replaced by the actual arguments given in the macro call. This is similar to parameters in function calls; however, macro arguments are treated as *strings of characters* and are substituted for parameters by the preprocessor; no evaluation takes place. Consider the example:

```
#define READ_FLT(fvar)   scanf("%f", &fvar)
```

The macro encapsulates the expression for reading a `float` number, i.e. a macro call is replaced by a string that represents a correct `scanf()` function call to read a `float` number into an object passed to the macro. The actual argument in a macro call replaces `fvar` in the replacement string. In other words, every time the macro is called, the expanded code is substituted literally except that `fvar` in the definition is replaced by the argument given in the actual call. Here are some examples of macro calls with parameters together with the expanded code:

```
        macro call          Expanded Code


READ_FLT(x);          scanf("%f", &x);
READ_FLT(rate);       scanf("%f", &rate);
```

Macro calls in these cases expand to C statements. Such calls are said to expand to `in-line code`, because the resulting code represents statements in the source code. These types of macro calls can be used in place of function calls, for example, instead of writing a function to square a number, we can define a macro:

```
#define SQ(x)       (x * x)
```

We can use such a macro in any expression, e.g.,

```
y = SQ(radius);
printf("Square of %d is %d\n", radius, SQ(radius));
```

However, remember, macro calls are *substitutions*, and macro parameters are neither evaluated nor checked for data type consistency. Therefore, proper placement of parentheses is important in macro definitions. For example, consider the following macro call and expanded code:

```
SQ(x+y)
```

expanded becomes

```
(x + y * x + y)
```

The expanded code is not the square of `(x + y)`, as we would expect. By precedence rules, it is a sum of three terms, `x`, `y * x`, and `y`. A proper definition of a macro for square should be:

```
#define SQ(x)  ((x) * (x))
```

With this definition,

```
SQ(x+y)
```

will expand correctly to

```
((x + y) * (x + y))
```

Here is a simple example program:

```
/* File: macro.c */
#define READ_FLT(fvar)   scanf("%f", &fvar)
#define PI     3.14159
#define SQ(x)  ((x) * (x))

main()
{
     float radius;

     printf("Type Radius: ");
     READ_FLT(radius);
     printf("Area of a circle with radius %6.2f is %6.2f\n",
               radius, PI * SQ(radius));
}
```

The output of a sample run is:

```
Type Radius:  10
Area of a circle with radius 10.00 is 314.15
```

Why use macros with arguments when functions will serve the same purpose? The advantage is practical, NOT logical. When a function is called, there is a certain amount of run time overhead, i.e. extra time needed during execution. The overhead comes from passing arguments, transferring control, returning a value, and returning control. If a function is called just a few times, the overhead is negligible. However, if a function is used numerous times, e.g. in a loop executed many times, then the overhead can become significant.

A macro on the other hand has no run time overhead. It is expanded at compile time into in-line code which has no overhead at run time. If execution time for a program is a problem because of a frequently used routine, then writing a macro for that routine makes good sense, as long as the operation can be simply expressed as a macro.

An Example Program

Let us look at another example program to make use of these new facilities.

Task

Read a set of high temperature readings for some number of days and to count the number of nice days, bad days, and the average temperature for the period. Nice days are those days whose temperature falls within some "comfort zone".

The high level algorithm for this task is straight forward;

```
prompt the user and read first temperature
while there are more days to read
      process one day's temperature
      accumulate total temperature
      read the next temperature
print results
```

With this algorithm, we next consider what information we will be working with in this program. We read daily temperatures, so we will need a variable for that, and variables to count the number of nice and bad days. Since we compute the average temperature, we accumulate the total of all the daily temperatures, so we need a variable for that. Next we consider how we will implement the algorithm using functions to hide details. For example, the step to print results, printing the number of nice and bad days as well as computing and printing the average temperature can be done in a function, `print_results()`, which is given the number of nice days, bad days, and the cumulative total of temperatures. The step of processing one day's temperature is another candidate; however, this step involves updating our counts of nice and bad days. Since, as we

have seen, functions cannot access variables local to main, we refine our algorithm to fill in some of the details of this step:

```
prompt the user and read first temperature
while there are more days to read
      if it's a nice day, count a nice day
      otherwise count a bad day
      accumulate total temperature
      read the next temperature
print results
```

We can use a function to test if a day is nice, thus hiding the details of this operation. We are now ready to write the code for `main()` as shown in Figure 3.9. It should be noted we have made an additional design decision here; we use a zero value for the temperature read in as the loop termination. Also not that we have provided prototype statements for our functions, `nice_day()` and `print_results()`. This is sufficient information about these functions when considering the logic of `main()`. (We have specified the return value of print_results as type `int`, but the function has no real meaningful return value).

We next turn out attention to the function, `nice_day()`. This function is given the temperature and should return True if this qualifies as a nice day, and False otherwise. The task specified that the temperature of a nice day is to fall within some "comfort zone", i.e. not too cold and not too hot. We can write the algorithm for this function from this information:

```
if temperature is too cold, return False
if temperature is too hot, also return False
otherwise, this is a nice day, return true
```

We choose to implement the too cold and too hot tests using macro:

```
#define   TOO_COLD      80
#define   TOO_HOT       90

#define   HOT_DAY(t)    ((t) > TOO_HOT)
#define   COLD_DAY(t)   ((t) < TOO_COLD)
```

Coding of the function is straight forward. Similarly, for `print_results()`, the algorithm is:

```
print number of nice days and bad days
if there are any days counted
      compute the average temperature
      print the average temperature
```

```
/*   File: niceday.c
     Programmer: Programmer Name
     Date: Current Date
     This program counts the number of nice days in a set of high
     temperature data.
*/

int nice_day(int temp);
int print_results(int nice, int bad, int temp_sum);

main()
{  /*  declarations  */
   int temperature,         /*  daily temperature  */
       total = 0,           /*  cumulative total   */
       num_nice_days = 0,
       num_bad_days  = 0;

   /*  print title and prompt  */
   printf("***Count Nice Days***\n\n");
   printf("Type daily high temperature readings (0 to quit): ");

   /*  read the first temperature  */
   scanf("%d", &temperature);
   while (temperature != 0)  {

         /*   process one temperature  */
         if ( nice_day(temperature))
              num_nice_days = num_nice_days + 1;
         else
              num_bad_days = num_bad_days + 1;
         /*  accumulate  total of temperatures  */
         total = total + temperature;

         /*   read next temperature  */
         scanf("%d", &temperature);
   }

   print_results(num_nice_days, num_bad_days, total);

}
```

Figure 3.9: Driver for `niceday.c`

```
/*   File: niceday.c (continued) */

#define   TRUE           1
#define   FALSE          0

#define   TOO_COLD      80
#define   TOO_HOT       90

#define   HOT_DAY(t)    ((t) > TOO_HOT)
#define   COLD_DAY(t)   ((t) < TOO_COLD)

#define   ANY_DAYS(n,b) (((n) + (b)) > 0)

/*  Function to test for a nice day given the temperature        */
int nice_day(int temp)
{
   if( COLD_DAY(temp))  return FALSE;

   if( HOT_DAY(temp))   return FALSE;

   return TRUE;

}

/*  Function to print results given number of nice and bad days  */
/*    and total of temperatures                                  */
int print_results( int nice_days, int bad_days, int total)
{
   float average_temp;

   printf("There were %d nice days and %d bad days\n",
                        nice_days, bad_days);

   if ( ANY_DAYS( nice_days, bad_days))  {
      average_temp = (float) total / (float) (nice_days + bad_days);
      printf("The average temperature for %d days was %f\n",
                        nice_days +  bad_days, average_temp);
   }
}
```

Figure 3.10: Functions for `niceday.c`

The resulting code for these functions is shown in Figure 3.10

Compiling and executing this program with some sample data produces the following sample session:

```
***Count Nice Days***

Type daily high temperature readings (0 to quit):  83
85
88
92
94
86
82
80
79
0
There were 6 nice days and 3 bad days
The average temperature for 9 days was 85.444443
```

3.3.3 Including Header Files

The second feature provided by the preprocessor allows us to break our source files into smaller pieces to be reassembled at compile time. Using functions to hide details of algorithms and macros to hide the syntax and "magic numbers" to make our programs more readable often results in many function prototype statements and macro definitions at the beginning of source code files. These may also be hidden in separate files, and included in the source file by the preprocessor. The files containing this information to be included are called **include files** or **header files**, and by convention, are named with a `.h` extension on the file name. Header files are also often used to provide common macro definitions and prototype statements that may be useful in may programs (or as we shall see later, in many files making up a single program). An example of the later case are the standard library functions provided in C; the prototype statements for these functions should be available to any program which chooses to use the functions. In many of our programs so far, we have used the library functions `printf()` and `scanf()`. Where are the prototypes for these? As well as providing the code for library functions, all standard C implementations provide a set of `.h` files with this information. The file `stdio.h` contains the prototypes and macros needed to use the I/O library. (We have not needed this file before because the compiler will make assumptions about functions if prototypes are not provided. Sometimes these assumptions are "safe", but often they are not. It is a good idea, from now on, to include `stdio.h` in any program using the I/O library).

The statements and directives in an include file are inserted in a source file when the preprocessor encounters an `#include` directive in the original source file. To include `stdio.h` the directive is:

```
#include <stdio.h>
```

The angle brackets, < and >, surrounding the filename indicate to the preprocessor that the file, `stdio.h`, is to be found in "the usual place" where standard header files are kept on the system (this is system dependent), and its contents placed in the source code in place of the `#include` directive. Any other directives within the included file (such as `#define` or other `#include` directives) are also processed at this time.

Besides the standard header files, as a programmer you can create and include your own header files for your programs. For example, in our `niceday.c` program, we defined macros for `TRUE` and `FALSE`. These macros are very common in many programs, so it would be convenient if we could enter those definitions in a single header file and simply include that header file in any program the uses those macros. This header file might be called `tfdef.h` and contain:

```
/*   File: tfdef.h
     Programmer: Programmer Name
     This file contains the definitions of TRUE and FALSE
*/


#define   TRUE           1
#define   FALSE          0
```

To include these definitions in a `.c` source file, use the directive:

```
#include "tfdef.h"
```

Notice in this instance that the file name is surrounded by double quote, ", characters rather than the angle brackets used before. This syntax tells the preprocessor that the header file is to be found in the same directory as the `.c` source file currently being processed.

Again, in our nice day program, all of the other macro definitions and prototypes relating just to this program may also be placed in a header file, say `niceday.h`:

```
/*   File: niceday.h
     Programmer: Programmer Name
     This file contains the definitions of macros and prototypes
     for functions used by the niceday program.
*/


#define   TOO_COLD      80
#define   TOO_HOT       90
```

```
#define   HOT_DAY(t)    ((t) > TOO_HOT)
#define   COLD_DAY(t)   ((t) < TOO_COLD)

#define   ANY_DAYS(n,b) (((n) + (b)) > 0)

int nice_day(int temp);
int print_results(int nice, int bad, int temp_sum);
```

and replaced in `niceday.c` with:

```
#include "niceday.h"
```

Thus, the beginning of `niceday.c` has been reduced to:

```
/*   File: niceday.c
     Programmer: Programmer Name
     Date: Current Date
     This program counts the number of nice days in a set of high
     temperature data.
*/

#include <stdio.h>
#include "tfdef.h"
#include "niceday.h"

main()
{  ...
```

Notice we include `stdio.h` at the head of the source file. Its contents are available for use by the entire source file. We also declare the function prototypes for `nice_day()` and `print_results()` in the file `niceday.h` outside `main()`. A declaration outside a function is called an external declaration. The scope of an external declaration is the entire file from the point of the declaration; i.e. all code that follows the external declaration can use the declared item. Since `stdio.h` is included outside `main()`, the declarations for `scanf()` and `printf()` are also external. External declaration of functions is convenient since it avoids repeated declarations of the same function. On the other hand, external declarations of variables leads to poorly structured programs and destroys modularity of functions. External declarations of variables is strongly discouraged.

In summary, the syntax of the `#include` directive is:

#include <filename>
#include "filename"

with the semantics that the contents of the file, filename, is to be inserted in the source file in place of the `#include` directive. (Note: here the angle brackets are part of the syntax of the directive). Other directives in the included file are also processed. In the first form of the directive, the header file is searched for in the "usual place" for system header files, and in the second case, it is to be found in the current directory. The advantages of using the `#include` directive are twofold:

1. Information such as macro definitions and prototype statements that are useful in multiple program files need only be entered in a single place and then included where needed. This also facilitates changes; the change need be made only in a single place.

2. Details of macro definitions and prototypes are hidden from the view of the reader, thus alleviating clutter and information overload and allowing a reader of the program to concentrate on the logic of the code itself.

3.3.4 Conditional Compilation

The third useful facility provided by the preprocessor is **conditional compilation**; i.e. the selection of lines of source code to be compiled and those to be ignored. While conditional compilation can be used for many purposes, we will illustrate its use with debug statements. In our previous programming examples, we have discussed the usefulness of `printf()` statements inserted in the code for the purpose of displaying debug information during program testing. Once the program is debugged and accepted as "working", it is desirable to remove these debug statements to use the program. Of course, if later an undetected bug appears during program use, we would like to put some or all debug statements back in the code to pinpoint and fix the bug. One approach to this is to simply "comment out" the debug statements; i.e. surround them with comment markers, so that if they are needed again, they can be "uncommented". This is a vast improvement over removing them and later having to type them back. However, this approach does require going through the entire source file(s) to find all of the debug statements and comment or uncomment them.

The C preprocessor provides a better alternative, namely conditional compilation. Lines of source code that may be sometimes desired in the program and other times not, are surrounded by `#ifdef`,`#endif` directive pairs as follows:

```
#ifdef DEBUG
printf("debug:x = %d, y = %f\n", x, y);
     ...
#endif
```

The `#ifdef` directive specifies that if `DEBUG` exists as a defined macro, i.e. is defined by means of a `#define` directive, then the statements between the `#ifdef` directive and the `#endif` directive are retained in the source file passed to the compiler. If `DEBUG` does not exist as a macro, then these statements are not passed on to the compiler.

Thus to "turn on" debugging statements, we simply include a definition:

```
#define DEBUG  1
```

in the source file; and to "turn off" debug we remove (or comment) the definition. In fact, the replacement string of the macro, `DEBUG` is not important; all that matters is the fact that its definition exists. Therefore,

```
#define DEBUG
```

is a sufficient definition for conditional compilation purposes. During the debug phase, we define `DEBUG` at the head of a source file, and compile the program. All statements appearing anywhere between `#ifdef` and matching `#endif` directives will be compiled as part of the program. When the program has been debugged, we take out the `DEBUG` definition, and recompile the program. The program will be compiled excluding the debug statements. The advantage is that debug statements do not have to be physically tracked down and removed. Also, if a program needs modification, the debug statements are in place and can simply be reactivated.

In general, conditional compilation directives begin with an if-part and end with an endif-part. Optionally, an else-part or an elseif-part may be present before the endif-part. The keywords for the different parts are:

if-part: if, ifdef, ifndef
else-part: else
elseif-part: elif
endif-part: endif

The syntax is:

```
#<if-part>
        <statements>
[ # <elseif-part>
        <statements> ]
[ #<else-part>
        <statements> ]
#<endif-part>
```

If the if-part is True, then all the statements until the next <else-part>, <elseif-part> or <endif-part> are compiled; otherwise, if the <else-part> is present, the statements between the <else-part> and the <endif-part> are compiled.

We have already discussed the keyword ifdef. The keyword ifndef means "if not defined". If the identifier following it is NOT defined, then the statements until the next <else-part>, <elseif-part> or <endif-part> are compiled.

The keyword if must be followed by a constant expression, i.e. an expression made up of constants and operators. If the constant expression is True, the statements until the next else-part, elseif-part or endif-part are compiled. In fact, the keyword ifdef is just a special case of the if form. The directive:

```
#ifdef ident
```

is equivalent to:

```
#if defined ident
```

We can also use `#if` to test for the presence of a device, for example, so that if it is present, we can include an appropriate header file.

```
#if DEVICE == MOUSE
     #include mouse.h
#endif
```

Here, both `DEVICE` and `MOUSE` are assumed to be constant identifiers.

The `#elif` provides a multiway branching in conditional compilation analogous to `else ... if` in C. Suppose, we wish to write a program that must work with any one of a variety of printers. We need to include in the program a header file to support the use of a specific printer. Let us assume that the specific printer used in an installation is defined by a macro `DEVICE`. We can then write conditional compilation directives to include the appropriate header file.

```
#if DEVICE == IBM
     #include ibmdrv.h
#elif DEVICE == HP
     #include hpdrv.h
#else
     #include gendrv.h
#endif
```

Only constant expressions are allowed in conditional compilation directives. Therefore, in the above code, `DEVICE`, `IBM`, and `HP` must be be defined constants.

The niceday Example Again

Using compiler directives is a convenience for the programmer and makes program source files easier to understand. One goal in understandable files is to make them small, the less a reader has to look at in trying to understand a program, the better. Good programming style includes

the hiding of details at the algorithm level with functions, at the source code level using macros, and at the source file level using header files and conditional compilation. One comment should be made about header files. The information stored in header files is meant to be directives and prototype statements, NOT code statements or function definitions. Also **DO NOT**:

```
#include "somefile.c"
```

The syntax of the `#include` directive allows these, but it is considered bad style. A final version of our file `niceday.c` using these compiler directives is shown in Figure 3.11.

3.4 Interacting with the Operating System

In the programs we have developed so far, we have used C library functions `scanf()` and `printf()` to perform the input and output for our programs. These library routines are simply functions that call on the facilities of the operating system to cause data to be the read from the keyboard and written to the screen. In this section we look in more detail at these features of the operating system.

3.4.1 Standard Files and EOF

In our payroll programs, we used a sentinel value of id number, namely 0, to indicate the end of input data. There are many instances when it is not possible to use a special sentinel value of input data to terminate the input. For example, suppose we wish to read a sequence of numbers and determine the largest of them. It is impossible to select any one number as a signal to terminate input since any selected number may be one of the valid numbers in our sequence and may appear before the entire sequence of numbers is exhausted. We need a way to indicate that the end of input is reached without entering any special value of input which may also be valid data.

C provides such mechanism to indicate the end of data input through the way it handles all input and output. All data read by a C program or written from a program can be considered to be simply a **stream** or sequence of characters, i.e. symbols we use to type or print information: alphabetic letters, digits, punctuations, etc. This stream of characters is called a **file** and is organized like any other file in the system. Three files, called **standard input**, **standard output**, and **standard error**, are predefined files available to all programs. By default, **standard input** is the keyboard, and **standard ouput** is the screen. The function `scanf()` reads data from the standard input file, and `printf()` writes data to the standard output file. Run time error messages are written to **standard error**, which is also the screen, by default.

The end of a file is indicated by a special marker which is an unusual character not commonly used for any other purpose. When input is typed at the keyboard, an end of file mark is indicated by what is called a control character. A control character is typed by pressing the *control key*, (CTRL), and pressing another key while keeping CTRL key pressed. For example, control-A is

```
/*   File: niceday.c
     Programmer: Programmer Name
     Date: Current Date
     This program counts the number of nice days in a set of high
     temperature data.
*/

#include <stdio.h>
#include "tfdef.h"
#include "niceday.h"

main()
{  /*  declarations  */
   int temperature,         /*  daily temperature  */
       total = 0,           /*  cumulative total   */
       num_nice_days = 0,
       num_bad_days  = 0;

   /*  print title and prompt  */
   printf("***Count Nice Days***\n\n");
   printf("Type daily high temperature readings (0 to quit): ");

   /*  read the first temperature  */
   scanf("%d", &temperature);
   while (temperature != 0)  {

         /*   process one temperature  */
         if ( nice_day(temperature))
              num_nice_days = num_nice_days + 1;
         else
              num_bad_days = num_bad_days + 1;
         /*  accumulate  total of temperatures  */
         total = total + temperature;
#ifdef DEBUG
printf("debug: %d temps read, total = %d\n",
                   num_nice_days + num_bad_days, total);
#endif

         /*   read next temperature  */
         scanf("%d", &temperature);
   }

   print_results(num_nice_days, num_bad_days, total);

}
```

```
/*  Function to test for a nice day given the temperature        */
int nice_day(int temp)
{
   if( COLD_DAY(temp))  return FALSE;

   if( HOT_DAY(temp))   return FALSE;

   return TRUE;

}

/*  Function to print results given number of nice and bad days  */
/*    and total of temperatures                                  */
int print_results( int nice_days, int bad_days, int total)
{
   float average_temp;

   printf("There were %d nice days and %d bad days\n",
                        nice_days, bad_days);

   if ( ANY_DAYS(nice_days,bad_days))  {
      average_temp = (float) total / (float) (nice_days + bad_days);
      printf("The average temperature for %d days was %f\n",
                        nice_days +  bad_days, average_temp);
   }
}
```

Figure 3.11: Using Directives in `niceday.c`

entered by pressing CTRL and pressing A while keeping CTRL pressed. Control characters are displayed on screen or paper by a caret followed by a letter. For example, control-A is written as ^A. The Control character entered on a keyboard to indicate an end of file is ^D on most Unix machines and ^Z on DOS machines. A keyboard file (stream) with an end of file keystroke is shown in Figure 3.12. Here, three lines of input are represented, followed by the end of file marker as if the user had typed:

89
78
0
^D

How does `scanf()` inform the calling function that an end of file has been reached? It does so by returning a special value to indicate an end of file. The function `scanf()` is just like any

8	9	\n	7	8	\n	0	\n	**EOF**	\n		

Figure 3.12: End of File Marker

other function in C; it has arguments passed to it and it returns a value. So far, we have simply ignored whatever value has returned. Normally, when `scanf()` reads data, it returns a value to indicate the number of data items read successfully. We can save this value returned by `scanf()` and examine whether all data items have been read. For example, consider:

```
flag = scanf("%d", &n);
flag = scanf("%d %f %d", &n, &y, &id);
```

Assuming that both the above statements read data items successfully, then the first `scanf()` will return 1 since it reads one decimal integer, and the second will return 3 since it reads three data items, two `int`'s and a `float`. We have not used this value so far, but we can use it to check if a correct number of items are read.

When `scanf()` detects the special end of file marker, it returns a value of either 0 or -1 (depending on implementation). The actual value returned is defined as a macro called `EOF` in the file `stdio.h`.

We can now write a loop that terminates when the end of standard input file is reached.

```
#include <stdio.h>
...
flag = scanf();
while (flag != EOF) {
     ...
     flag = scanf();
}
```

The value returned by `scanf()` is saved in the variable `flag`. The loop repeats until `flag` receives the value `EOF`. The above code is portable to any implementation since the correct value of `EOF` is defined in `stdio.h` in every implementation. We can now write a program that uses end of file to terminate reading of data.

Task

BIG: Find the largest absolute value in a sequence of integers typed in by the user. An end of file keystroke terminates the input.

The algorithm maintains the current largest absolute value. Each time a new number is read, the absolute value of the item read is compared with the largest value, and if necessary the largest value is updated. The algorithm uses a loop that is terminated when an end of file keystroke is typed. Here is the algorithm.

```
initialize largest to 0
read first integer, n
while there is still data
     compare absolute value of n and largest, update largest
     read next integer
print largest
```

We will need a function `absolute()` which takes an integer argument `n`, and returns its absolute integer value. Notice we initialize our largest absolute value to 0, since that is the smallest absolute value we can ever encounter. The entire program is shown in Figure 3.13 and a sample session is:

```
***Largest Absolute Integer***

Type integers, EOF to quit:  ^Z for DOS, ^D for Unix
-20
0
30
-60
^D
Largest absolute value = 60
```

In our program, `main()` first prompts the user to type integers, and it also tells the user how to terminate the input. It is best to assume that the user does not know how to press a keystroke for `EOF` (however, in the future we will omit this reminder and assume the user knows the correct `EOF` character). The prompt is written by:

```
printf("Type integers, EOF to quit: "
          "^Z for DOS, ^D for Unix\n");
```

Observe that the argument of `printf()` consists of two adjoining strings of characters, each in double quotes. When the compiler encounters two adjoining strings, it replaces them by a concatenated string, i.e. it joins them together into a single string:

```
"Type integers, EOF to quit: ^Z for DOS, ^D for Unix\n"
```

When a string gets too large, it is best to split it into two adjoining strings, since strings cannot be broken across lines.

```
/*   File: maxabs.c
     Programmer: Programmer Name
     Date: Current Date
     This program reads in a sequence of integers until an end of file.
     Among the numbers read, the program determines the largest absolute value.
*/
#include <stdio.h>
int absolute(int n);

main()
{    int largest = 0,
         n, flag;

     printf("***Largest Absolute Integer***\n\n");
     printf("Type integers, EOF to quit: ");
               "^Z for DOS, ^D for Unix\n");
     flag = scanf("%d", &n);

     while (flag != EOF) {
          if (absolute(n) > largest)
               largest = absolute(n);
          flag = scanf("%d", &n);
     }
     printf("Largest absolute value = %d\n", largest);
}

/* Function returns the absolute value of n */
int absolute(int n)
{
     if (n < 0)
          return -n;
     else
          return n;
}
```

Figure 3.13: Code for maxabs.c

After the prompt, `main()` reads the first integer. The while loop tests for the end of the input and compares the value of largest and the absolute value of the last number read, `n`. If necessary largest is updated, a new number is read, and so forth. The loop is terminated when an end of file character (^D or ^Z) is encountered by the function `scanf()` and it returns a value `EOF`. Remember, only the value of `flag`, NOT that of `n`, gets the value, `EOF`. The value of `n` will remain unchanged from its previous value when `scanf()` encounters end of file. Finally, the largest absolute value is printed out.

We have seen that `scanf()` returns a value of items read or `EOF`. It also performs the task of reading one or more items, converting them to internal form, and storing them at specified addresses. This additional task does not directly contribute to the returned value and is called a **side effect**. Functions may be used solely for their side effects, solely for their returned values, or for both side effects and returned values. For example, we use `printf()` for its side effect and ignore its value. We also frequently ignore the value of `scanf()`. In this section, we have used `scanf()` for both its side effect as well as its return value.

3.4.2 Standard Files and Redirection

As we stated, normally the standard input and standard output files are defined by default to be the keyboard and the screen. This may not always be convenient. For example, in our nice day program, we might want to gather statistics for an entire year of temperature data, or an entire decade. While we may have all this data readily available in a file, to use our program we would have to type it all in at the keyboard again (and what happens if we make a mistake and have to start all over). Operating systems such as Unix and MS-DOS allow a user to *redirect* the standard input and output files to files other than the keyboard and screen.

If our program in file, `niceday.c` were compiled using the command:

```
cc -o niceday niceday.c
```

producing the executable file `niceday`, we can execute the program with input data from a file called `temperatures` by typing the following command to the shell:

```
niceday < temperatures
```

The symbol < in the command redirects the standard input to come from the file `temperatures` instead of the keyboard.

Similarly, we can redirect the standard input to our payroll program, `pay5`, from a file containing monthly data for many employees:

```
pay5 < pay_data.march
```

However, in this case, unless we can read *very* fast, most of the output generated by the program will scroll past the screen before we can read it. In addition, we might want to save the results of our program execution in a file to send to a printer for a hard copy. A similar redirection of the standard output to a file can be done with the symbol > as follows:

```
pay5 < pay_data.march > pay_results.march
```

One problem remains with this technique: all output generated by the program from `printf()` statements will be redirected to the file, including the prompts we put in the program. In this case, the prompts are not necessary since the data is coming from a file, not from the user at the keyboard. For programs who's input and output are meant to be redirected from/to files, it is best to remove the `printf()` statements which produce prompts. We might even consider using conditional compilation to include or exclude the prompts, but remember, the program must be recompiled to change from one which prompts to one which does not, and vice versa.

3.5 Debugging Guidelines

As programs become large, finding bugs and debugging become a time consuming job. Debugging is an art that can be learned and developed. However, it requires plenty of experience in writing and debugging programs. The structured, top down approach to writing programs discussed in this chapter is one valuable tool for producing quality, working programs. However, there is no substitute for extensive programming experience and the best way to gain programming experience is to write, test, and debug programs; write, test, and debug programs; write, test, and debug programs; etc. etc.

Certain debugging guidelines are presented here to make the learning process easier:

1. The first step cannot be emphasized enough. Spend plenty of time in preparing the algorithm. A logically clear algorithm is much easier to debug than an ad hoc algorithm with many fixes for previously found bugs. Trial and error programming may never be bug free.

2. Use top down development for your algorithms, and use modular programming for your implementation. Top down development makes logic transparent at each stage and hides unnecessary details by relegating them to later stages. Modular programming localizes errors in small functions, which can be easily debugged.

3. Document your program using comments as you write it. It is a poor habit to delay documenting a program until it is done. Frequently, the very process of documenting a program makes the logic clearer and may well eliminate sources of errors.

4. Trace your program flow manually. This means: examine what happens to values of key variables at key points in the program. Use judicious starting values for these variables. Particularly, check values of variables at critical points, such as loop beginnings and ends, function calls, and other key points in the program.

5. If your compiler comes with a symbolic debugger, learn to use it. The time spent to learn the use of a debugger makes debugging of most programs an easier task.

6. Otherwise, use trace statements in your program. That is, use statements to print out values of key variables at key positions in the program to help pin-point the program segment where the bug may be located. The program segment containing a bug can be narrowed until the exact one or two lines of code are pin-pointed. It is then easier to spot the error and correct it. Trace statements are also called debug statements.

7. Pin-point the functions which generate errors. Rewrite the functions if they are overly complex or long. Many times, it is easier to rewrite a function than to rectify poor logic.

8. In program development, initially we need debug statements. Later, once a program is debugged, the debug statements must be removed. C provides conditional compilation which was discussed above. One use of conditional compilation is to conditionally compile debug statements. Initially the program, including debug statements, is compiled. Later, when the program has been debugged, it can be compiled without compiling the debug statements. Debug statements need not be removed from the code.

3.6 Common Errors

This section contains a list of common errors made by programmers — things to watch out for in your programming.

1. The wrong value is tested for `EOF` instead of the returned value of `scanf()`:

```
flag = scanf("%d", &n);
while (n != EOF)          /* should be: while (flag != EOF) */
     ...
```

   The value read is stored at the address given by `&n`, i.e. it is stored in `n`. The statement `scanf("%d", &n)` evaluates to a returned value which is either the number of data items read or `EOF`. In the above case, if an integer data item is read, the value returned will be 1. If no data item is read, `scanf()` returns `EOF`. The value returned by `scanf()` is stored in the variable `flag`, NOT in `n`. Test `flag` for `EOF`, NOT `n`.

2. An attempt is made by a called function to access a variable defined in the calling function.

```
#include <stdio.h>
#define TRUE 1
main()
{    int x, square(int x);

     x = 3;
     square(x);              /* x cannot be unchanged by square() */
```

```
        printf("x = %d\n", x);          /* prints: x = 3 */
}

int square(int x)           /* x is a new object, with initial value */
                            /* passed by an argument in the function call */
{
        x = x * x;          /* new x is changed */
        return TRUE;    /* a value is returned as the value of square() */
}
```

The variable `x` in `main()` is a different object from `x` in `square()`. The value of the local cell, `x`, is changed in `square()`, but that does not affect the cell `x` in `main()`. The cell, `x`, in `main()` will still have the value 3 after the function call to `square()`. If `main()` needs the squared value of `x`, then `square()` should return the squared value of `x`, NOT `TRUE`. This returned value should be saved in a local variable in `main()`. For example, if the `return` statement in `square()` is:

```
return x;
```

then the returned value can be saved in `main()`:

```
x = square(x);
```

3. A function is not declared with a prototype statement. Without a prototype, the compiler will not be able to check for consistency in usage of the function. When a function is declared, the compiler checks for a correct number of arguments in function calls and checks for correct types.

4. A default declaration of a function assumes an integer type function value. If the actual definition of that function returns a non-integer type, then the compiler will consider it an attempt to redeclare a function. The compiler will flag it as an error.

5. An erroneous keystroke is entered when an end of file is to be entered. For example, an attempt is made to enter 0 or -1 for an end of file. These values are not the end of file keystrokes; they represent the possible values returned by `scanf()` when an end of file keystroke (^D or ^Z) is encountered.

3.7 Summary

This chapter has presented a key concept in the design of good programs; namely, top down design. Beginning with the algorithm, complex programming tasks are divided into logical subtasks which themselves may be further divided. This structured design is a form of information hiding — hiding the details of an operation in its abstraction. We have described how these logical subtasks may be implemented using functions in C. A function is a block of code, which when given some information, performs some operations on the data and returns a value. To invoke (call) a function, use a statement with the form:

<function_name> ([<argument>[,<argument>...]])

where each argument may be an arbitrary expression. A function is defined by specifying a function_header and a function_body. A function header takes the form:

<function_name> ([<parameter>[,<parameter>...]])

and a function body is simply a block containing local variable declarations followed by executable statements to perform the task of the function.

We saw that the <parameter>'s in the function header are really just special forms of variable declarations; containing a type specifier and an identifier. They declare additional local variables within the function which are initialized to the values passed as arguments in the call. We also saw how declaration statements can initialize variables when a block is entered:

<type_specifier><var_name> [=<init_expr>] [, <var_name> [=<init_expr>]...];

Remember, all local variables local to a function may be accessed ONLY within the body of the function, not by functions calling this function and not by functions called by this function.

The value returned by a function is specified in a `return` statement of the form:

return <expression>;

If the last statement of the function is reached without executing a `return` statement, the function returns with an unknown return value.

Next we discussed another form of information hiding using compiler directives processed by the C preprocessor. These included macros, with and without arguments, including header files, and conditional compilation.

#define <symbol_name> <substitution_string>

#include <filename>
#include "filename"

#ifdef <identifier>

(and other variations of the `#if` directive).

Finally, we described the relationship between I/O in C and files, including end of file and redirection of standard input and output files.

3.8 Exercises

1. What will the following code do?

```
#define SQ(x) x * x;
printf("%d\n", SQ(3));
```

2. What will the following code do?

```
#define SQ(x) x * x;
printf("%d\n", SQ(2+3));
```

3. What will be the output of the following code:

```
#define DEBUG 0
#define TWICEZ  z + z

main()
{    int z = 5;

     #ifdef DEBUG
          printf("%d\n", TWICEZ * 2);
     #endif
}
```

4. Check the following program for errors, if any, and use a manual trace to verify the program averages two numbers.

```
#include <stdio.h>
main()
{    float x, y, average;

     printf("Type two numbers: ");
     scanf("%f %f", &x, &y);
     calc_avg(x, y);
     printf("Average of %f and %f is %f\n", x, y, average);
}

calc_avg(float a, float b)
{
     return a + b / 2;
}
```

5. Check the following program for errors, if any, and manually trace its execution.

```
main()
{    float x, y, average;

     printf("Type numbers\n");
     scanf("%f", &x);
     while (x != EOF) {
          printf("Number read = %f\n", x);
          scanf("%f", &x);
     }
}
```

3.9 Problems

1. Write a function `float speed_mph(float distance, float time);` where distance traveled is specified in feet and time interval is in seconds. The function should return the speed in miles per hour. A mile is 5280 feet. Show a manual trace.

2. Write a program that prints out an integer and its square for all integers in the range from 7 through 17. Use a function to calculate the square of an integer. Show a manual trace.

3. Write a program to sum all input numbers until end of file. The program should keep a count of the numbers entered and compute an average of the input numbers. Show a manual trace for the first three numbers.

4. Write a function `float max(float n1, float n2);` that returns the greater of `n1` and `n2`. Write a function `float min(float n1, float n2);` that returns the lesser of `n1` and `n2`. Write a program that reads in numbers and uses the above functions to find the maximum and the minimum of all the numbers. The end of input occurs when zero is typed. Zero is a valid number for determining the maximum and the minimum. Use debug statements to ensure that the maximum and the minimum are updated correctly.

5. Write a program that generates a table of equivalent Celsius (C) and Fahrenheit (F) temperatures from 0 to 212 degrees F. The table entries should be at five degree (F) intervals. Use a function to convert degrees F to C. The conversion between the two is given by:

```
C = (F - 32) * 5.0 / 9.0
```

6. Write a program that uses a function to determine if a given year is a leap year. A year is a leap year if it is divisible by 400; or if it is divisible by 4 and it is not divisible by 100.

7. Write a function `float sum_rec_n(int n);` which returns the sum of the reciprocals of integers from 1 through n. Write a program that reads positive integers until end of file. For each positive integer, `x`, read, it prints `sum_rec_n(x)`. Reciprocals must be `float` values. Use a cast operator to convert an integer to `float` before the reciprocal is calculated.

8. Modify the pay calculation program of Figure 3.2 so that a function `print_data()` prints out the input data as well as the pay. The function `print_data()` should return the number of items it writes to the output.

9. Assume that C does not provide a multiply operator. Write a function, `int multiply(int n1, int n2);` that multiplies two integers `n1` and `n2`, and returns their product. Write a driver to test the function.

10. Write a function, `int factors(int n)`, where `n` is a positive integer. The function prints the smallest integer factors of `n`, excluding 1 and itself. For example, if `n` is 120, then `factors(n)` will print 2, 2, 2, 3, 5. The function returns TRUE if `n` has no factors and FALSE otherwise.

11. Write a program that reads a positive integer and tests if it is a prime number by using `factors()` from Problem 10

12. Write a function `int gcd(int n, int m);` that returns the greatest common divisor (GCD) of non-negative integers `n` and `m`. A GCD may be obtained as follows: if `m` is zero, then GCD is `n`; otherwise, replace current `n` by the current `m` and replace current `m` by `(n % m)`. Repeat until `m` becomes zero and GCD is found.

13. Assume that C does not have a divide operator. Write a function `int_divide()` with two integer arguments that returns an integer quotient when the first argument is divided by the second argument.

14. Assume that C does not have a modulus operator. Write a function `modulus()` with two integer arguments that returns the remainder when the first argument is divided by the second.

15. Write a program that prints the accumulated value of an initial investment invested at a specified annual interest and compounded annually for a specified number of years. Annual compounding means that the entire annual interest is added at the end of a year to the invested amount. The new accumulated amount then earns interest, and so forth. If the accumulated amount at the start of a year is `acc_amount`, then at the end of one year the accumulated amount is:

    ```
    acc_amount = acc_amount + acc_amount * annual_interest
    ```

    Use a function that returns the accumulated value given the amount, interest, and years. The prototype is:

    ```
    float calc_acc_amt(float acc_amount, float annual_interest, int years);
    ```

16. Modify the function in Problem 15 so that the interest may be compounded annually, monthly, or daily. Assume 365 days in the year. Hint: Use an argument to specify annual, monthly, or daily compounding of interest. If interest is not to be compounded annually, the annual interest must be converted to monthly (i.e., interest / 12) or daily interest (i.e., interest / 365). The interest must then be compounded each year, each month, or each day as the case may be.

17. Write a function that calculates the factorial of an integer `n`. Use a driver to test the function for values of `n` from 1 to 7. Factorial of a positive integer, `n`, is given by the product of positive integers from 1 through `n`. Use a variable that stores the value of the cumulative product. The cumulative product is multiplied by a new value of an integer each time a loop is executed:

    ```
    cum_prod = cum_prod * i;
    ```

    The initial value of the cumulative product should be 1 so the first multiple accumulates correctly.

18. Write a function, `float pos_power(float base, int exponent);` which returns the value of base raised to a positive exponent. For example, if base is 2.0 and exponent is 3, the function should return 8.0. If the exponent is negative, the function should return 0.

19. Write a function, `neg_power()`, which returns base raised to a negative exponent.

20. Modify the functions in Problems 18 and 19 to write a function `float power(float base, int exponent);` which returns an exponent power of base, where exponent may be positive or negative. If the exponent is zero, it should return 1.

21. Write a function `int weight(int n);` where `n` is a positive integer. The function returns the weight of the most significant digit, i.e., the highest power of ten which does not exceed `n`. For example, if `n` is 2345, `weight(n)` returns 1000. Assume `n` is less than 10000.

22. Write a function, `int sig_dig_value(int n);` that returns the integer value of the most significant digit of a positive integer `n` less than 10000. For example, if `n` is 2345, `sig_dig_value(n)` returns integer 2.

23. Write a function, `int suppress_msd(int n);` that returns an integer value of a positive integer after the most significant digit is removed. For example, if `n` is 2345, `suppress_msd(n)` returns 345.

24. Use Problems 22 and 23 to write a function, `print_dig_int(int n);` that prints successive integer values of digits of a positive integer `n`. Each digit value is printed on a separate line. For example, if `n` is 2345, `print_dig_int(n)` prints 2 on one line, 3 on the next, 4 on the next, and 5 on the last line.

25. Write a function `print_dig_float(float x);` that writes the value of each digit of a floating point number `x`. For example, if `x` is 2345.1234, then `print_dig_float(x)` will print integer values of digits 2, 3, 4, 5, 1, 2, 3, and 4 in succession.

26. Write a macro to evaluate the sum of the squares of two parameters. Make sure the macro can be called with any argument expressions. Write a program that reads two values and uses the above macro to print the sum of the squares.