Chapter 2

Basic Concepts

Learning to program is a lot like learning to speak a new language. You must learn new *vocabulary*, i.e. the *words* of the language; the *syntax*, (also called the **grammar**); i.e. the form of statements in the language, as well as the *semantics*, i.e. the *meaning* of the words and statements. This learning process usually begins slowly but often you find that with just a few basic words and phrases you can begin conversing and getting your thoughts across. In this chapter we present a few of the basic statements of the C language so that you can write programs from the beginning.

As in spoken languages, the first thing you need is something to say – an idea. In the programming world, this idea is often in the form of a *task*, i.e. something you would like to have done by the computer. The task may be described in terms of what information is to be provided to the computer, what is to be done with this information, and what results should be produced by the program. A program is often developed in small increments, starting with a relatively simple version of the task and progressing to more complex ones, adding features until the entire task can be solved. The focus is always on the task to be performed. The task must be clearly understood in order to proceed to the next step, the development of an *algorithm.* As was discussed in the previous chapter, an **algorithm** is a step by step description of what must be done to accomplish a task. These can be considered to be the most important steps in programming; specifying and understanding the task (what is to be done), and designing the algorithm (how it is to be done). We take this approach beginning in this chapter, and we will discuss task development and algorithm design in more detail in Chapter 3.

Once an algorithm is clearly stated, the next step is to translate the algorithm into a *programming language.* In our case this will be the C language. Using the vocabulary, syntax, and semantics of the language, we can *code* the program to carry out the steps in the algorithm. After coding a program, we must test it by running it on the computer to ensure that the desired task is indeed performed correctly. If there are **bugs**, i.e. errors in the program, they must be removed; in other words an erroneous program must be *debugged* so it performs correctly. The job of programming includes the entire process: algorithm development, and coding, testing and debugging the program.

At the end of the Chapter, you should know:

- How to code simple programs in C.
- How a program allocates memory to store data, called **variables**.
- How variables are used to store and retrieve data, and to make numeric calculations.
- How decisions are made based on certain events, and how a program can branch to different paths.
- How a set of computations can be repeated any number of times.
- How a program can be tested for errors and how the errors may be removed.

2.1 A Simple C Program

The easiest way to learn programming is to take simple tasks and see how programs are developed to perform them. In this section we will present present one such program explaining what it does and showing how it executes. A detailed description of the syntax of the statments used is given in Section 2.2.

2.1.1 Developing the Algorithm

In the previous chapter we introduced a payroll task which can be summarized as a task to calculate pay for a number of people employed by a company. Let us assume that each employee is identified by an id number and that his/her pay is computed in terms of an hourly rate of pay. We will start with a simple version of this task and progress to more complex versions. The simplest version of our task can be stated as follows.

Task

PAY0: Given the hours worked and rate of pay, write a program to compute the pay for a person with a specified id number. Print out the data and the pay.

The algorithm in this case is very simple:

```
print title of program;
set the data: set id number, hours worked, and rate of pay;
set pay to the product of hours worked and rate of pay;
print the data and the results;
```

With this algorithm, it should be possible, without too much trouble, to implement the corresponding program in almost any language since the fundamental constructs of most algorithmic programming languages are similar. While we will discuss the features of C, similar features are usually available for most high level languages.

2.1.2 Translating the Algorithm to C

A program in a high level language, such as C, is called a **source program** or **source code**. (*Code* is a generic term used to refer to a program or part of a program in any language, high or low level). A program is made up of two types of items: *data* and *procedures*. *Data* is information we wish to process and is referred to using its *name*. *Procedures* are descriptions of the required steps to process the data and are also given names. In C, all procedures are called **functions**. A program may consist of one or more functions, but it must always include a function called *main*. This special function, `main()`, acts as a controller; directing all of the steps to be performed and is sometimes called the **driver**. The driver, like a conductor or a coordinator, may call upon other functions to carry out subtasks. When we refer to a function in the text, we will write its name followed by parentheses, e.g. `main()`, to indicate that this is the name of a function.

The program that implements the above algorithm in C is shown in Figure 2.1. Let us first look briefly at what the statements in the above program do during execution.

Any text between the markers, `/*` and `*/` is a **comment** or an explanation; it is not part of the program and is ignored by the compiler. However, comments are very useful for someone reading the program to understand what the program is doing. We suggest you get in the habit of including comments in your programs right from the first coding. The first few lines between `/*` and `*/` are thus ignored, and the actual program starts with the function name, `main()`. Parentheses are used in the code after the function name to list any information to be given to the function, called **arguments**. In this case, `main()` has no arguments. The *body* of the function `main()` is a number of *statements* between braces { and }, each terminated by a semi-colon.

The first two statements declare variables and their data types: `id_number` is an integer type, and `hours_worked`, `rate_of_pay`, and `pay` are floating point type. These statements indicate that memory should be allocated for these kinds of data and gives names to the allocated locations. The next statement writes or prints the title of the program on the screen.

The next three statements set the variables `id_number`, `hours_worked`, and `rate_of_pay` to some initial values: `id_number` is set to 123, `hours_worked` to 20.0, and `rate_of_pay` to 7.5. The next statement sets the variable `pay` to the product of the values of `hours_worked` and `rate_of_pay`. Finally, the last three statements print out the initial data values and the value of `pay`.

2.1.3 Running the Program

The program is entered and stored in the computer using an editor and saved in a *file* called `pay0.c`. The above *source program* must then be *compiled*, i.e. translated into a machine language *object program* using a *compiler*. Compilation is followed, usually automatically, by a *linking* process during which the compiled program is joined with other code for functions that may be defined

```
/*   File: pay0.c
     Programmer: Programmer Name
     Date: Current Date
     This program calculates the pay for one person, given the hours worked
     and rate of pay.
*/

main()
{    /* declarations */
     int id_number;
     float hours_worked,
          rate_of_pay,
          pay;

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

     /* initialize variables */
     id_number = 123;
     hours_worked = 20.0
     rate_of_pay = 7.5;

     /* calculate pay */
     pay = hours_worked * rate_of_pay;

     /* print data and results */
     printf("ID Number = %d\n", id_number);
     printf("Hours Worked = %f, Rate of Pay = %f\n",
               hours_worked, rate_of_pay);
     prinf("Pay = %f\n", pay);
}
```

Figure 2.1: Code for pay0.c

elsewhere. The C language provides a **library** of standard functions which are linked to every program and are available for use in the program. The end result is an *executable* machine language program also in a file. The executable machine language program is the only one that can be executed on a machine. We will use the term **compilation** to mean both compiling and linking to produce an executable program.

When the above program is compiled and executed on a computer, a sample session produces the following on the terminal:

```
***Pay Calculation***

ID Number = 123
Hours Worked = 20.000000, Rate of Pay = 7.500000
Pay = 150.000000
```

Throughout this text, we will show all information printed by the computer in `typewriter style characters`. As programs will frequently involve data entry by the user of the program during execution, in a sample session, all information typed in by the user will be shown in *slanted characters*.

2.2 Organization of C Programs — Simple Statements

We will now explain the syntax and semantics of the above program statements in more detail. Refer back to the source program in Figure 2.1 as we explain the statements in the program.

2.2.1 Comment Statements

As already mentioned, the text contained within `/*` and `*/` is called a comment. When the character pair `/*` is encountered, all subsequent text is ignored until the next `*/` is encountered. Comments are not part of the program; they are private notes that the programmer makes about the program to help one understand the logic. Comments may appear anywhere in a program but cannot contain other comments, i.e., they cannot be nested. For example:

```
/* This is a comment. /* Nested comments are not allowed */ this part
is not in a comment. */
```

The comment starts with the first `/*`. When the first matching `*/` is encountered after the word `allowed`, the comment is ended. The remaining text is not within the comment and the compiler tries to interpret the remaining text as program statement(s), most likely leading to errors.

2.2.2 Defining a Function — `main()`

To define a function in C, the programmer must specify two things: the **function header**, giving a name and other information about the function; and the **function body**, where the variables used in the function are defined and the statements which perform the steps of the function are specified.

The Function Header

In C, `main()` is the function that controls the execution of every program. The program starts executing with the first statement of `main()` and ends when `main()` ends. As we shall soon see, `main()` may call upon, i.e. use, other functions to perform subtasks.

The first line of any function is the function header which specifies the name of the function together with a parenthesized (possibly empty) argument list. In the above case, there is no argument list. We will discuss the concepts of arguments and argument lists in the next chapter.

The Function Body

The body of the function is contained within braces { and }. In C, a group of statements within braces is called a **block** which may contain zero or more statements and which may be nested, i.e. there may be blocks within blocks. A block is treated and executed as a single unit and is often called a **compound statement**. Such a compound statement may be used anywhere a statement can occur.

A program statement is like a sentence in English, except that it is terminated by a semi-colon. Statements within a block may be written in free form, i.e. words in programs may be separated by any amount of **white space**. (White space consists of spaces, tabs, or newlines (carriage returns)). Use of white space to separate statements and parts of a single statement makes programs more readable and therefore easier to understand.

The function body (as for any block) consists of two parts: *variable declarations* and a *list of statements*. Variable declarations will be described in more detail in the next section; however, all such declarations must occur at the beginning of the block. Once the first executable statement is encountered, no more declarations may occur for that block.

There are two types of statements used in our example (Figure 2.1); assignment statements and statements for printing information from the program. These will be discussed more below. The execution *control flow* proceeds sequentially in this program; when the function is executed, it begins with the first statement in the body and each statement is executed in succession. When the end of the block is reached, the function terminates. As we will soon see, certain control statements can alter this sequential control flow in well defined ways.

2.2.3 Variable Declarations

A **variable** is a language construct for identifying the data items used within a block. The declaration statements give names to these data items and specify the *type* of the item. The first two statements in our program are such declarations. The information we have in our task is the employee ID, the number of hours worked by the employee and the rate of pay. In addition, we will compute the total amount of pay for the employee and must declare a variable for this information. We have named variables for this information: `id_number`, `hours_worked`, `rate_of_pay`, and `pay`. We have also specified the type of each; for example, `id_number` is a whole number which requires an integer type, so the keyword `int` is used. The remaining data items are real numbers (they can have fractional values), so the keyword `float` is used to specify floating point type.

Variables of appropriate type (`int`, `float`, etc.) must be declared at the head of the block in which they are used. Several variables of the same type may be grouped together in a declaration, separated by commas.

```
int id_number;
float hours_worked,
      rate_of_pay,
      pay;
```

The names we have chosen for the variables are somewhat arbitrary; however, to make programs readable and easier to understand, variable names should be descriptive and have some meaning to the programmer. In programming languages, names are called **identifiers** and must satisfy certain rules.

First, identifiers may not be **keywords** (such as `int` and `float`) which have special meaning in C and are therefore reserved. All of these reserved words are listed in Appendix A. Otherwise, identifiers may include any sequence of lower and upper case letters, digits, and underscores; but the first character must be a letter or an underscore (though the use of an underscore as a first character is discouraged). Examples of legal identifiers include `PAD12`, `pad39`, `room_480`, etc. Alphabetic letters may be either lower case or upper case which are different; i.e. `PAY`, `Pay`, and `pay` are distinct identifiers for three different objects. There is no limit to the length of an identifier, however, there may be an implementation dependent limit to the number of significant characters that can be recognized by a compiler. (This means that if two identifiers do not differ in their first n characters, the compiler will not recognize them as distinct identifiers. A typical value for n might be 31).

The general form for a declaration statement is:

<type_specifier> <identifier>[, <identifier>...];

Throughout this text we will be presenting syntax specifications as shown above. The items surrounded by angle brackets (<>) are constructs of the language, for example <type_specifier> is a type specifier such as `int` or `float`, and <identifier> is a legal identifier. Items surrounded

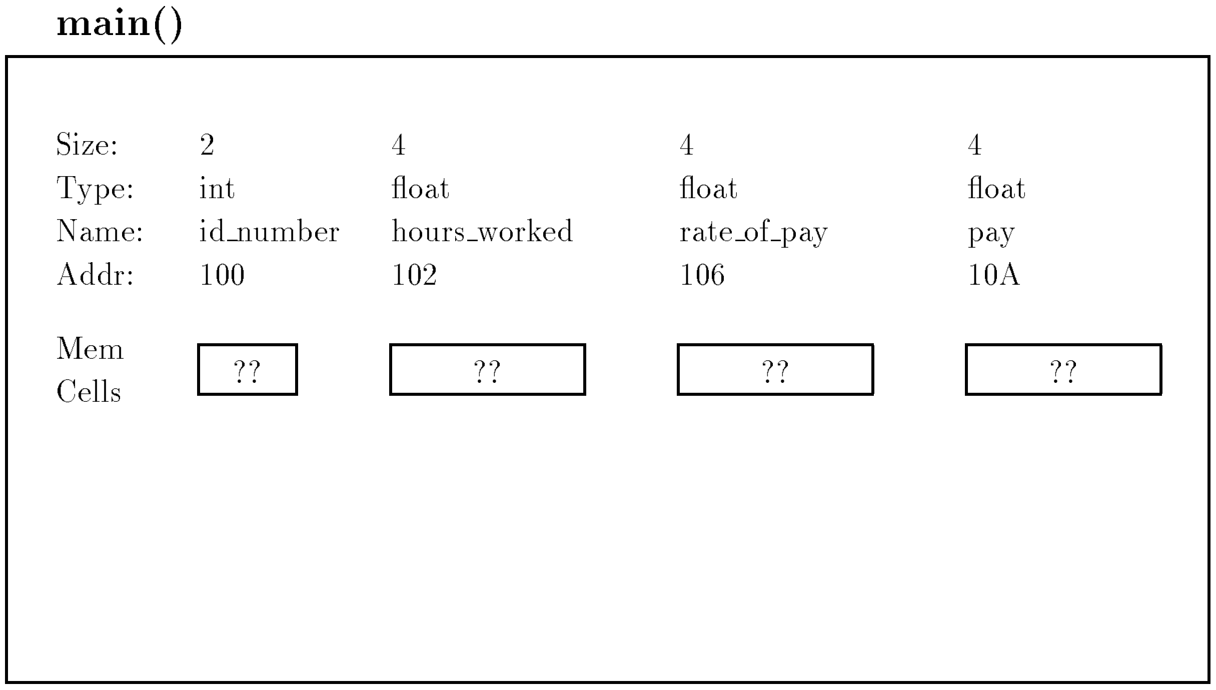
main()
Size: 2 4 4 4
Type: int float float float
Name: id_number hours_worked rate_of_pay pay
Addr: 100 102 106 10A
Mem Cells
??
??
??
??

Figure 2.2: Allocation of Memory Cells or Objects

by square brackets ([ ]) are optional, i.e. they may or may not appear in a legal statement. The ellipsis (...) indicates one or more repetitions of the preceding item. Any other symbols are included in the statement exactly as typed. So, in words, the above syntax specification says that a declaration statement consists of a type specifier followed by an identifier and, optionally, one or more other identifiers separated by commas, all terminated by a semicolon.

As for the semantics (meaning) of this statement, a declaration statement does two things: allocates memory within the block for a data item of the indicated type, and assigns a name to the location. As we saw in Chapter 1, data is stored in the computer in a binary form, and different types of data require different amounts of memory. Allocating memory for a data item means to reserve the correct number of bytes in the memory for that type, i.e. choosing the address of the memory cells where the data item is to be stored.

Figure 2.2 shows memory allocation for the declarations in our program as it might occur on a 16 bit machine. The outer box shows that these variables have been allocated for the function `main()`. For each variable we show the size of the data item (in bytes), its type, name and assigned address assignment (in hex) above the box representing the cell itself. In the future, we will generally show only the memory cell and its name in similar diagrams. Note that the declaration statements **do not** put values in the allocated cells. We indicate this with the *??* in the boxes.

Memory cells allocated for specific data types are called objects. An object is identified by its starting address and its type. The type determines the size of the object in bytes and the encoding used to represent it. A variable is simply a named object which can be accessed by using its name. An analogy is gaining access to a house identified by the name of the person living there: Smith house, Anderson house, etc.

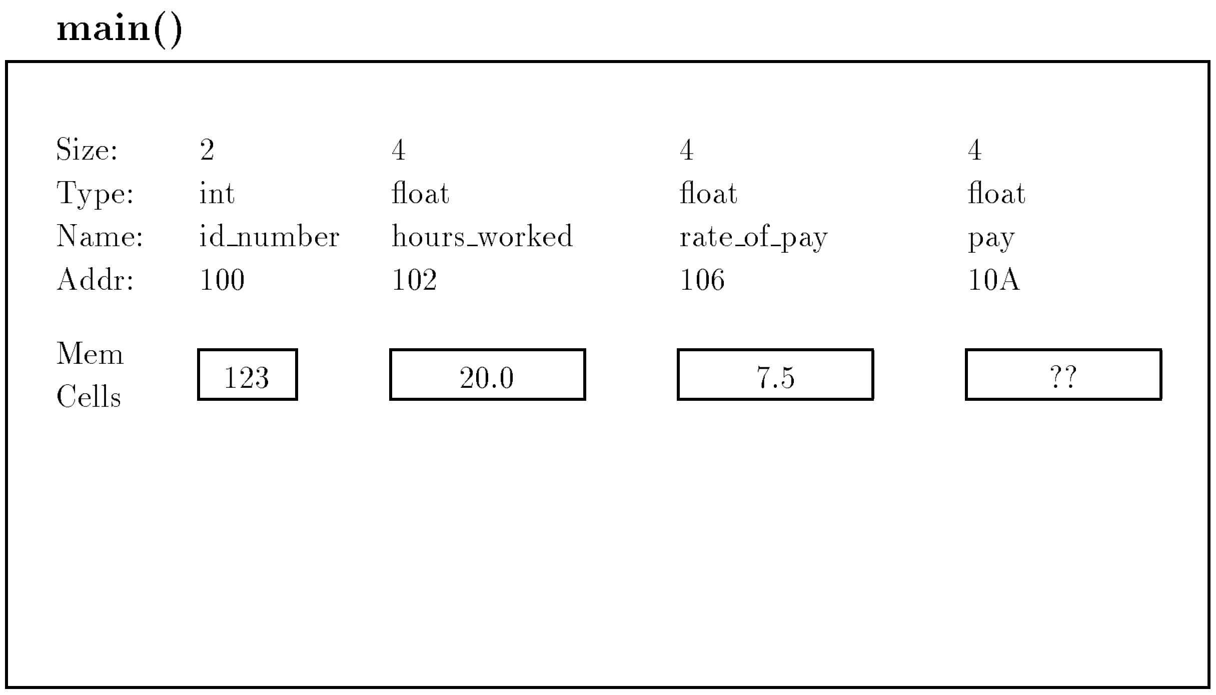
main()
Size: 2 4 4 4
Type: int float float float
Name: id_number hours_worked rate_of_pay pay
Addr: 100 102 106 10A
Mem Cells
123
20.0
7.5
??

Figure 2.3: Assignment of Values

Memory is automatically allocated for variables declared in a block when the block is entered during execution, and the memory is freed when the block is exited. Such variables are called **automatic variables**. The **scope** of automatic variables, i.e. the part of a program during which they can be used directly by name, is the block in which they are defined.

2.2.4 The Assignment Statement

The next three statements in our program assign initial values to variables, i.e. store initial values into objects represented by the variables. The assignment operator is `=`.

```
id_number = 123;
hours_worked = 20.0;
rate_of_pay = 7.5;
```

Each of the above statements stores the value of the expression on the right hand side of the assignment operator into the object referenced by the variable on the left hand side, e.g. the value stored in `id_number` is 123 (Figure 2.3). We will say the (current) value of `id_number` is 123. The value of a variable may change in the course of a program execution; for example, a new assignment can store new data into a variable. Storing new data overwrites the old data; otherwise, the value of a variable remains unchanged.

The "right hand side" of these three assignments is quite simple, a decimal constant. (The compiler will take care of converting the decimal number we use in the source code into its

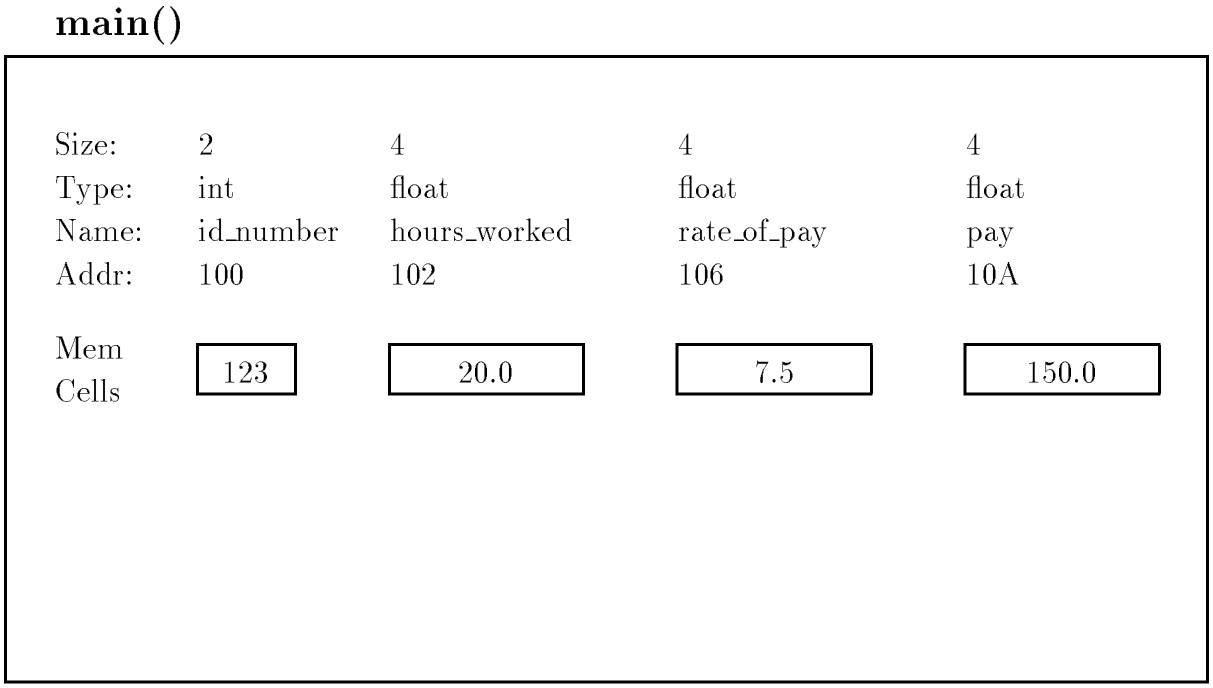
main()
Size: 2 4 4 4
Type: int float float float
Name: id_number hours_worked rate_of_pay pay
Addr: 100 102 106 10A
Mem Cells
123
20.0
7.5
150.0

Figure 2.4: Computation of `pay`

appropriate binary representation). However, in general the right hand side of an assignment may be an arbitrary *expression* consisting of constants, variable names and arithmetic operators (functions may also occur within expressions). For example, next, we calculate the product of the value of `hours_worked` and the value of `rate_of_pay`, and assign the result to the variable pay. The multiplication operator is `*`.

```
pay = hours_worked * rate_of_pay;
```

The semantics of the assignment operator is as follows: the expression on the right hand side of the assignment operator is first evaluated by replacing each instance of a variable by its current value and the operators are then applied to the resulting operands. Thus, the above right hand side expression is evaluated as:

```
20.0 * 7.5
```

The resulting value of the expression on the right hand side is then assigned to the variable on the left hand side of the assignment operator. Thus, the value of 20.0 * 7.5, i.e. 150.0, is stored in `pay` (Figure 2.4).

The above assignment expression may be paraphrased in English as follows:

"SET pay TO THE VALUE OF hours_worked * rate_of_pay"

or,

"ASSIGN TO pay THE VALUE OF hours_worked * rate_of_pay"

The syntax of an assignment statement is:

<Lvalue>=<expression>;

The class of items allowed on the left hand side of an assignment operator is called an **Lvalue**, a mnemonic for *left value*. Of course, <Lvalue> must always reference an object where a value is to be stored. In what we've see so far, only a variable name can be an <Lvalue>. Later we will see other ways of referencing an object which can be used as an <Lvalue>.

As we can see from the above discussion, variables provide us a means for accessing information in our program. Using a variable on the left hand side of an assignment operator allows us to store a value in its memory cell. Variables appearing elsewhere in expressions cause the current value of the data item to be read and used in the expression.

In C every expression evaluated during execution results in a value. Assignment is also an expression, therefore also results in a value. Assignment expressions may be used anywhere expressions are allowed. The rule for evaluating an assignment expression is: evaluate the expression on the right hand side of the assignment operator, and assign the value to the variable on the left hand side. The value of the entire assignment expression is the value assigned to the left hand side variable. For example, `x = 20` assigns 20 to `x`, and the value of the entire assignment expression is 20. So if we wrote `y = x = 20`, the variable `y` would be assigned the value of the expression `x = 20`, namely 20. In our programming example we have used assignment expressions as statements but ignored their values.

Any expression terminated by a semi-colon is a statement. Of course, a statement is typically written to perform some useful action. Some additional examples of expressions as statements are:

```
20;
5 + 10;
z = 20 * 5 + 10;
;
```

The last statement is an empty statement which does nothing. The expressions in the first two statements accomplish nothing since nothing is done with their values.

C has a rich set of operators for performing computations in expressions. The common arithmetic operators and their meanings are shown in Table 2.1. Two types of operators are shown; unary operators which take one operand, and binary operators which take two operands. The unary operators, `+` and `-` affect the sign of the operand. The binary operators are those you are familiar with, except possibly `%`. This is the **mod** operator, which we will describe below, but first one other point to make is that for the division operator `/`, if both operands are type integer, then integer division is performed, discarding and fractional part with the result also being type

Operator	Name	Example and Comments
$+$	plus sign	$+x$
$-$	minus sign	$-x$
$+$	addition	$x + y$
$-$	subtraction	$x - y$
$*$	multiplication	$x * y$
/	division	x/y if x, y are both integers, then x/y is integer, e.g., 5/3 is 1.
%	modulus	$x\%y$ x and y MUST be integers: result is remainder of (x/y), e.g., 5%3 is 2.

Table 2.1: Arithmetic Operators

integer. Otherwise, a floating point result is produced for division. The mod operator evaluates to the remainder after integer division. Specifically, the following equality holds:

$$(x/y) * y + (x\%y) = x.$$

In words, if x and y are integers, multiplying the result of integer division by the denominator and adding the result of mod produces the numerator. We will see many more operators in future chapters.

2.2.5 Generating Output

Writing programs which declare variables and evaluate expressions would not be very useful if there were no way to communicate the results to the user. Generally this is done by printing (or writing) messages on the output.

Output of Messages

It is a good practice for a program to indicate its name or title when it is executed to identify the task which is being performed. The next statement in our program is:

```
    printf("***Pay Calculation***\n\n");
```

The statement prints the program title on the terminal. This statement invokes the standard function `printf()` provided by every C compiler in a standard library of functions. The function `printf()` performs the subtask of writing information to the screen. When this statement is executed, the flow of control in the program passes to the code for `printf()`, and when `printf()` has completed whatever it has to do, control returns to this place in the program. These sequence of events is called a **function call**.

As can be seen in this case, a function can be called by simply using its name followed by a (possibly empty) pair of parentheses. Anything between the parentheses is called an **argument** and is information being sent to the function. In the above case, `printf()` has one argument, a string of characters surrounded by double quotes, called a **format string**. As we shall soon see, `printf()` can have more than one argument; however, the first argument of `printf()` must always be a format string. This `printf()` statement will write the following to the screen:

```
***Pay Calculation***
```

followed by two newlines. Note that all of the characters *inside* the double quotes have been printed (but not the quotes themselves), except those at the end of the string. The backslash character, '\', in the string indicates an **escape sequence**. It signals that the next character must be interpreted in a special way. In this case, '\n' prints out a newline character, i.e. all further printing is done on the next line of output. We will encounter other escape sequences in due time. Two newline escape sequences are used here; the first completes the line where "***Pay Calculation***" was written, and the second leaves a blank line in the output.

Output of Data

In addition to printing fixed messages, `printf()` can be used to print values of expressions by passing the values as additional arguments separated by commas. We print out values of the initial data and the result with the statements:

```
printf("ID Number = %d\n", id_number);
printf("Hours Worked = %f, Rate of Pay = %f\n",
          hours_worked, rate_of_pay);
printf("Pay = %f\n", pay);
```

The first argument of `printf()` must always be a format string and may be followed by any number of addition argument expressions (in this case simple variable names). As before, all regular characters in the format string are printed until the symbol %. The % and the following character, called a **conversion specification**, indicate that the value of the next argument is to be printed at this position in the output. The conversion character following % determines the format to be printed. The combination %d signals that a decimal integer value is to be printed at this position. Similarly, %f indicates that a decimal floating point value is to be printed at the indicated position. (To write a % character itself, use %% in the format string). Each conversion specifier in the format string will print the value of one argument in succession.

The first `printf()` statement prints the value of `id_number` in the position where %d is located. The internal form of the value of `id_number` is converted to a decimal integer format and printed out. The output is:

```
ID Number = 123
```

The next `printf()` writes the value of `hours_worked` at the position of the first %f, and the value of `rate_of_pay` at the position of the second %f. The internal forms are converted to decimal real numbers (i.e., floating point) and printed. The output is:

```
Hours Worked = 20.000000, Rate of Pay = 7.500000
```

Observe that all regular characters in the format string, including the newline, are printed as before. Only the format conversion specification, indicated by a % followed by a conversion character d or f, is replaced by the value of the next unmatched argument. The floating point value is printed with six digits after the decimal point by default.

The final statement prints:

```
pay = 150.000000
```

2.3 Testing the Program

As mentioned, the above program must be typed using an editor and saved in a file which we have called `pay0.c`. The program in C, a high level language, is called the **source program** or **source code**. It must be translated into the machine language for the particular computer being used. The machine language program is the only one that can be understood by the hardware.

A special program called a **compiler** is used to compile, i.e. translate a source program into a machine language program. The resulting machine language program is called the **object code** or **object program**. The object code may be automatically or optionally saved in a file. The terms **source file** and **object file** refer to the files containing the corresponding source code and object code.

The compiled object code is usually still not executable. The object code needs to be linked to machine language code for certain functions, e.g. code for library functions such as `printf()`, to create an executable machine language code file. A linker or a link editor is used for this step of linking disparate object codes. The linking step is usually automatic and transparent to the user. We will refer to the executable code variously as the **object code**, the **compiled program**, or the **load module**.

The executable code is then loaded into memory and run. The loading step is also transparent to the user; the user merely issues a command to run the executable code.

For many systems, the convention is that the source file name should end in `.c` as in `pay0.c`. Conventions for object file names differ; on some systems object files end in `.obj`, on others they end in `.o`, (Consult your system manuals for details). For compilation and execution, some systems require separate commands, one to compile a C program and the other to execute a compiled program. Other systems may provide a single command that both compiles and executes a program. Check your operating system and compiler manuals for details.

For Unix systems, the `cc` command, with many available options, is used for compilation. Examples are:

```
    cc filename.c
    cc -o outname filename.c
```

The first command line compiles the file `filename.c` and produces an executable file `a.out`. The second directs that the executable file is to be named `outname`. These programs are then run by typing the executable file name to the shell.

2.3.1 Debugging the Program

A program may have bugs, i.e. errors, in any of the above phases so these bugs must be removed; a process called **debugging**. Some bugs are easy to remove; others can be difficult. These bugs may appear at one of three times in testing the program: compile time, link time, and run time.

When a program is compiled, the compiler discovers syntax (grammar) errors, which occur when statements are written incorrectly. These compile time errors are easy to fix since the compiler usually pinpoints them reasonably well. The astute reader may have noticed there are bugs in the program shown in Figure 2.1. When the file `pay0.c` is compiled on a Unix C compiler, the following message is produced:

```
"pay0.c", line 21: syntax error at or near variable name "rate_of_pay"
```

This indicates some kind of syntax error was detected in the vicinity of line 21 near the variable name `rate_of_pay`. On examining the file, we notice that there is a missing semi-colon at the end of the previous statement:

```
hours_worked = 20.0
```

Inserting the semi-colon and compiling the program again eliminates the syntax error. In another type of error, the linker may not be able to find some of the functions used in the code so the linking process cannot be completed. If we now compile our file `pay0.c` again, we receive the following message:

```
/bin/ld: Unsatisfied symbols:
   prinf (code)
```

It indicates the linker was unable to find the function `prinf` which must have been used in our code. The linker states which functions are missing so link time errors are also easy to fix. This error is obvious, we didn't mean to use a function, `prinf()`, but merely misspelled `printf()` in the statement

```
     prinf("Pay = %f\n", pay);
```

Fixing this error and compiling the program again, we can successfully compile and link the program, yielding an executable file. As you gain experience, you will be able to arrive at a program free of compile time and link time errors in relatively few iterations of editing and compiling the program, maybe even one or two attempts.

A program that successfully compiles to an executable does not necessarily mean all bugs have been removed. Those remaining may be detected at run time; i.e. when the program is executed and may be of two types: computation errors and logic errors. An example of the former is an attempt to divide by zero. Once these are detected, they are relatively easy to fix. The more difficult errors to find and correct are program logic errors, i.e. a program does not perform its intended task correctly. Some logic errors are obvious immediately upon running the program; the results produced by the program are wrong so the statement that generates those results is suspect. Others may not be discovered for a long time especially in complex programs where logic errors may be hard to discover and fix. Often a complex program is accepted as correct if it works correctly for a set of well chosen data; however, it is very difficult to prove that such a program is correct in all possible situations. As a result, programmers take steps to try to avoid logic errors in their code. These techniques include, but are not limited to:

Careful Algorithm Development

As we have stated, and will continue to state throughout this text, careful design of of the algorithm is perhaps the most important step in programming. Developing and refining the algorithm using tools such as the structural diagram and flow chart discussed in Chapter 1 before any coding helps the programmer get a clear picture of the problem being solved and the method used for the solution. It also makes you think about what must be done before worrying about how to do it.

Modular Programming

Breaking a task into smaller pieces helps both at the algorithm design stage and at the debugging stage of program development. At the algorithm design stage, the modular approach allows the programmer to concentrate on the overall meaning of what operations are being done rather than the details of each operation. When each of the major steps are then broken down into smaller steps, again the programmer can concentrate on one particular part of the algorithm at a time without worrying about how other steps will be done.

At debug time, this modular approach allows for quick and easy localization of errors. When the code is organized in the modules defined for the algorithm, when an error does occur, the programmer can think in terms of *what* the modules are doing (not *how*) to determine the most likely place where something is going wrong. Once a particular module is identified, the same refinement techniques can be used to further isolate the source of the trouble without considering all the other code in other modules.

Incremental Testing

Just as proper algorithm design and modular organization can speed up the debugging process, incremental implementation and testing can assist in program development. There are two approaches to this technique. The first is to develop the program from simpler instances of the task to more complex tasks as we are doing for the payroll problem in this chapter. The idea is to implement and test a simplified program and then add more complicated features until the full specification of the task is satisfied. Thus beginning from a version of the program known to be working correctly (or at least thoroughly tested), when new features are added and errors occur, the location of the errors can be localized to added code.

The second approach to incremental testing stems from the modular design of the code. Each module defined in the design can be implemented and tested independently so that there is high confidence that each module is performing correctly. Then when the modules are integrated together for the final program, when errors occur, again only the added code need be considered to find and correct them.

Program Tracing

Another useful technique for debugging programs begins after the program is coded, but before it is compiled and run, and is called a **program trace**. Here the operations in each statement of the program are verified by the programmer. In essence, the programmer is executing the program manually using pencil and paper to keep track changes to key variables. Diagrams of variable allocation such as those shown in Figures 2.2—2.4 may be used for this manual trace. Another way of manually tracing a program is shown in Figure 2.5. Here the changes in variables is seen associated with the statement which caused that change.

Program traces are also useful later in the debug phase. When an error is detected, a selective manual trace of a portion or module of a program can be very instrumental in pinpointing the problem. One word of caution about manual traces — care must be taken to update the variables in the trace according to the statement as written in the program, not according to the intention of the programmer as to what that statement should do.

Manual traces can become very complicated and tedious (one rarely traces an entire program),

```
/*   File: pay0.c
     Programmer: Programmer Name
     Date: Current Date
     This program calculates the pay for one person, given the
     hours worked and rate of pay.
*/

main()                                       PROGRAM TRACE
{                                              hours_  rate_of_
     /* declarations */             id_number worked  pay       pay
     int id_number;                    ??
     float hours_worked,                         ??
          rate_of_pay,                                ??
          pay;                                                  ??

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

     /* initialize variables */
     id_number = 123;                  123       ??    ??        ??
     hours_worked = 20;                123       20.0  ??        ??
     rate_of_pay = 7.5;                123       20.0  7.5       ??

     /* calculate results */
     pay = hours_worked * rate_of_pay;
                                       123       20.0   7.5      150.0

     /* print data and results */
     printf("ID Number = %d\n", id_number);
     printf("Hours Worked = %f, Rate of Pay = %f\n",
               hours_worked, rate_of_pay);
     printf("Pay = %f\n", pay);
}
```

Figure 2.5: Program Trace for pay0.c

however selective application of this technique is a valuable debugging tool. Later in this chapter we will discuss how the computer itself can assist us in generating traces of a program.

2.3.2 Documenting the Code

As a programmer, there are several "good" habits to develop for translating an algorithm into a source code program which support debugging as well as general understanding of the code. These habits fall under the topic of "coding for readability". We have already mentioned a few of these such as commenting the code and good choices of names for variables and functions. With good naming, the syntax of the C language allows for relatively good *self documenting code*; i.e. C source statements which can be read and understood with little effort.

Well documented code includes additional comments which clarify and amplify the meaning or intention of the statements. A good source for comments in your code are the steps of the algorithm you designed for the program. A well placed comment identifying which statements implement each step of the algorithm makes for easily understood programs.

Another good habit is to include judicious amounts of white space in your program. The C compiler would accept your program all written on one line; however, this would be very difficult for someone to read. Instead, space out your statements, separating groups of statements that perform logically different operations. It is also good to indent the statements in your program so that blocks are clearly identified at a glance. You will notice we have done that in Figure 2.1 and will continue throughout this text. There is no standard for indenting code, so you should choose a convention that is natural for you, as long as it is clear and you are consistent.

One last point: even though we have concentrated on the documentation of the code at the end of our discussion on this program, good documentation should be considered throughout the programming process. A bad habit to get into is to write the code and document it after it is working. A good habit is to include documentation in the code from the beginning.

In this section we have looked in detail at a C program that solves our simplified version of the payroll problem. The program in file `pay0.c` is not very useful since it can only be used to calculate pay for a specified set of data values because the data values are assigned to variables as constants in the program itself. If we needed to calculate the pay with some other employee, we would have to modify the program with new values and recompile and execute the program. For a program to be useful, it should be flexible enough to use any set of data values. In fact, the user should be able to enter a set of data during program execution, and the program should read and use these data values.

2.4 Input: Reading Data

To address the deficiency in our program mentioned above, the next task is to write a program that reads data typed by the user at the keyboard, calculates pay, and prints out the data and the results. In this case, the program must communicate with the user to get the input data.

Task

PAY1: Same as PAY0, except that the data values `id_number`, `hours_worked`, and `rate_of_pay` should be read in from the keyboard.

The algorithm is the same as before except that the data is read rather than set:

```
print title of program;
read the data for id_number, hours_worked, and rate_of_pay;
set pay to the product of hours worked and rate of pay;
print the data and the results;
```

In the implementation of the above algorithm, we must read in data from the keyboard. In a C program, all communication with a user is performed by functions available in the standard library. We have already used `printf()` to write on the screen. Similarly, a function, `scanf()`, is available to read data in from the keyboard and store it in some object. `Printf()` performs the output function and `scanf()` performs the input function.

The function `scanf()` must perform several tasks: read data typed at the keyboard, convert the data to its internal form, and store it into an object. In C, there is no way for any function, including `scanf()`, to directly access a variable by its name defined in another function. Recall that we said the scope of a variable was the block in which it was defined, and it is only within this scope that a variable name is recognized. But if `scanf()` cannot directly access a variable in `main()`, it cannot assign a value to that variable. So how does `scanf()` store data into an object? A function can use the address of an object to indirectly access that object.

Therefore,`scanf()` must be supplied with the address of an object in which a data value is to be stored. In C, the *address of* operator, `&`, can be used to obtain the address of an object. For example, the expression `&x` evaluates to the address of the variable x. To read the id number from the keyboard and store the value into `id_number`, `hours_worked` and `rate_of_pay` we use the statements:

```
scanf("%d", &id_number);
scanf("%f", &hours_worked);
scanf("%f", &rate_of_pay);
```

The first argument of `scanf()` is a format string as it was for `printf()`. The conversion specification, %d, specifies that the input is in decimal integer form. `Scanf()` reads the input, converts it to an internal form, and stores it into an integer object whose address is given by the next unmatched argument. In this case, the value read is stored into the object whose address is `&id_number`, i.e. the value is stored into `id_number`. The remaining two scanf statements work similarly, except the conversion specification is `%f`, to indicate that a floating point number is to be read, converted

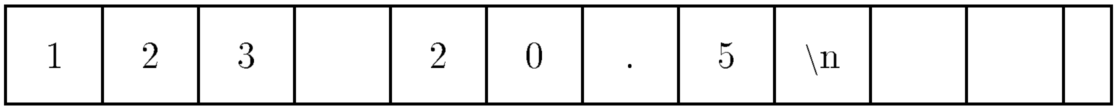

Figure 2.6: Keyboard Buffer

to internal form and stored in the objects whose addresses are `&hours_worked` and `&rate_of_pay` respectively. The type of the object must match the conversion specification, i.e. an integer value must be stored into an `int` type object and a floating point value into a `float object`.

To better understand how `scanf()` works, let us look in a little more detail. As a user types characters at the keyboard they are placed in a block of memory called a **buffer** (most but not all systems buffer their input). The function `scanf()` does not have access to this buffer until it is complete which is indicated when the user types the newline character, i.e. the RETURN key. (see Figure 2.6). The function `scanf()` then begins reading the characters in the buffer one at a time. When `scanf()` reads *numeric input*, it first skips over any leading white space and then reads a sequence of characters that make up a number of the specified type. For example, integers may only have a sign ($+or-$) and the digits 0 to 9. A floating point number may possibly have a decimal point and the `e` or `E` exponent indicators. The function stops reading the input characters when it encounters a character that does not belong to the data type. For example, in Figure 2.6, the first `scanf()` stops reading when it sees the space character after the 3. The data is then converted to an internal form and stored into the object address specified in the argument. Any subsequent `scanf()` performed will begin reading where the last left off in the buffer, in this case at the space. When the newline character has been read, `scanf()` waits until the user types another buffer of data.

At this point we can modify our program by placing the `scanf()` statements in the code replacing the assignments to those variables. However, when we compile and execute the new program, nothing happens; no output is generated and the program just waits. The user does not know when a program is waiting for input unless the program prompts the user to type in the desired items. We use `printf()` statements to print a message to the screen telling the user what to do.

```
    printf("Type ID Number: ");
    scanf("%d", &id_number);
    printf("Hours Worked: ");
    scanf("%f", &hours_worked);
    printf("Hourly Rate: ");
    scanf("%f", &rate_of_pay);
```

The prompts are not necessary to read the data, without them, `scanf()` will read what is typed; but the user will not know when to enter the required data. We can now incorporate these statements into a program that implements the above algorithm shown as the file `pay1.c` in Figure 2.7. When the program is run, here is the sample output:

```
    ***Pay Calculation***
```

```
/*   File: pay1.c
     Programmer: Programmer Name
     Date: Current Date
     This program calculates the pay for one person with the
     hours worked and the rate of pay read in from the keyboard.
*/

main()
{
     /* declarations */
     int id_number;
     float hours_worked,
          rate_of_pay,
          pay;

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

     /* read data into variables */
     printf("Type ID Number: ");
     scanf("%d", &id_number);
     printf("Hours Worked: ");
     scanf("%f", &hours_worked);
     printf("Hourly Rate: ");
     scanf("%f", &rate_of_pay);

     /* calculate results */
     pay = hours_worked * rate_of_pay;

     /* print data and results */
     printf("\nID Number = %d\n", id_number);
     printf("Hours Worked = %f, Rate of Pay = %f\n",
               hours_worked, rate_of_pay);
     printf("Pay = %f\n", pay);
}
```

Figure 2.7: Code for pay1.c

```
Type ID Number:  123
Hours Worked:  20
Hourly Rate:  7.5

ID Number = 123
Hours Worked = 20.000000, Rate of Pay = 7.500000
Pay = 150.000000
```

Everything the user types at the keyboard is also echoed to the screen, and is shown here in *slanted characters.*

We have now seen two ways of storing data into objects: assignment to an object and reading into an object. Assignment stores the value of an expression into an object. Reading into an object involves reading data from the input, converting it to an internal form, and storing it in an object at a specified address.

The function `scanf()` can read several items of data at a time just as `printf()` can print several items of data at a time. For example,

```
scanf("%d %f %f", &id_number, &hours_worked, &rate_of_pay);
```

would read an integer and store it in `id_number`, read a float and store it in `hours_worked`, and read a float and store it in `rate_of_pay`. Of course, the prompt should tell the user to type the three items in the order expected by `scanf()`.

2.5 More C Statements

Our program `pay1.c` is still very simple. It calculates pay in only one way, the product of `hours_worked` and `rate_of_pay`. Our original problem statement in Chapter 1 called for computing overtime pay and for computing the pay for many employees. In this section we will look at additional features of the C language which will allow us to modify our program to meet the specification.

2.5.1 Making Decisions with Branches

Suppose there are different pay scales for regular and overtime work, so there are alternate ways of calculating pay: regular pay and overtime pay. Our next task is to write a program that calculates pay with work over 40 hours paid at 1.5 times the regular rate.

Task

PAY2: Same as PAY1, except that overtime pay is calculated at 1.5 times the normal rate.

For calculating pay in alternate ways, the program must make decisions during execution; so, we wish to incorporate the following steps in our algorithm:

```
if hours_worked is greater than 40.0,
     then calculate pay as the sum of
          excess hours at the overtime rate plus
          40.0 hours at regular rate;
     otherwise, calculate pay at the regular rate.
```

The program needs to make a decision: is `hours_worked` greater than 40.0? If so, execute one computation; otherwise, execute the alternate computation. Each alternate computation is implemented as a different *path* for program control flow to follow, called a **branch**. C provides a feature for implementing this algorithm form as follows:

```
if (hours_worked > 40.0)
     pay = 40.0 * rate_of_pay +
               1.5 * rate_of_pay * (hours_worked - 40.0);
else
     pay = hours_worked * rate_of_pay;
```

The above `if` statement first evaluates the expression within parentheses:

```
hours_worked > 40.0
```

and if the expression is True, i.e. `hours_worked` is greater than 40.0, then the first statement is executed. Otherwise, if the expression is False, the statement following the `else` is executed. After one of the alternate statements is executed, the statement after the `if` statement will be executed. That is, in either case, the program control passes to the statement after the `if` statement.

The general syntax of an `if` statement is:

if (<expression>) <statement> [else <statement>]

The keyword if and the parentheses are required as shown. The two <statement>s shown are often called the **then clause** and the **else clause** respectively. The statements may be any valid C statement including a simple statement, a compound statement (a block), or even an empty statement. The else clause, the keyword else followed by a <statement>, is optional. Omitting this clause is equivalent to having an empty statement in the else clause. An `if` statement can be nested, i.e. either or both branches may also be `if` statements.

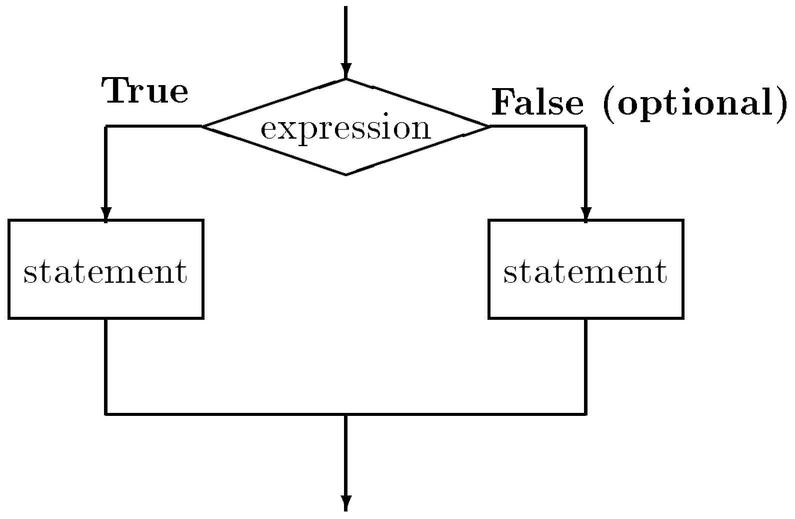

Figure 2.8: If statement control flow

The semantics of the `if` statement are that the expression (also called the **condition**) is evaluated, and the control flow branches to the then clause if the expression evaluates to True, and to the else clause (if any) otherwise. Control then continues with the statement immediately after the `if` statement. This control flow is shown in Figure 2.8.

It should be emphasized that only one of the two alternate branches is executed in an `if` statement. Suppose we wish to check if a number, x, is positive and also check if it is big, say greater than 100. Let us examine the following statement:

```
if (x > 0)
     printf("%d is a positive number\n", x);
else if (x > 100)
     printf("%d is a big number greater than 100\n", x);
```

If `x` is positive, say 200, the first `if` condition is True and the first `printf()` statement is executed. The control does not proceed to the `else` part at all, even though `x` is greater than 100. The else part is executed only if the first `if` condition is False. When two conditions overlap, one must carefully examine how the statement are constructed. Instead of the above, we should write:

```
if (x > 0)
     printf("%d is a positive number\n", x);
if (x > 100)
     printf("%d is a big number greater than 100\n", x);
```

Each of the above is a separate `if` statement. If `x` is positive, the first `printf()` is executed. In either case control then passes to the next `if` statement. If `x` is greater than 100, a message is again printed. Another way of writing this, since (x > 100) is True only when (x > 0), we could write:

```
if (x > 0) {
     printf("%d is a positive number\n", x);
     if (x > 100)
          printf("%d is a big number greater than 100\n", x);
}
```

If (x > 0) is true, the compound statement is executed. It prints a message and executes the if (x > 100) ... statement. Suppose, we also wish to print a message when x is negative. We can add an else clause to the first if statement since positive and negative numbers do not overlap.

```
if (x > 0) {
     printf("%d is a positive number\n", x);
     if (x > 100)
          printf("%d is a big number greater than 100\n", x);
}
else if (x < 0)
     printf("%d is a negative number\n", x);
```

Something for you to think about: is there any condition for which no messages will be printed by the above code?

Returning to our payroll example, suppose we wish to keep track of both regular and overtime pay for each person. We can write the if statement:

```
if (hours_worked > 40.0) {
     regular_pay = 40.0 * rate_of_pay;
     overtime_pay = 1.5 * rate_of_pay * (hours_worked - 40.0);
}
else {
     regular_pay = hours_worked * rate_of_pay;
     overtime_pay = 0.0;
}
pay = regular_pay + overtime_pay;
```

Note: both clauses in this case are compound statements; each block representing a branch is treated as a single unit. Whichever branch is executed, that entire block is executed. If hours_worked exceeds 40.0, the first block is executed; otherwise, the next block is executed. Note, both blocks compute regular and overtime pay so that after the if statement the total pay can be calculated as the sum of regular and overtime pay. Also observe that we have used consistent data types in our expressions to forestall any unexpected problems. Since variables in the expressions are float type, we have used floating point constants 40.0, 1.5, and 0.0.

Operator	Meaning
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
! =	not equal to

Table 2.2: Relational Operators

Relational Operators

The greater than operator, >, used in the above expressions is called a **relational operator**. Other relational operators defined in C, together with their meanings are shown in Table 2.2 Note that for those relational operators having more than one symbol, the order of the symbols must be as specified in the table (>= not =>). Also take particular note that the equality relational operator is ==, NOT =, which is the assignment operator.

A relational operator compares the values of two expressions, one on each side of it. If the two values satisfy the relational operator, the overall expression evaluates to True; otherwise, it evaluates to False. In C, an expression that evaluates to False has the value of zero and an expression that evaluates to True has a non-zero value, typically 1. The reverse also holds; an expression that evaluates to zero is interpreted as False when it appears as a condition and expression that evaluates to non-zero is interpreted as True.

2.5.2 Simple Compiler Directives

In some of the improvements we have made so far to our program for PAY2, we have used numeric constants in the statements themselves. For example, in the code:

```
if (hours_worked > 40.0) {
     regular_pay = 40.0 * rate_of_pay;
     overtime_pay = 1.5 * rate_of_pay * (hours_worked - 40.0);
}
else {
     regular_pay = hours_worked * rate_of_pay;
     overtime_pay = 0.0;
}
pay = regular_pay + overtime_pay;
```

we use the constant `40.0` as the limit on the number of regular pay hours (hours beyond this are considered overtime), and the constant `1.5` as the overtime pay rate (time and a half). Use of numeric constants (sometimes called "magic numbers") in program code is often considered bad style because the practice makes the program logic harder to understand and debug. In addition, the practice makes programs less flexible, since making a change in the values of numeric constants requires that the entire code be reviewed to find all instances where the "magic number" is used.

C, like many other programming languages, allows the use of symbolic names for constants in programs. This facility makes use of the C *preprocessor* and takes the form of **compiler directives**. Compiler directives are not, strictly speaking, part of the source code of a program, but rather are special directions given to the compiler about how to compile the program. The directive we will use here, the `define` directive, has syntax:

```
#define <symbol_name> <substitution_string>
```

All compiler directives, including `define`, require a `#` as the first non-white space character in a line. (Some older compilers require that `#` be in the first column of a line but most modern compilers allow leading white space on a line before `#`). The semantics of this directive is to define a string of characters, <substitution_string>, which is to be substituted for every occurrence of the symbolic name, <symbol_name>, in the code for the remainder of the source file. Keep in mind, a directive is not a statement in C, nor is it terminated by a semi-colon; it is simply additional information given to the compiler.

In our case, we might use the following compiler directives to give names to our numeric constants:

```
#define   REG_LIMIT      40.0
#define   OT_FACTOR      1.5
```

These directives define that wherever the string of characters `REG_LIMIT` occurs in the source file, it is to be replaced by the string of characters `40.0` and that the string `OT_FACTOR` is to be replaced by `1.5`. With these definitions, it is possible for us to use `REG_LIMIT` and `OT_FACTOR` in the program statements instead of numeric constants. Thus our code would become:

```
if (hours_worked > REG_LIMIT) {
     regular_pay = REG_LIMIT * rate_of_pay;
     overtime_pay = OT_FACTOR * rate_of_pay * (hours_worked - REG_LIMIT);
}
else {
     regular_pay = hours_worked * rate_of_pay;
     overtime_pay = 0.0;
}
pay = regular_pay + overtime_pay;
```

The code is now more readable; it says in words exactly what we mean by these statements. Before compilation proper, the preprocessor replaces the symbolic constants with strings that constitute actual constants; the string of characters `40.0` for the string `REG_LIMIT` and `1.5` for `OT_FACTOR` throughout the source program code.

The rules for the symbol names in directives are the same as those for identifiers. A common practice used by many programmers is to use upper case for the symbolic names in order to distinguish them from variable names. Remember, `define` directives result in a literal substitution without any data type checking, or evaluation. It is the responsibility of the programmer to use defines correctly. The source code is compiled after the preprocessor performs the substitutions.

The implementation of the PAY2 algorithm incorporating the above defines and other improvements discussed so far is shown in Figure 2.9. Note in the code, when the hours worked do not exceed `REG_LIMIT`, the overtime pay is set to zero. A constant zero value in a program code is not unreasonable when the logic is clear enough.

Here is a sample session from the resulting executable file:

```
***Pay Calculation***

Type ID Number:  456
Hours Worked:  50
Hourly Rate:  10

ID Number = 456
Hours Worked = 50.000000, Rate of Pay = 10.000000
Regular Pay = 400.000000, Overtime Pay = 150.000000
Total Pay = 550.000000
```

2.5.3 More on Expressions

Expressions used for computation or as conditions can become complex, and considerations must be made concerning how they will be evaluated. In this section we look at three of these considerations: precedence and associativity, the data type used in evaluating the expression, and logical operators.

Precedence and Associativity

Some of the assignment statements in the last section included expressions with more than one operator in them. The question can arise as to how such expressions are evaluated. Whenever there are several operators present in an expression, the order of evaluation depends on the **precedence** and **associativity** (or grouping) of operators as defined in the programming language. If operators have unequal precedence levels, then the operator with higher precedence is evaluated first. If operators have the same precedence level, then the order is determined by their associativity. The order of evaluation according to precedence and associativity may be overridden by using parentheses; expressions in parentheses are always evaluated first.

```
/*   File: pay2.c
     Programmer: Programmer Name
     Date: Current Date
     This program calculates the pay for one person, given the
     hours worked and rate of pay.
*/
#define   REG_LIMIT      40.0
#define   OT_FACTOR      1.5

main()
{    /* declarations */
     int id_number;
     float hours_worked,
          rate_of_pay,
          regular_pay, overtime_pay, total_pay;

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

     /* read data into variables */
     printf("Type ID Number: ");
     scanf("%d", &id_number);
     printf("Hours Worked: ");
     scanf("%f", &hours_worked);
     printf("Hourly Rate: ");
     scanf("%f", &rate_of_pay);

     /* calculate results */
     if (hours_worked > REG_LIMIT) {
          regular_pay = REG_LIMIT * rate_of_pay;
          overtime_pay = OT_FACTOR * rate_of_pay *
                                   (hours_worked - REG_LIMIT);
     }
     else {
          regular_pay = hours_worked * rate_of_pay;
          overtime_pay = 0.0;
     }
     total_pay = regular_pay + overtime_pay;
     /* print data and results */
     printf("\nID Number = %d\n", id_number);
     printf("Hours Worked = %f, Rate of Pay = %f\n",
               hours_worked, rate_of_pay);
     printf("Regular Pay = %f, Overtime Pay = %f\n",
               regular_pay, overtime_pay);
     printf("Total Pay = %f\n", total_pay);
}
```

Figure 2.9: Code for pay2.c

Operator	Associativity	Type
$+$, $-$	right to left	unary arithmetic
$*$, $/$, $\%$	left to right	binary arithmetic
$+$, $-$	left to right	binary arithmetic
$<$, $<=$, $>$, $>=$	left to right	binary relational
$==$, $!=$	left to right	binary relational

Table 2.3: Precedence and Associativity of Operators

Table 2.3 shows the arithmetic and relational operators in precedence level groups separated by horizontal lines. The higher the group in the table, the higher its precedence level. For example, the precedence level of the binary operators `*`, `/`, and `%` is the same but it is higher than that of the binary operator group `+`, `-`. Therefore, the expression

```
x + y * z
```

is evaluated as

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

Associativity is also shown in the table. Left to right associativity means operators with the same precedence are applied in sequence from left to right. Binary operators are grouped from left to right, and unary from right to left. For example, the expression

```
x / y / z
```

is evaluated as

```
(x / y) / z
```

The precedence of the relational operators is lower than that of arithmetic operators, so if we had an expression like

```
x + y >= x - y
```

it would be evaluated as

```
(x + y) >= (x - y)
```

However, we will often include the parentheses in such expressions to make the program more readable.

From our payroll example, consider the assignment expression:

```
        overtime_pay = OT_FACTOR * rate_of_pay * (hours_worked - REG_LIMIT);
```

In this case, the parentheses are required because the product operator, `*`, has a higher precedence than the sum operator. If these parentheses were not there, the expression would be evaluated as:

```
        overtime_pay = (((OT_FACTOR * rate_of_pay) * hours_worked) - REG_LIMIT);
```

where what we intended was:

```
        overtime_pay = ((OT_FACTOR * rate_of_pay) * (hours_worked - REG_LIMIT));
```

That is, the subtraction to be done first, followed by the product operators. There are several product operators in the expression; they are evaluated left to right in accordance with their associativity. Finally, the assignment operator, which has the lowest precedence, is evaluated.

Precise rules for evaluating expressions will be discussed further in Chapter 5 where a complete table of the precedence and associativity of all C operators will be given. Until then, we will point out any relevant rules as we need them and we will frequently use parentheses for clarity.

Data Types in Expressions

Another important consideration in using expressions is the type of the result. When operands of a binary operator are of the same type, the result is of that type. For example, a division operator applied to integer operands results in an integer value. If the operands are of mixed type, they are both converted to the type which has the greater range and the result is of that type; so, if the operands are `int` and `float`, then the result is floating point type. Thus, 8/5 is 1 and 8/5.0 is 1.6. The C language will automatically perform type conversions according to these rules; however, care must be taken to ensure the intent of the arithmetic operation is implemented. Let us look at an example.

Suppose we have a task to find the average for a collection of exam scores. We have already written the code which sums all the the scores into a variable `total_scores` and counted the number of exams in a variable `number_exams`. Since both of these data items are integer values, the variables are declared as type `int`. The average, however is a real number (has a fractional part) so we declared a variable `average` to be of type `float`. So we might write statements:

```
    int total_scores, number_exams;
    float  average;

    ...

    average = total_scores / number_exams;
```

in our program. However, as we saw above, since `total_scores` and `number_exams` are both integers, the division will be done as integer division, discarding any fractional part. C will then automatically convert that result to a floating point number to be assigned to the variable `average`. For example, if `total_scores` is 125 and `number_exams` is 10, the the right hand side evaluates to the integer 12 (the fractional part is truncated) which is then converted to a `float`, 12.0 when it is assigned to `average`. The division has already truncated the fractional part, so our result will always have 0 for the fractional part of `average` which may be in error. We could represent either `total_scores` or `number_exams` as `float` type to force real division, but these quantities

Logical	C
AND	&&
OR	||
NOT	!

Table 2.4: Logical Operator Symbols in C

are more naturally integers. We would like to temporarily convert one or both of these values to a real number, only to perform the division. C provides such a facility, called the **cast operator**. In general, the syntax of the cast operator is:

```
(<type-specifier>) <expression>
```

which converts the value of <expression> to a type indicated by the <type-specifier>. Only the value of the expression is altered, not the type or representation of the variables used in the expression. The average is then computed as:

```
average = (float) total_scores / (float) number_exams;
```

The values of the variables are first both converted to `float` (e.g. 125.0 and 10.0), the division is performed yielding a `float` result (12.5) which is then assigned to `average`. We cast both variables to make the program more understandable. In general, it is good programming practice to cast variables in an expression to be all of the same type. After all, C will do the cast anyway, the cast is simply making the conversion clear in the code.

Logical Operators

It is frequently necessary to make decisions based on a logical combination of True and False values. For example, a company policy may not allow overtime pay for highly paid workers. Suppose only those workers, whose rate of pay is not higher than a maximum allowed value, are paid overtime. We need to write the pay calculation algorithm as follows:

```
if ((hours_worked > REG_LIMIT) AND (rate_of_pay <= MAXRATE))
     calculate regular and overtime pay
else
     calculate regular rate pay only, no overtime.
```

If hours worked exceeds the limit, `REG_LIMIT`, AND rate of pay does not exceed `MAXRATE`, then overtime pay is calculated; otherwise, pay is calculated at the regular rate. Such logical combinations of True and False values can be performed using **logical operators**. There are three generic logical operators: AND, OR, and NOT. Symbols used in C for these logical operators are

e1	e2	e1 && e2	e1 || e2	!e1
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

Table 2.5: Truth Table for Logical Combinations

shown in Table 2.4 Table 2.5 shows logical combinations of True and False values and the resulting values for each of these logical operators. We have used T and F for True and False in the table. From the table we can see that the result of the AND operation is True only when the two expression operands are both True; the OR operation is True when either or both operands are True; and the NOT operation, a unary operator, is True when its operand is False.

We can use the above logical operators to write a pay calculation statement in C as follows:

```
if ((hours_worked > REG_LIMIT) && (rate_of_pay <= MAXRATE)) {
     regular_pay = REG_LIMIT * rate_of_pay;
     overtime_pay = OT_FACTOR * rate_of_pay *
                         (hours_worked - REG_LIMIT);
}
else {
     regular_pay = hours_worked * rate_of_pay;
     overtime_pay = 0;
}
```

(We assume that `MAXRATE` is defined using a define directive). We use parentheses to ensure the order in which expressions are evaluated. The expressions in the innermost parentheses are evaluated first, then the next outer parentheses are evaluated, and so on. If `(hours_worked >` `REG_LIMIT)` is True AND `(rate_of_pay <= MAXRATE)` is True, then the whole if expression is True and pay is calculated using the overtime rate. Otherwise, the expression is False and pay is calculated using regular rate.

In C, an expression is evaluated for True or False only as far as necessary to determine the result. For example, if `(hours_worked > REG_LIMIT)` is False, the rest of the logical AND expression need not be evaluated since whatever its value is, the AND expression will be False.

A logical OR applied to two expressions is True if either expression is True. For example, the above statement can be written in C with a logical OR operator, `||`.

```
if ((hours_worked <= REG_LIMIT) || (rate_of_pay > MAXRATE)) {
     regular_pay = hours_worked * rate_of_pay;
     overtime_pay = 0;
}
```

```
     else {
          regular_pay = REG_LIMIT * rate_of_pay;
          overtime_pay = OT_FACTOR * rate_of_pay *
                              (hours_worked - REG_LIMIT);
     }
```

If either hours worked does not permit overtime OR the rate exceeds `MAXRATE` for overtime, calculate regular rate pay; otherwise, calculate regular and overtime pay. Again, if `(hours_worked <= REG_LIMIT)` is True, the logical OR expression is not evaluated further since the result is already known to be True. Precedence of logical AND and OR operators is lower than that of relational operators so the parentheses in the previous two code fragments are not required; however, we have used them for clarity.

Logical NOT applied to a True expression results in False, and vice versa. We can rewrite the above statement using a logical NOT operator, `!`, as follows:

```
     if ((hours_worked > REG_LIMIT) && !(rate_of_pay > MAXRATE)) {
          regular_pay = REG_LIMIT * rate_of_pay;
          overtime_pay = OT_FACTOR * rate_of_pay *
                              (hours_worked - REG_LIMIT);
     }
     else {
          regular_pay = hours_worked * rate_of_pay;
          overtime_pay = 0;
     }
```

If hours worked exceed `REG_LIMIT`, AND it is NOT True that rate of pay exceeds `MAXRATE`, then calculate overtime pay, etc. The NOT operator is unary and its precedence is higher than binary operators; therefore, the parentheses are required for the NOT expression shown.

2.5.4 A Simple Loop — `while`

Our latest program, `pay2.c`, still calculates pay for only one individual. If we have 50 people on the payroll, we must run the above program separately for each person. For our program to be useful and flexible, we should be able to repeat the same logical process of computation as many times as desired; i.e. it should be possible to write a program that calculates pay for any number of people.

Task

PAY3: Same as PAY2, except that the program reads data, computes pay, and prints the data and the pay for a known number of people.

Let us first see how to repeat the process of reading data, calculating pay, and printing the results a fixed number, say 10, times. To repeatedly execute an identical group of statements, we use what is called a **loop**. To count the number of times we repeat the computation, we use an integer variable, `count`. The logic we wish to implement is:

```
set count to 0
repeat the following as long as count is less than 10
     read data
     calculate pay
     print results
     increase count by 1
```

Initially, we set `count` to zero and we will repeat the process as long as `count` is less than 10. Each time we execute the loop, we increment `count` so that for each value of `count` (0, 1, 2, ..., 9), one set of data is processed. When `count` is 10, i.e. it is NOT less than 10, the repeating or looping is terminated.

The C language provides such a control construct; a `while` statement is used to repeat a statement or a block of statements. The syntax for a `while` statement is:

```
while ( <expression> ) <statement>
```

The keyword while and the parentheses are required as shown. The <expression> is a condition as it was for the `if` statement, and the <statement> may be any statement in C such as an empty statement, a simple statement, or a compound statement (including another `while` statement).

The semantics of the `while` statement is as follows. First, the while expression or condition, <expression>, is evaluated. If True, the <statement> is executed and the <expression> is evaluated again, etc. If at any time the <expression> evaluates to False, the loop is terminated and control passes to the statement after the `while` statement. This control flow for a `while` statement is shown in Figure 2.10.

To use the `while` statement to implement the algorithm above, there are several points to note about loops. The loop variable(s), i.e. variables used in the expression, must be initialized prior to the loop; otherwise, the loop expression is evaluated with unknown (garbage) value(s) for the variable(s). Second, if the loop expression is initially True, the loop variable(s) must be modified within the loop body so that the expression eventually becomes False. Otherwise, the loop will be an infinite loop, i.e. the loop repeats indefinitely. Therefore, a proper loop requires the following steps:

```
initialize loop variable(s)
while ( <expression> ) {
     ...
     update loop variable(s)
}
```

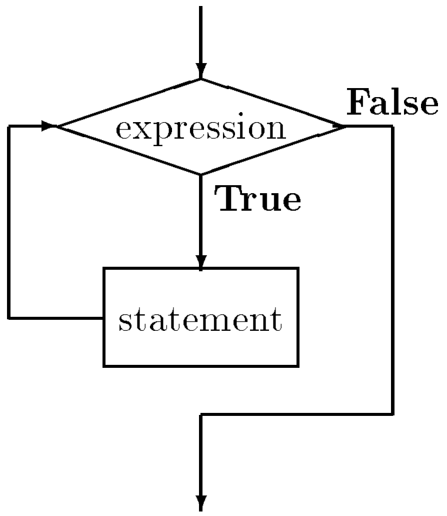

Figure 2.10: Control Flow for `while` statement

Keeping this syntax and semantics in mind, the code for the above algorithm fragment using a `while` loop is shown in Figure 2.11.

First, `count` is initialized to zero and tested for loop termination. The `while` statement will repeat as long as the while expression, i.e. `(count < 10)`, is True. Since `count` is 0, the condition is true, so the body of the loop is executed. The loop body is a block which reads data, calculates pay, prints results, and increases the value of count by one. Except for updating `count`, the statements in the loop body are the same as those in the previous program in Figure 2.9. The count is updated by the assignment statement:

```
count = count + 1;
```

In this statement, the right hand side is evaluated first, i.e. one is added to the current value of `count`, then the new value is then stored back into `count`. Thus, the new value of count is one greater than its previous value. For the first iteration of the loop, `count` is incremented from 0 to 1 and the condition is tested again. Again `(count < 10)` is True, so the loop body is executed again. This process repeats until count becomes 10, `(count < 10)` is False, and the `while` statement is terminated. The program execution continues to the next statement, if any, after the `while` statement.

The above while loop is repeated ten times, once each for `count` = 0, 1, 2, ..., 9. We can also count the number of iterations to be performed as follows:

```
count = 10;
while (count > 0) {
     ...
     count = count - 1;
}
```

```
count = 0;
while (count < 10) {
     /* read data into variables */
     printf("Type ID Number: ");
     scanf("%d", &id_number);
     printf("Hours Worked: ");
     scanf("%f", &hours_worked);
     printf("Hourly Rate: ");
     scanf("%f", &rate_of_pay);

     /* calculate results */
     if (hours_worked > REG_LIMIT) {
          regular_pay = REG_LIMIT * rate_of_pay;
          overtime_pay = OT_FACTOR * rate_of_pay *
                                   (hours_worked - REG_LIMIT);
     }
     else {
          regular_pay = hours_worked * rate_of_pay;
          overtime_pay = 0;
     }
     total_pay = regular_pay + overtime_pay;

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

     /* update the count */
     count = count + 1;
}
```

Figure 2.11: Coding a While Loop

The initial value of count is 10 and the loop executes `while (count > 0)`. Each time the loop is processed, the value of `count` is decremented by one. Eventually, `count` becomes 0, `(count > 0)` is False, and the loop terminates. Again, the loop is executed ten times for values of `count` = 10, 9, 8, ..., 1.

We can easily adapt the second approach to process a loop as many times as desired by the user. We merely ask the user to type in the number of people, and read into `count`. Here is the skeleton code.

```
printf("Number of people: ");
scanf("%d", &count);
while (count > 0) {
     ...
     count = count - 1;
}
```

We use the latter approach to implement the program for our task. The entire program for `pay3.c` is shown in Figure 2.12 A sample session from the execution of this program is shown below.

```
***Pay Calculation***

Number of people:  2

Type ID Number:  123
Hours Worked:  20
Hourly Rate:  7.5

ID Number = 123
Hours Worked = 20.000000, Rate of Pay = 7.500000
Regular Pay = 150.000000, Overtime Pay = 0.000000
Total Pay = 150.000000

Type ID Number:  456
Hours Worked:  50
Hourly Rate:  10

ID Number = 456
Hours Worked = 50.000000, Rate of Pay = 10.000000
Regular Pay = 400.000000, Overtime Pay = 150.000000
Total Pay = 550.000000
```

```
/*   File: pay3.c
     Programmer: Programmer Name
     Date: Current Date
     This program reads in hours worked and rate of pay and calculates
     the pay for a specified number of persons.
*/
#define   REG_LIMIT      40.0
#define   OT_FACTOR      1.5
main()
{
     /* declarations */
     int id_number, count;
     float hours_worked, rate_of_pay,
          regular_pay, overtime_pay, total_pay;


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

     printf("Number of people: ");
     scanf("%d", &count);
     while (count > 0) {
          /* read data into variables */
          printf("\nType ID Number: ");
          scanf("%d", &id_number);
          printf("Hours Worked: ");
          scanf("%f", &hours_worked);
          printf("Hourly Rate: ");
          scanf("%f", &rate_of_pay);

          /* calculate results */
          if (hours_worked > REG_LIMIT) {
               regular_pay = REG_LIMIT * rate_of_pay;
               overtime_pay = OT_FACTOR * rate_of_pay *
                                   (hours_worked - REG_LIMIT);
          }
          else {
               regular_pay = hours_worked * rate_of_pay;
               overtime_pay = 0.0;
          }
          total_pay = regular_pay + overtime_pay;
```

```
        /* print data and results */
        printf("\nID Number = %d\n", id_number);
        printf("Hours Worked = %f, Rate of Pay = %f\n",
                  hours_worked, rate_of_pay);
        printf("Regular Pay = %f, Overtime Pay = %f\n",
                  regular_pay, overtime_pay);
        printf("Total Pay = %f\n", total_pay);

        /* update the count */
        count = count - 1;
    }
}
```

Figure 2.12: Code for pay3.c

2.5.5 Controlling Loop Termination

The program in the last section illustrates one way to control how many times a loop is executed, namely counting the iterations. Rather than build the number of iterations into the program as a constant, `pay3.c` requires the user to type in the number of people for whom pay is to be computed. That technique may be sufficient sometimes, but the user may not be happy if each time a program is used, one has to count tens or hundreds of items. It might be more helpful to let the user signal the end of data input by typing a special value for the data. For example, the user can be asked to type a zero for the id number of the employee to signal the end of data (as long as zero is not an otherwise valid id number). This suggests another refinement to our task:

Task

PAY4: Same as PAY3, except that pay is to be calculated for any number of people. In addition, we wish to keep a count of the number of people, calculate the gross total of all pay disbursed, and compute the average pay. The end of data is signaled by a negative or a zero id number.

Logic for the while loop is quite simple. The loop repeats as long as `id_number` is greater than 0. This will also require us to initialize the `id_number` to some value before the loop starts and to update it within the loop body to ensure loop termination. For our task, we must also keep track of the number of people and the gross pay. After the `while loop`, we must calculate the average pay by dividing gross pay by the number of people. Here is the algorithm logic using the `while` loop construct.

```
    set gross pay and number of people to zero
    prompt user and read the first id number
    while (id number > 0) {
        read remaining data, compute pay, print data
        update number of people
        update gross pay
        prompt user and read next id number
    }
    set average pay to (gross pay / number of people)
```

Values of gross pay and number of people must be kept as cumulative values, i.e. each time pay for a new person is computed, the number of people must be increased by one, and gross pay must be increased by the pay for that person. Cumulative sum variables must be initialized to zero before the loop, similar to our counting variable in the last example; otherwise those variables will contain garbage values which will then be increased each time the loop is processed. Our algorithm is already "code like", and its implementation should be straightforward, but first let us consider the debugging process for the program.

As programs get more complex, manual program tracing becomes tedious; so let's let the program itself generate the trace for us. During program development, we can introduce `printf()` statements in the program to trace the values of key variables during program execution. If there are any bugs in program logic, the program trace will alert us. Such `printf()` statements facilitating the debug process are called **debug statements**. Once the program is debugged, the debug statements can be removed so that only relevant data is output. In our example, we will introduce debug statements to print values of gross pay and number of people.

In the program, we should not only prompt the user to type in an ID number but should also inform him/her that typing zero will terminate the data input. (Always assume that users do not know how to use a program). Prompts should be clear and helpful so a user can use a program without any special knowledge about the program. Figure 2.13 shows the program that implements the above algorithm.

Much of the code is similar to our previous program. We have introduced two additional variables, `number`, an integer counting the number of employees processed, and `gross`, a `float` to hold the cumulative sum of gross pay. Before the `while` loop, these variables are initialized to zero; otherwise only garbage values will be updated. Each time the loop body is executed, these values are updated: `number` by one, and `gross` by the new value of `total_pay`.

A debug statement in the `while` loop prints the updated values of `gross` and `number` each time the loop is executed. The output will begin with the word `debug` just to inform us that this is a debug line and will be removed in the final version of the program. Enough information should be given in debug lines to identify what is being printed. (A debug print out of line after line of only numbers isn't very useful for debugging). The values can alert us to possible bugs and to probable causes. For example, if we did not initialize `gross` to zero before the loop, the first iteration will print a garbage value for `gross`. It would instantly indicate to us that `gross` is probably not initialized to zero. We have also not indented the debug `printf()` statement to make it stand out in the source code.

Once the `while` loop terminates, the average pay must be computed as a ratio of `gross` and `number`. We have added another declaration at the beginning of the block for `average` and the

```
/*   File: pay4.c
     Programmer: Programmer Name
     Date: Current Date
     This program reads in hours worked and rate of pay and calculates
     the pay for several persons. The program also computes the gross pay
     disbursed, number of people, and average pay. The end of data is
     signaled by a negative or a zero id number.
*/
#define   REG_LIMIT      40.0
#define   OT_FACTOR      1.5
main()
{
     /* declarations */
     int id_number, number;
     float hours_worked, rate_of_pay,
          regular_pay, overtime_pay, total_pay,
          gross, average;

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

     /* initialize cumulative sum variables */
     number = 0;
     gross = 0;
     /* initialize loop variables */
     printf("Type ID Number, 0 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 results */
          if (hours_worked > REG_LIMIT) {
               regular_pay = REG_LIMIT * rate_of_pay;
               overtime_pay = OT_FACTOR * rate_of_pay *
                                    (hours_worked - REG_LIMIT);
          }
          else {
               regular_pay = hours_worked * rate_of_pay;
               overtime_pay = 0;
          }
          total_pay = regular_pay + overtime_pay;

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

          /* update cumulative sums */
          number = number + 1;
          gross = gross + total_pay;
          /* debug statements, print variable values */
printf("\ndebug: gross = %f, number = %d\n", gross, number);
          /* update loop variables */
          printf("\nType ID Number, 0 to quit: ");
          scanf("%d", &id_number);
     }
     if (number > 0) {
          average = gross / (float) number;
          printf("\n***Summary of Payroll***\n");
          printf("Number of people = %d, Gross Disbursements = $%f\n",
                    number, gross);
          printf("Average pay = $%f\n", average);
     }
}
```

Figure 2.13: Code for pay4.c

appropriate assignment statement to compute the average at the end. Note we have used the cast operator to cast `number` to a `float` for the division. This is not strictly necessary; the compiler will do this automatically; however, it is good practice to cast operands to like type in expressions so that we are aware of the conversion being done.

It is possible that no data was entered at all, i.e. the user enters 0 as the first id, in which case `number` is zero. If we try to divide `gross` by `number`, we will have a "divide by zero" run time error. Therefore, we check that `number` is greater than zero and only calculate the average and print the result when employee data has been entered.

With all of these changes made as shown in Figure 2.13, the program is compiled, and run resulting in the following sample session:

```
***Pay Calculation***
```

```
Type ID Number, 0 to quit:  123
Hours Worked:  20
Hourly Rate:  7.5

ID Number = 123
Hours Worked = 20.000000, Rate of Pay = $7.500000
Regular Pay = $150.000000, Overtime Pay = $0.000000
Total Pay = $150.000000

debug:  gross = 150.000000, number = 1

Type ID Number, 0 to quit:  456
Hours Worked:  50
Hourly Rate:  10

ID Number = 456
Hours Worked = 50.000000, Rate of Pay = $10.000000
Regular Pay = $400.000000, Overtime Pay = $150.000000
Total Pay = $550.000000

debug:  gross = 700.000000, number = 2

Type ID Number, 0 to quit:  0

***Summary of Payroll***
Number of people = 2, Gross Disbursements = $700.000000
Average pay = $350.000000
```

The debug lines show the changes in `gross` and `number` each time the loop is executed. The first such line shows the value of `gross` the same as that of the total pay and the value of `number` as 1. The next pass through the loop shows the variables are updated properly. The program appears to be working properly; nevertheless, it should be thoroughly tested with a variety of data input. Once the program is deemed satisfactory, the debug statements should be removed from the source code and the program recompiled.

2.5.6 More Complex Loop Constructs — Nested Loops

As we mentioned above, the <`statement`> that is the body of the loop can be any valid C statement and very often it is a compound statement. This includes a `while` statement, or a `while` statement with the block. Such a situation is called a **nested loop**. Nested loops frequently occur when several items in a sequence are to be tested for some property, and this testing itself requires repeated testing with several other items in sequence. To illustrate such a process, consider the following task:

Task

Find all prime numbers less than some maximum value.

The problem statement here is very simple; however, the algorithm may not be immediately obvious. We must first understand the problem.

A prime number is a natural number, i.e. 1, 2, 3, 4, etc., that is not exactly divisible by any other natural number, except 1 and itself. The number 1 is a prime by the above definition. The algorithm must find the other primes up to some maximum. One way to perform this task is to use a process called **generate and test**. In our algorithm, we will *generate* all positive integers in the range from 2 to a maximum (constant) value `PRIME_LIM`. Each generated integer becomes a candidate for a prime number and must be *test*ed to see if it is indeed prime. The test proceeds as follows: divide the candidate by every integer in sequence from 2 up to, but not including itself. If the candidate is not divisible by any of the integers, it is a prime number; otherwise it is not.

The above approach involves two phases: one generates candidates and the other tests each candidate for a particular property. The generate phase suggests a loop, each iteration of which performs the test phase, which is also a loop; thus we have a nested loop. Here is the algorithm.

```
set the candidate to 2
while (candidate < PRIME_LIM) {
     test the candidate for prime property
     print the result if a prime number
     generate the next candidate
}
```

In testing for the prime property, we will first assume that the candidate is prime. We will then divide the candidate by integers in sequence. If it is divisible by any of the integers excluding itself, then the candidate is not prime and we may generate the next candidate. Otherwise, we print the number as prime and generate the next candidate.

We need to keep track of the state of a candidate: it is prime or it is not prime. We can use a variable, let's call it `prime` which will hold one of two values indicating True or False Such a state variable is often called a **flag**. For each candidate, `prime` will be initially set to True. If the candidate is found to be divisible by one of the test integers, `prime` will be changed to False. When testing is terminated, if `prime` is still True, then the candidate is indeed a prime number and can be printed. This testing process can be written in the following algorithm:

```
set prime flag to True to assume candidate is a prime
set test divisor to 2
while (test divisor < candidate) {
     if remainder of (candidate/test divisor) == 0
          candidate is not prime
     else get the next test divisor in sequence
}
```

We will use the modulus (mod) operator, `%` described earlier, to determine the remainder of `(candidate / divisor)`. Here is the code fragment for the above algorithm:

```
prime = TRUE;
divisor = 2;
while (divisor < candidate)  {
     if ((candidate % divisor) == 0)
          prime = FALSE;
     else
          divisor = divisor + 1;
}
```

where `TRUE` and `FALSE` are symbolic constants defined using the `define` compiler directive. The complete program is shown in Figure 2.14.

The program follows the algorithm step by step. We have defined symbols `TRUE` and `FALSE` to be 1 and 0, respectively. The final `if` statement uses the expression `(prime)` instead of `(prime == TRUE)`; the result is the same. The expression `(prime)` is True (non-zero) if `prime` is `TRUE`, and False (zero) if `prime` is `FALSE`. Of course, we could have written the if expression as `(prime == TRUE)`, but it is clear, and maybe more readable, as written.

We have included a debug statement in the inner loop to display the values of `candidate`, `divisor`, and `prime`. Once the we are satisfied that the program works correctly, the debug statement can be removed.

Here is a sample session with the debug statement and `PRIME_LIM` set to 8:

```
***Prime Numbers Less than 8***

1 is a prime number
2 is a prime number
debug: candidate = 3, divisor = 2 prime = 1
3 is a prime number
debug: candidate = 4, divisor = 2 prime = 1
debug: candidate = 4, divisor = 3 prime = 0
debug: candidate = 5, divisor = 2 prime = 1
debug: candidate = 5, divisor = 3 prime = 1
debug: candidate = 5, divisor = 4 prime = 1
5 is a prime number
debug: candidate = 6, divisor = 2 prime = 1
debug: candidate = 6, divisor = 3 prime = 0
debug: candidate = 6, divisor = 4 prime = 0
debug: candidate = 6, divisor = 5 prime = 0
debug: candidate = 7, divisor = 2 prime = 1
debug: candidate = 7, divisor = 3 prime = 1
debug: candidate = 7, divisor = 4 prime = 1
```

```
/*   File: prime.c
     Programmer: Programmer Name
     Date: Current Date
     This program finds all prime numbers less than PRIME_LIM.
*/
#define   PRIME_LIM      20
#define   TRUE           1
#define   FALSE          0

main()
{  int candidate, divisor, prime;

     printf("***Prime Numbers Less than %d***\n\n", PRIME_LIM);
     printf("%d is a prime number\n", 1);    /* print 1 */
     candidate = 2;                     /* start at candidate == 2 */

     while (candidate < PRIME_LIM) {    /* stop at candidate == 20 */
          prime = TRUE;                 /* for candidate, set prime to True */
          divisor = 2;                  /* initialize divisor to 2 */

          /* stop when divisor == candidate */
          while (divisor < candidate) {
printf("debug: candidate = %d, divisor = %d prime = %d\n",
          candidate, divisor,prime);

               /* if candidate is divisible by divisor, */
               /* candidate is not prime, set prime to False */
               if (candidate % divisor == 0)
                    prime = FALSE;
               divisor = divisor + 1;   /* update divisor */
          }
          if (prime)                         /* if prime is set to True, */
                                             /* print candidate. */
               printf("%d is a prime number\n", candidate);
          candidate = candidate + 1;         /* update candidate */
     }
}
```

Figure 2.14: Code for prime.c

```
debug: candidate = 7, divisor = 5 prime = 1
debug: candidate = 7, divisor = 6 prime = 1
7 is a prime number
```

We have shown part of a sample session with debug printing included. Notice, that the values printed for `prime` are 1 or 0; remember, `TRUE` and `FALSE` are symbolic names for 1 and 0 used in the source code program only. In this output the nested loops are shown to work correctly. For example, for candidate 5, divisor starts at 2 and progresses to 4; the loop terminates and the candidate is a prime number. A sample session without the debug statement is shown below.

```
***Prime Numbers Less than 20***

1 is a prime number
2 is a prime number
3 is a prime number
5 is a prime number
7 is a prime number
11 is a prime number
13 is a prime number
17 is a prime number
19 is a prime number
```

In looking at the debug output, you might see that the loop that tests for the prime property of a candidate is not an efficient one. For example, when `candidate` is 6, we know that it is not prime immediately after divisor 2 is tested. We could terminate the test loop as soon as `prime` becomes false (if it ever does). In addition, it turns out that a candidate needs to be tested for an even more limited range of divisors. The range of divisors need not exceed the square root of the candidate. (See Problem 6 at the end of the chapter).

2.6 Common Errors

In this section we list some common problems and programming errors that beginners often make. We also suggest steps to avoid these pitfalls.

1. Program logic is incorrect. This could be due to an incorrect understanding of the problem statement or improper algorithm design. To check what the program is doing, manually trace the program and use debug statements. Introduce enough debug statements to narrow down the code in which there is an error. Once an error is localized to a critical point in the code or perhaps to one or two statements, it is easier to find the error. Critical points in the code include before a loop starts, at the start of a loop, at the end of a loop and so forth.

2. Variables are used before they are initialized. This often results in garbage values occurring in the output of results. For example:

```
int x, y;

x = x * y;
```

There is no compiler error, `x` and `y` have unknown, garbage values. Be sure to initialize all variables.

3. The assignment operator, `=`, is used when an "equal to" operator, `==`, is meant, e.g.:

   ```
   while (x = y)
        ...

   if (x = y)
        printf("x is equal to y\n");
   ```

   There will be no compiler error since any valid expression is allowed as an `if` or `while` condition. The expression is True if non-zero is assigned, and False if zero is assigned. Always double check conditions to see that a correct equality operator, `==`, is used.

4. Object names are passed, instead of addresses of objects, in function calls to `scanf()`:

   ```
   scanf("%d", n);          /* should be &n */
   ```

   Again this is not a compile time error; the compiler will assume the value of `n` is the address of an integer object and will attempt to store a value in it. This often results in a run time addressing error. Make sure the passed arguments in `scanf()` calls are addresses of the objects where data is to be stored.

5. Loop variables are not initialized:

   ```
   while (i < n)
        ...
   ```

   `i` is garbage; the `while` expression is evaluated with unknown results.

6. Loop variables are not updated:

   ```
   i = 0;
   while (i < n) {
        ...
   }
   ```

   `i` is unchanged within the loop; it is always 0. The result is an infinite loop.

7. Loop conditions are in error. Suppose, a loop is to be executed ten times:

```
n = 10;
i = 0;
while (i <= n) {
     ...
     i = i + 1;
}
```

`(i <= n)` will be True for `i` = 0, 1, ..., 10, i.e. 11 times. The loop is executed one more time than required. Loop expressions should be examined for values of loop variables at the boundaries. Suppose n is zero; should the loop be executed? Suppose it is 1, suppose it is 10, etc.

8. User types in numbers incorrectly. This will be explained more fully in Chapter 4. Consider the loop:

```
while (x != 0) {
     ...
     scanf("%d", &x);
}
```

Suppose a user types: *23r*. An integer is read by `scanf()` until a non-digit is reached, in this case, until *r* is reached. The first integer read will be 23. However, the next time `scanf()` is executed it will be unable to read an integer since the first non-white space character is a non-digit. The loop will be an infinite loop.

9. Expressions should use consistent data types. If necessary, use a cast operator to convert one data type to another.

```
int sum, count;
float avg;

avg = sum / count;
```

Suppose `sum` is 30 and count is 7. The operation `sum / count` will be the integer value of 30 / 7, i.e. 4; the fractional part is truncated. The result 4 is assigned to a `float` variable `avg` as 4.0. If a floating point value is desired for the ratio of `sum / count`, then cast the integers to `float`:

```
avg = (float) sum / (float) count;
```

Now, the expression evaluates to 30.0 / 7.0 whose result is a floating point value 4.285 assigned to `avg`

2.7 Summary

In this chapter we have begun looking at the process of designing programs. We have stressed the importance of a correct understanding of the problem statement, and careful development of the algorithm to solve the problem. This is probably the most important, and sometimes the most difficult part of programming.

We have also begun introducing the *syntax* and *semantics* of the C language. We have seen how to define the special function, `main()` by specifying the *function header* followed by the *function body*, a collection of statements surrounded by brackets, { and }. The function body begins with variable declarations to allocate storage space and assign names to the locations, followed by the executable statements. Variable declarations take the form:

<type_specifier> <identifier>[, <identifier>...];

where <type_spec> may be either `int` or `float` for integers or floating point variables, respectively. (We will see other type specifiers in later chapters). We gave rules for valid <identifier>s used as variable names.

We have discussed several forms for executable statements in the language. The simplest statement is the assignment statement:

<Lvalue>=<expression>;

where (for now) <Lvalue> is a variable name and <expression> consists of constants, variable names and operators. We have presented some of the operators available for arithmetic computations and given rules for how expressions are evaluated. The assignment statement evaluates the expression on the right hand side of the operator `=` and stores the result in the object referenced by the <Lvalue>. We pointed out the importance of variable type in expressions and showed the cast operator for specifying type conversions within them.

(<type-specifier>) <expression>

We also described how the library function `printf()` can be used to generate output from the program, as well as how information may be read by the program at run time using the `scanf()` function.

We next discussed two program control constructs of the language: the `if` and `while` statements. The syntax for `if` statements is:

if (<expression>) <statement> [else <statement>]

where the <expression> is evaluated and if the result is True (non-zero) then the first <statement> (the "then" clause) is executed; otherwise, the <statement> after the keyword else (the "else" clause) is executed. For a `while` statement, the syntax is:

```
while ( <expression> ) <statement>
```

where the <expression> is evaluated, and as long as it evaluates to True, the <statement> is repeatedly executed.

In addition we discussed one of the simple compiler directives:

```
#define <symbol_name> <substitution_string>
```

which can be used to define symbolic names to character strings within the source code; used here for defining constants in the program.

With these basic tools of the language you should be able to begin developing your own programs to compile, debug and execute. Some suggestions are provided in the Problems Section below. In the next chapter, we will once again concentrate on the proper methods of designing programs, and in particular modular design with user defined functions.

2.8 Exercises

Given the following variables and their initializations:

```
int a, x, y, z;
float b, u, v, w;

x = 10; y= 20; z = 30;
u = 4.0; v = 10.0;
```

What are the values of the expressions in each of the following problems:

1.
```
(a)  a = x - y - z;
(b)  a = x + y * z;
(c)  a = z / y + y;
(d)  a = x / y / z;
(e)  a = x % y % z
```

2.
```
(a)  a = (int) (u / v);
(b)  a = (int) (v / u);
(c)  b = v - u;
(d)  b = v / u / w;
```

3. What are the results of the following mod operations:

```
(a)   5  %  3
(b)  -5  %  3
(c)   5  % -3
(d)  -5  % -3
\item
\begin{verbatim}
(a)  (x <= y && x >= z)
(b)  (x <= y || x >= z)
(c)  (x <= y && !(x >= z))
(d)  (x = y && z > y)
(e)  (x == y && z > y)
```

4. Under what conditions are the following expressions True?

```
(a)  (x = y && y = z)
(b)  (x == y && y == z)
(c)  (x == y || y == z)
(d)  (x >= y && x <= z)
(e)  (x > y && x < z)
```

5. Make required corrections in the following code.

   (a)

   ```
   main()
   {    int n;

        scanf("%d", n);
   }
   ```

   (b)

   ```
   main()
   {    float n;

        printf("%d", n);
   }
   ```

   (c)

   ```
   main()
   {    int n1, n2;

        if (n1 = n2)
             printf("Equal\n");
        else
             printf("Not equal\n");
   }
   ```

6. Find and correct errors in the following program that is supposed to read ten numbers and print them.

   ```
   main()
   {    int n, count;

        scanf("%d", &n);
        while (count < 10) {
             printf("%d\n", n);
             scanf("%d", &n);
        }
   }
   ```

7. We wish to print integers from 1 through 10. Check if the following loop will do so correctly.

   ```
   i = 1;
   while (i < 10) {
        printf("%d\n", i);
        i =  i + 1;
   }
   ```

8. Suppose a library fine for late books is: 10 cents for the first day, 15 cents per day thereafter. Assume that the number of late days is assigned to a variable `late_days`. Check if the following will compute the fine correctly.

```
if (late_days == 1)
     fine = 0.10;
else
     fine = late_days * 0.15;
```

2.9 Problems

1. Write a program that reads three variables `x`, `y`, and `z`. The program should check if all three are equal, or if two of the three are equal, or if none are equal. Print the result of the tests. Show the program with manual trace.

2. Velocity of an object traveling at a constant speed can be expressed in terms of distance traveled in a given time, If distance, s, is in feet and time, t, is in seconds, the velocity in feet per second is:

   $$v = d/t$$

   Write a program to read distance traveled and time taken, and calculate the velocity for a variety of input values until distance traveled is zero. Print the results for each case. Show a manual trace.

3. Acceleration of an object due to gravity, g, is 32 feet per second per second. The velocity of a falling body starting from rest at time, t, is given by:

   $$v = g * t$$

   The distance traveled in time, t, by a falling body starting from rest is given by:

   $$d = g * t * t/2$$

   Write a program that repeatedly reads experimental values of time taken by a body to hit the ground from various heights. The program calculates for each case: the height of the body and the velocity of the body when it hits the ground.

4. Write a program that reads a set of integers until a zero is entered. Excluding zero, the program should print a count of and a sum of:

   (a) positive numbers
   (b) negative numbers
   (c) even numbers
   (d) odd numbers
   (e) positive even numbers
   (f) negative odd numbers.
   (g) all numbers

   Use debug statements to show cumulative sums as each new number is read and processed.

5. We wish to convert miles to kilometers, and vice versa. Use the loose definition that a kilometer is 5.0 / 8.0 of a mile. Write a program that generates two tables: a table for kilometer equivalents to miles for miles 1 through 10, and a table for mile equivalents of kilometers for kilometers from 1 to 20.

6. Improve the program `prime.c` of Section 2.5.6 in the following ways:

(a) Terminate the inner loop as soon as it is detected that the number is not prime.

(b) Test each candidate only while `(divisor * divisor <= candidate)`.

(c) Test only candidates that are odd numbers greater than 3.

For each of these improvements, how many times is the inner loop executed when `PRIME_LIM` is 20? How does that compare to our original program?

7. Write a program to generate Fibonacci numbers less than 100. Fibonacci numbers are 1, 1, 2, 3, 5, 8, 13, 21, etc. The first two Fibonacci numbers are 1 and 1. All other numbers follow the pattern: a Fibonacci number is the sum of previous two Fibonacci numbers in the sequence. In words, the algorithm for this problem is as follows:

   We will use two variables, `prev1` and `prev2`, such that `prev1` is the last fibonacci number and `prev2` is the one before the last. Print the first two fibonacci numbers, 1 and 1; and initialize `prev1` and `prev2` as 1 and 1. The new `fib_number` is the sum of the two previous numbers, `prev1` and `prev2`; the new `fib_number` is now the last fibonacci number and `prev1` is the one before the last. So, save `prev1` in `prev2` and save `fib_number` in `prev1`. Repeat the process while `fib_number` is less than 100.

8. (Optional) Write a program to determine the largest positive integer that can be stored in an `int` type variable. An algorithm to do this is as follows:

   Initialize a variable to 1. Multiply by 2 and add 1 to the variable repeatedly until a negative value appears in the variable. The value of the variable just before it turned negative is the largest positive value.

   The above follows from the fact that multiplying by 2 shifts the binary form to the left by one position. Adding one to the result makes all ones in the less significant part and all zeros in the more significant part. Eventually a 1 appears in the leading sign bit, i.e. a negative number appears. The result just before that happens is the one with all ones except for the sign bit which is 0. This is the largest positive value.

9. (Optional) Write a program to determine the negative number with the largest absolute value.

10. Write a program that reads data for a number of students and computes and prints their GPR. For each student, an id number and transcript data for a number of courses is read. Transcript data for each course consists of a course number (range 100-900), number of credits (range 1-6), and grade (range 0-4). The GPR is the ratio of number of total grade points for all courses and the total number of credits for all courses. The number of grade points for one course is the product of grade and credits for the course. The end of transcript data is signaled by a zero for the course number; the end of student data is signaled by a zero id number.