CHAPTER 2

The Art of Programming

In Chapter 1 we introduced the concept of an "instruction" and a "sequences of instructions", which we call a program. Most of the programming today is done in high-level programming languages languages (e.g., C, C#, C++, Java, FORTRAN). Although these languages differ in terms of notation and features they support, they nevertheless share many properties in common.

No mater which language you use, the design of a program usually consists of three major steps: (a) developing an algorithm, (b) deciding on the data structures, and (c) the writing the actual code.

CHAPTER OBJECTIVES

- To describe the concept of an algorithm and how to develop one.
- To provide coverage of basic computer programming.

2.1 Algorithms

An **algorithm** is a finite set of instructions for accomplishing some specific task. (The word is derived from the name of the Persian mathematician Al-Khwarizmi). The instructions need not be computer-based instructions, and an algorithm need not be implemented as a computer program. Rather, it is a formal way of specifying a solution to a specific problem one wishes to solve.

To illustrate, suppose you want to sort a set of N ($N \geq 2$) integers. Let *orig* be the name of the original set to be sorted and let *final* be the name of the new set consisting of the sorted integers. One way of sorting the set as a sequence of increasing integer values is to define the following algorithm (Algorithm 1), which consists of the following set of instructions:

1. Initialize the *orig* set.
2. Copy the first integer in the *orig* set and place it on the side.

3. Compare the “next” integer in the *orig* set with the integer on the side. If the “next” integer is smaller than the integer you placed on the side exchange the “next” integer with the one on the side.
4. If this is not the last integer in the *orig* set go back to step (3).
5. If this is the last integer in the “current” *orig* set then the integer on the side is the smallest integer in the “current” *orig* set. (we use the term “current” since the size of the set changes during the execution of the algorithm; this will become clear shortly).
    a. Place this smallest integer from the side into the the *final* set as the next element in the *final* set.
    b. Discard the first integer (the one that was copied at step of 2) from the “current” *orig* set (Note that the *orig* set has now shrunk and has 1 less element).
6. If the *orig* set consists now of more than 1 integer go back to step (2).
7. The *orig* set has now only 1 integer. Place this last integer (from the *orig* set) into the the *final* set, as the last element in that set.

We encourage the reader to create a set of 10 paper cards, each consisting of an arbitrary integer, and to “execute” the above algorithm, to ascertain that this algorithm indeed does the sorting correctly.

The overall logical structure of an algorithm can be expressed graphically by a *flow chart*, which is built up from the following components:

- **Rectangles**, which represent simple action steps.
- **Diamonds**, which represent decision branching steps.
- **Arrowed lines**, which represent iteration steps and link some of the steps (rectangles and diamonds) to some other steps.

Figure 2.1 depicts the flow chart corresponding to the sorting algorithm above.

The above example illustrates several important properties of what an algorithm is:

- An algorithm always starts with an initial state and “hopefully” terminates in a well-defined final state.
- An algorithm consists of a finite set of steps. Some of these steps are quite simple (e.g., exchange a set of integers), some of these steps are decision-based (e.g., if this is not the last element in the *orig* list ...), and some of these steps are iteration-based (e.g., repeat a set of steps until the algorithm completes).
- An algorithm usually has some data it operates on (e.g., the *orig* and *final* lists, and the “integer on the side”).
- An algorithm usually has some input (e.g., the *orig* list) and some output (e.g., the *final* list).

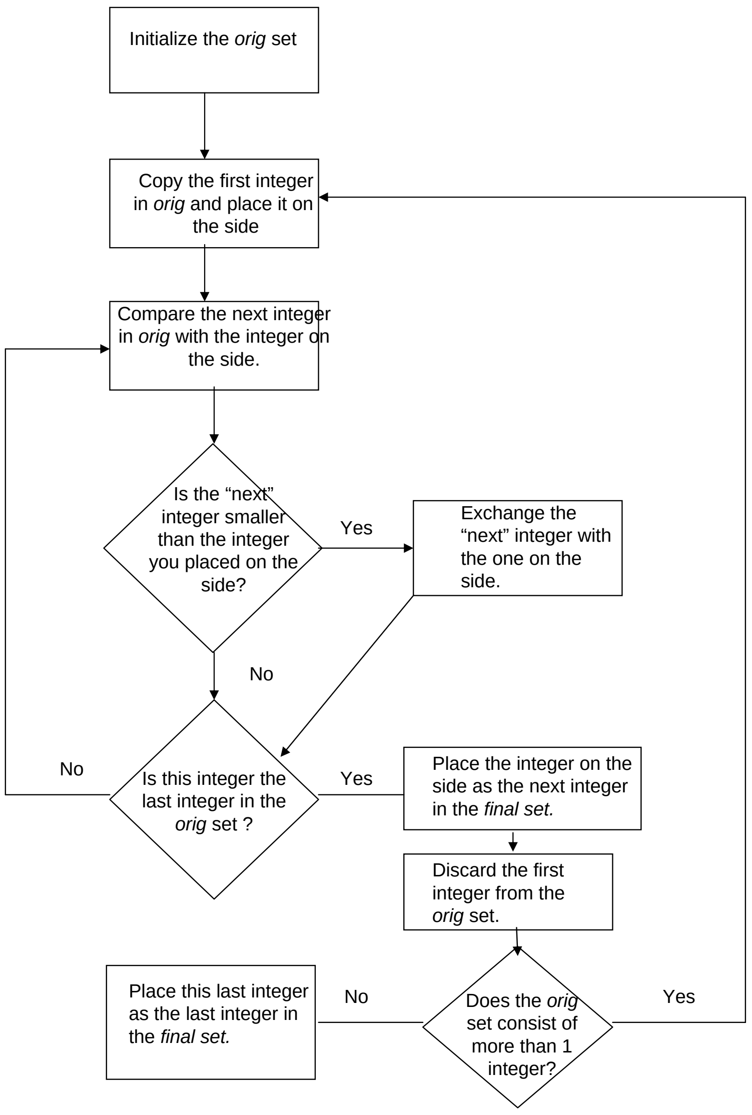
Initialize the *orig* set
Copy the first integer in *orig* and place it on the side
Compare the next integer in *orig* with the integer on the side.
Is the "next" integer smaller than the integer you placed on the side?
Yes
Exchange the "next" integer with the one on the side.
No
Is this integer the last integer in the *orig* set ?
No
Yes
Place the integer on the side as the next integer in the *final set.*
Discard the first integer from the *orig* set.
Does the *orig* set consist of more than 1 integer?
No
Yes
Place this last integer as the last integer in the *final set.*

Figure 2.1 Algorithm 1 to sort a set of integers.

Writing an algorithm to solve a particular problem does not ensure that when executing the algorithm the problem will be indeed solved. Indeed, one may specify an incorrect algorithm (e.g, if one wrote in step 4 "go to step (2)" instead of "go to step (3)", the algorithm will fail to sort the list correctly). An algorithm is **correct** only if its execution will always produce the correct result. As you will see, proving that an algorithm is correct is sometimes a very challenging task.

As stated above, an algorithm may not always terminate; this may be intentional (when we compute a non-terminating function like listing the entire sequence of Π), or be the result of specifying an incorrect algorithm (e.g, if one wrote in step 4 "go to step (4)" instead of "go to step (3)").

There are many different correct algorithms for solving the exact same problem. For example, we could have written the algorithm for sorting a list of N integers, by searching in each iteration (or at each step) for the largest integer first (rather then searching for the smallest integer). The end result would be the same (if we wrote the algorithm correctly). As we shall see (Section 2.7.2), not all algorithms for solving the same exact problem are created equal. Some may take less time to complete than others, may require less memory space than others, etc. Hence, it is not always sufficient to devise a correct algorithm; one may want also to make sure that it executes efficiently. Again, this is a major topic that we will consider in later chapters.

2.2 Data

Integral to computer programming is the presence of data. As we shall see throughout this text, there is a rich body of knowledge concerning data structure and organization. Indeed, most undergraduate programs offer a dedicated course on this subject alone. In this chapter we will provide a brief gentle introduction to to this topic. In Part **??** of the text we devote two chapters to this important and fascinating subject.

Each data element in a computer system, in its simplest form, is a sequence of bits that represents some information. In this chapter we will focus our attention on only integer-type data items (usually represented by a 32-bit word).

The data items in a computer program are stored in **variables**. A variable is a "datum container" that at any point in time has some single value. The value of a variable can change over time. A variable can only contain values of the same type, which for the purpose of this chapter are only integers. Each variable has a unique name (e.g., length, age). Two variables may contain the same value; that is, the "length" and "age" variable may each contain the value "28".

You can think of a variable as a named "box" that can contain only one integer at a time. That is, if you have a box named "length" and two integers, say "6" and "28", the two integers cannot be stored at the same time in the same box "length". If you want to store both integers at the same time, you will need to have another box available.

The value stored in a variable can change over time. For example, a variable "length" may currently have the value "10" and a minute later have the value "30". We use the phrase "variable X is assigned the value Y" to denote the fact that we have changed the value of variable X to the new value Y.

To recapture. You can think of the data in your computer program as being stored in a collection of boxes (variables). Each box has a unique name. Each box can hold only one datum value at a time, but the value in a box can change over time. You can have in your program as many boxes as you wish!

It is sometimes convenient to have a collection of variables be grouped under one unique name. For this purpose, most programming languages provide an **array** type data structure. An array, in it simplest form, is a one-dimensional table consisting of a fixed number of entries of the same type,

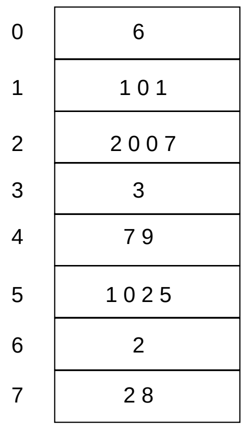

Figure 2.2 Basic structure of an array.

which for the purpose of this chapter are only integers. An array has a name, say A and size, say N. An entry in an array is identified by its position, using an index, which usually ranges between "0" and "N − 1".

You can think of an array as a stack of boxes. The array (stack) has a single unique name, say "drawers". If the array "drawers" consists of 6 entries (boxes), then the first entry (box) in the array can be referred to by the name "drawers[0]", the second one by the name "drawers[1]", etc.

To illustrate, let A be an array of size "8" (see Figure 2.2). Thus, the index ranges between 0 to 7. The 6^{th} entry is identified by A[5], and in Figure 2.2 has the value "1025". Please remember that the index starts at "0" and therefore the sixth entry is A[5] not A[6].

2.3 Prelude To Computer Programming

Looking back at Algorithm 1 depicted in Figure 2.1, you will notice that the description is a bit vague. It is basically given in English, with very little details of the type of data structures and instructions one needs to employ. It is preferable to have a tighter, more specific algorithm that is "closer" to a computer program. In this section we provide the basic building blocks for allowing one to transform a "high-level English like" description of an algorithm to a "low-level mathematical like" description of an equivalent algorithm.

2.3.1 Basic Statements

There are three basic "high-level" statements that can operate on some arbitrary data items (variables) `A` and `B`.

- **assignment**, which takes the form:
  - `A` $\leftarrow$ `B`

The assignment statement assigns the value of variable B to variable A. That is, the assignment replaces the value of A with the one of B. Thus, after the assignment take place the value of variables A and B are the same. Note that instead of variable B one can use a constant (e.g., $A \leftarrow 25$), which assigns the value "25" to variable A.

- **arithmetic**, which take one of the following four forms:
  - $A + B$, the addition operation.
  - $A - B$, the subtraction operation.
  - $A * B$, the multiplication operation.
  - $A \div B$, the division operation.

  The arithmetic statements simply take the values of variables A and B and compute the appropriate operation. For example, if the values of variables A and B are 25 and 6 respectively, then the result of "A + B" is the integer value 31. Similarly, the result of "$A * B$" is the integer value 150.

- **comparisons**, which take one of the following forms:
  - $A = B$, the equality operation
  - $A > B$, the greater than operation
  - $A < B$, the less than operation
  - $A \geq B$, the greater than or equal operation
  - $A \leq B$, the less than or equal operation
  - $A \neq B$, the non-equality operation.

  The comparison statements simply take the values of variables A and B and perform the appropriate logical compare operation yielding either the Boolean value "true" or "false". For example, if the values of variables A and B are 25 and 6 respectively, then the result of "A = B" is the Boolean value "false" Similarly, the result of "A > B" is the Boolean value "true".

  The comparison statements above can be combined with the two logical operators "**and**" and "**or**". To remind you, "x **and** y" is "true" only if both "x" and "y" are true. "x **or** y" is "true" only if either "x", "y", or both are true.

There are many more statement types available, but for the time being these will suffice for our purpose.

Please note that the assignment statement can be combined with several (possibly zero) arithmetic statements. For example, one can have a statement:

$$A \leftarrow B + C$$

This statement adds the current values of variable B and C, and assigns the resulting value to variable A. Thus, if the values of variables A, B, and C are "10", "7", and "5", respectively, then after the assignment takes place, the values of variables A, B, and C are "12", "7", and "5", respectively. Note that only the value of variable A was effected. The value of the other two variables remain the same as before.

2.3.2 Example 1

As you will see at the end of the chapter (Section 2.6), the sorting algorithm we depicted in Figure 2.1 is a bit complex. You are not ready yet to design a low-level equivalent algorithm corresponding to it. Instead, we will start with a simpler example of taking a set of n ($n \geq 2$) integers and sum all the integers up.

For example, if the set consists of 6 items:

10, 2, 7, 22, 1, 100

then the result will be:

142

In what follows, we will first design the high-level algorithm (Algorithm 2a) for that task, design the appropriate data structures and instructions, and finally come up with a low-level, more precise equivalent algorithm (Algorithm 2b). This new algorithm is sufficiently detailed to allow us to construct the corresponding computer program.

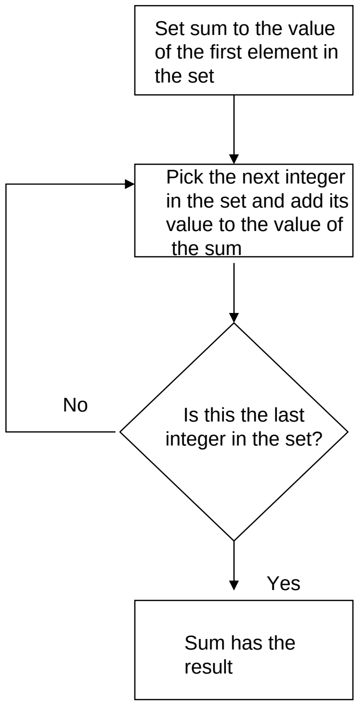
Set sum to the value of the first element in the set
Pick the next integer in the set and add its value to the value of the sum
No
Is this the last integer in the set?
Yes
Sum has the result

Figure 2.3 Algorithm 2a to sum up a set of integers.

The high-level algorithm presented in Figure 2.3 is fairly intuitive and easy to follow. The input is a set of integers and the output is an integer named *sum*.

Let's take a closer look at the algorithm 2a. In its current high-level form it is a bit too vague. The first thing we need to do is to "formally" define the data items that are to be used in the algorithm.

The various data structures needed to implement the summation algorithm are:

- The set of integers. We represent the set as an array of integers of size "n", named `in` (the individual entries in the array are `in[0]`, `in[1]`, . . . , `in[n-1]`).
  Thus, for the above set we have: `in[0]` = 10, `in[1]` = 2, . . . , `in[5]` = 100.
- The *sum* value. We represent it as an integer named: `sum`.
- We represent the array size "n" by an integer named `size`.
- We need to have an index to "point" to the various entries in the array `in`. This index is represented by an integer named `i`.

Next, we design the summation algorithm, which uses the above data structures and only employs a combination of some of the formal statements (assignment, arithmetic, and comparisons). The resulting low-level algorithm (Algorithm 2b) is depicted in Figure 2.4. To help with explaining how the algorithm work, we label each of the components in the flow chart (rectangles and diamonds) with a number.

Let us examine this algorithm more closely to make sure that you really understand it. We will do so by tracing the execution of the algorithm for the case where `size` = 6 and the set consists of the 6 integers above.

The algorithm starts by assigning the content of `in[0]`, which is the value "0" to the variable `i` and the value "10" to the variable `sum` (boxes 1 and 2 respectively). Following this, the value of variable `i` is incremented by "1" (box 3) resulting in variable `i` being set to the value "1". Next, the value of `sum` is set to "12" (box 4) since the value of of `sum` is "10" and the value of `in[i]` is "2", and their addition yields the value "12", which is assigned to `sum`. Next, the comparison of variables `i` and `size` is performed (box 5). Since "1" is less than "6", the comparison yields the Boolean value "true" and the execution shifts to box 3.

In box 3, the value of variable `i` is incremented by "1" resulting in variable `i` being set to the value "2". Next, the value of `sum` is set to "19" (box 4) since the value of of `sum` is "12" and the value of`in[i]` is "7", and their addition yields the value "19". Next, the comparison of variables `i` and `size` is performed (box 5). Since "2" is less than "6", the comparison yields the Boolean value "true" and again the execution shifts to box 3.

In box 3, the value of variable `i` is incremented by "1" resulting in variable `i` being set to the value "3". Next, the value of `sum` is set to "41" (box 4) since the value of of `sum` is "19" and the value of `in[i]` is "22", and their addition yields the value "41". Next, the comparison of variables `i` and `n` is performed (box 5). Since "3" is less than "6", the comparison yields the Boolean value "true" and again the execution shifts to box 3.

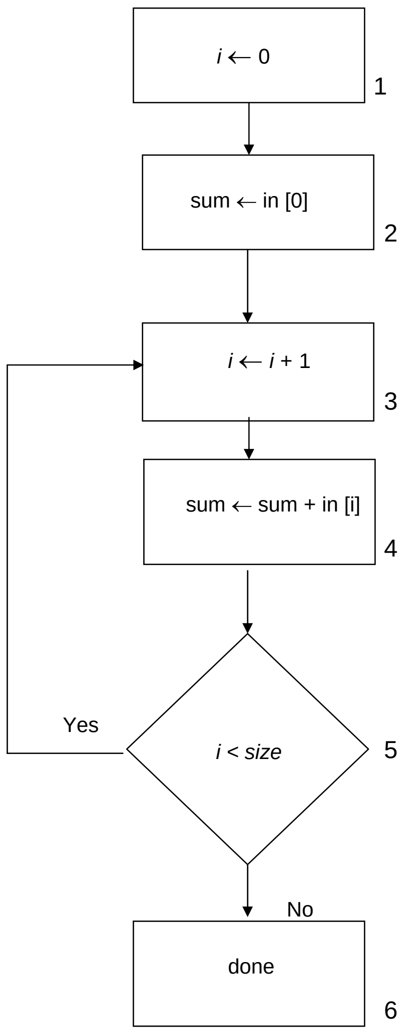

Figure 2.4 Algorithm 2b: A low-level algorithm corresponding to Algorithm 2a.

By now you should have gotten the general idea. There is an iterative process going on where boxes 3, 4, and 5, are executed in each **iteration**. This process continues until the value of variable `i` reaches the value "6". At that time, the comparison yields the Boolean value "false" and the execution shifts to box 6, and algorithm terminates.

We encourage the reader to complete the entire execution of the algorithm and verify that at the point of termination the value of variable `sum` is indeed "142".

2.3.3 Example 2

Let us provide one additional example to illustrate our concepts. Suppose that instead of summing up a set of integers, we sum up the integers:

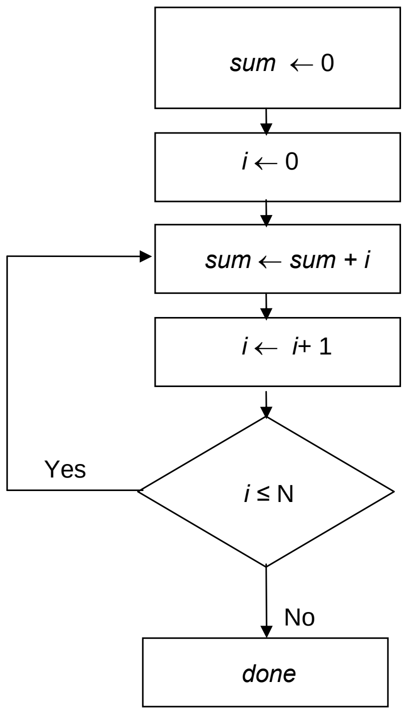

Figure 2.5 Algorithm 3 to sum up the integers 1 to N.

$$1, 2, 3, \dots, N$$

For example, if the N = 6, then the result will be:

$$21$$

In contrast to the development of the Algorithm 2b (Example 1), we skip the presentation of a high-level algorithm and present a low-level precise algorithm.

The first thing we need to do is to "formally" define the data items that are to be used in the algorithm:

- An integer `sum`, whose value represents the summation result.
- An integer variable `N`, whose value represents the largest number to be summed up.
- An integer variable `i`, whose value ranges between "1" and N.

Next, we design the algorithm (Algorithm 3), which uses the above data structures and only employs a combination of some of the formal statements (assignment, arithmetic, and comparisons). The algorithm is depicted in Figure 2.5.

```
#include <stdio.h>
main()
{

    data declaration

    your program

}
```

Figure 2.6 General structure of a typical C program.

The algorithm presented in Figure 2.5 is fairly intuitive and easy to follow. The input is the integer named N and the output is an integer named *sum*. As we shall shortly see, this algorithm is sufficiently detailed and precise that it can be easily transformed to a formal computer program.

We encourage the reader to trace the execution of the algorithm (just as we have done for Algorithm 2b—Figure 2.4) and verify that indeed at the point of termination the value of variable `sum` is "21".

2.4 Basic Computer Programming

An algorithm provides the blue print for creating a computer program. Once you have devised an algorithm and convinced yourself that is correct, you are ready to start designing the corresponding program.

There are two distinct activities that must take place after your initial algorithm design:

- The design of the appropriate data structures.
- The design of the appropriate code (program).

The amount of effort that needs to take place in designing both the data structures and the program depends on the level of details of the algorithm you have devised.

In this chapter, we have chosen the C programming language to illustrate all the concepts. We could have as well chosen any other high-level language, but we feel that C is easier to master than some of the other languages,

2.4.1 General Program Structure

Computer programs must follow strict guidelines to compile correctly. Almost any deviation from these standards will result in a compilation error. The language C, requires that all programs have the general structure shown in Figure 2.6. Please do not worry about the precise meaning of this general structure. Simply make sure that all the programs you write adhere to that structure. The only parts of the program that you will need to provide are the appropriate data and programming code segments.

The data declaration part is the place where you declare all the data structures that will be used in your program (see Section 2.4.2). The programming code segment consists of a finite sequence of statements, each of which must be one of the following type:

- **An I/O statement**; see Section 2.4.3.
- **An assignment statement**; see Section 2.5.1.
- **A decision statement**; see Section 2.5.2.
- **An iterative statement**; see Section 2.5.3.

Each statement must end with a semicolon.

Any C program must have the following name convention:

```
<name>.c
```

Where <name> is an arbitrary alphanumeric string, such as MyProgram.c, sum.c, etc.

To compile a C program, say MyProgram.c, you must execute the statement:

```
cc MyProgram.c
```

The object code corresponding to the compilation is placed in:

```
a.out
```

Finally, to execute that program, simply do:

```
a.out
```

2.4.2 Data Declaration

As we stated above, in this Chapter, we will confine our attention to the very simple data structures discussed in Section 2.2, namely simple variables and arrays.

An integer variable, say `a` is defined in the language C as follows:

```
int a;
```

One can define several integer variables using the short hand notation:

```
int a, b, c;
```

An integer array, say `b`, of size “8” is defined in the language

```
int b[8];
```

As was the case with integer variables one can define several arrays using the short hand notation:

```
int b[8], c[100], d[28];
```

Please note that each data declaration statement must end with a semicolon.

2.4.3 Simple I/O Statements

The I/O structure of C can be quite complex. For the time being we will focus on a very simple I/O structures that will allow you to write some simple programs.

2.4.3.1 The Output Command

The output command takes the form:

```
printf(A);
```

where A can be either a variable name, or a string of alphanumeric characters surrounded by quotes. (e.g, "hello").

The execution of the printf command will take the value of "A" and display it on your computer screen.

We can now write our first C program, which simply prints out (on your computer screen) the string:

My First C Program

The program is depicted in Figure 2.7. We encourage you to write this program, compile it, and execute it, as instructed above. Please note that for this simple program there is no data declaration.

When you execute the above program, you will discover that the output of the program is displayed on a single line with no carriage return. Indeed, the execution of a number of different `printf` commands will continuously output on a single line, with no line break. To force the output to start at a new line, you must explicitly add the "new-line" character to your output command. The new-line character in C is designated by "\n".

Thus, in the above program, to force a new line, simply add to the program:

```
printf("\n");
```

after the first "printf" statement. Alternatively, you can combine both `printf` commands into a single command as:

```
printf("My First C Program\n");
```

Lastly, you need to know how to print out an integer variable, say `A`. The output command will look like:

```
printf("%d", A);
```

Please do not worry, for the time being, about the precise meaning of the various symbols in the command. The only thing you need to do when you want to output an variable, say `test`, is to to use test instead of A.

```
#include <stdio.h>
main()
{

     printf(“My First C Program”);

}
```

Figure 2.7 Simple C program to print out the string ‘My First C Program”.

```
#include <stdio.h>
main()
{

     int A;

     scanf(“%d”, &A);
     printf(“%d”, A);

}
```

Figure 2.8 Simple C program to input an integer and print it out.

2.4.3.2 The Input Command

We shift now to discussing the input command, that allows one to take an input (usually from what you type on your keyboard) and assign the input to a single integer variable.

Let `A` be the integer variable that you want to input to. The input command will look like:

```
scanf(“%d”, &A);
```

As before, please do not worry, for the time being, about the precise meaning of the various symbols in the command. The only thing you need to do when you want to input an integer to a variable, say `test`, is to to use &test instead of &A.

We can now write our second C program to input an integer from the keyboard and print it out. This program is shown in Figure 2.8.

2.4.4 Comments

When you write a program, it is very useful to add comments explaining your choice of data and what the various instructions are suppose to accomplish. The comments are optional; you do not need to add comments to your program. It is up to you decide if to add comments and how much commenting you wish to add.

The comments are useful not only for yourself but to others who may need to look at your program or even modify your program.

In the language C, comments can be placed anywhere in the program using the notation:

```
/* place your comment here */
```

For example, in the program depicted in Figure 2.8, you could have added just before the declaration "`int A`", the following:

```
/* Program to input two integers and output their sum */
```

2.5 Instruction Set

In this section we will cover the basic instructions sets supported by C. Before proceeding with this, let us quickly review some of the notations used in C that differ from the ones we presented in Section 2.3.1. Please note that various programming languages may uses their own notation, which may vary slightly from language to language.

- $A \leftarrow B$ is written in C as "`A = B`".
- $A \div B$ is written in C as "`A / B`".
- $A = B$ is written in C as "`A == B`".
- $A \neq B$ is written in C as "`A != B`".
- $A \geq B$ is written in C as "`A >= B`".
- $A \leq B$ is written in C as "`A <= B`".
- `x` **and** `y` is written in C as "`A && B`".
- `x` **or** `y` is written in C as "`A || B`".

2.5.1 Assignment Construct

As stated above, the assignment statement in C looks like:

```
A = B;
```

Let us illustrate this by creating a simple program to input two integers, add them up, and print the result out. This program is depicted in Figure 2.9.

2.5.2 Decision Constructs

As we have seen, it is crucial to be able to test some condition and depending on the outcome of the test execute different parts of a program. There are various constructs in C to accomplish this. In this section we focus our attention on the the "**if**" statement, which has the following form:

```
if (boolean expression)
    { <statements> }
```

The boolean expression consists of one or more comparison statements connected by the **and** and **or** operators. If the evaluation of the boolean expression yields the value true, then the <statements> are executed. The

```
#include <stdio.h>
main()
{

     int A, B, sum;

     scanf("%d", &A);
     scanf("%d", &B);
     sum = A + B;
     printf("%d", sum);

}
```

Figure 2.9 C program to input 2 integers, add them up, and print the result.

<statements> can be a number of different instructions, including other **if** statements.

To illustrate, let us write a simple program that inputs two integers and outputs the smallest of the two. The program is depicted in Figure 2.10.

There is a variation of the **if** statement that make writing programs a bit more convenient. This variation, the "if then else" statement, which takes the form:

```
if (boolean expression)
   { <statements 1> }
   else { <statements 2> }
```

This statement is similar to the first form of the **if** statement except that if the evaluation of boolean expression yields the value "false", then the <statements 2> part is executed.

To illustrate, let us rewrite the program depicted in Figure 2.10 using the if-then-else statement, The program is depicted in Figure 2.11.

```
#include <stdio.h>
main()
{

     int A, B, temp;

     scanf("%d", &A);
     scanf("%d", &B);
     if A < B
        {temp = A;}
     if A >= B
        {temp = B;}
     printf("%d", temp);

}
```

Figure 2.10 C program to illustrate the `if` statement.

```
#include <stdio.h>
main()
{

     int A, B, temp;

     scanf(“%d”, &A);
     scanf(“%d”, &B);
     if A < B
        {temp = A;}
        else {temp = B;}
     printf(“%d”, temp);

}
```

Figure 2.11 C program to illustrate the `if then else` statement.

2.5.3 Iteration Constructs

There are a number of different iteration constructs in C that allows one to specify “loops”. Here, we will introduce one of these constructs—the **while** statement, which has the following form:

```
while (boolean expression)
   { <statements> }
```

The boolean expression consists of one or more comparison statements connected by the **and** and **or** operators. The <statements> part may consist of a number of different instructions, including other **while** statements.

The general idea behind the **while** statement is that the <statements> part is repeatedly executed as long as the evaluation the boolean expression yields the value “true”. Notice that if the <statements> part does not change some of the variables appearing in the boolean-expression part, this will result in an **infinite loop**—the program will never terminate.

Let us illustrate the `while` construct by creating a C program corresponding to Algorithm 3 (Figure 2.5). The program is shown in Figure 2.12.

2.6 Putting it All Together

We are finally in a position to revisit our original sorting algorithm of Figure 2.1. As we stated before, this algorithm, in its current form, is a bit too vague and we need to do some tightening.

The first thing we need to do is to “formally” define the data items that are to be used in the algorithm.

- The *orig* list. We represent the list as an array of integers named: `orig[0]`, `orig[1]`, ... , `orig[n-1]`.
- The *final* list. We represent the list as an array of integers named: `final[0]`, `final[1]`, ... , `final[n-1]`.

```
#include <stdio.h>
main()
{

     int sum, i, n;

     sum = 0;
     i = 0;
     scanf("%d", &n)
     while ( i <= n )
            { sum = sum + i; i = i + 1; }
     printf("%d", sum);

}
```

Figure 2.12 C program to print out the sum 1 to N.

- The "item on side". We represent it as an integer named: `side`.
- The "next" item. We represent it as an integer named: `next`.
- We represent the array size "n" by an integer named `size`.
- We need to have two index "pointers" to the various locations in both the `orig` and `final` arrays. We represent the index pointers by two integers named: `i` and`k`.

The low-level algorithm is shown in Figure 2.13. Note that step 1 to initialize the array `orig` needs to be expanded upon reading in the n values to sort. We leave this task as an exercise to the user.

We can now write the C program corresponding to this algorithm We will write the program for the case where the size of the set is "10". The program, depicted in Figure 2.14, reads the integer values one-by-one from the keyboard input into the array `orig`. Then, the algorithm from Figure 2.13 is coded. It consists mainly of two nested **while**-loops. The program terminates with the print-out of the sorted numbers from array `final`.

2.7 Epilogue

Now that you have a basic understanding on how to write some simple programs, it is time to discuss a number of important issues that one must consider when designing and writing a program.

2.7.1 Testing

After you have completed writing your program you need to to convince yourself that the program is indeed correct; that is, that it does what it is suppose to accomplish.

Ideally, one would like to have a "formal proof" that this is indeed the case. This, however, is quite a tall order, since formal proof methods have not fully matured yet, and are quite tedious.

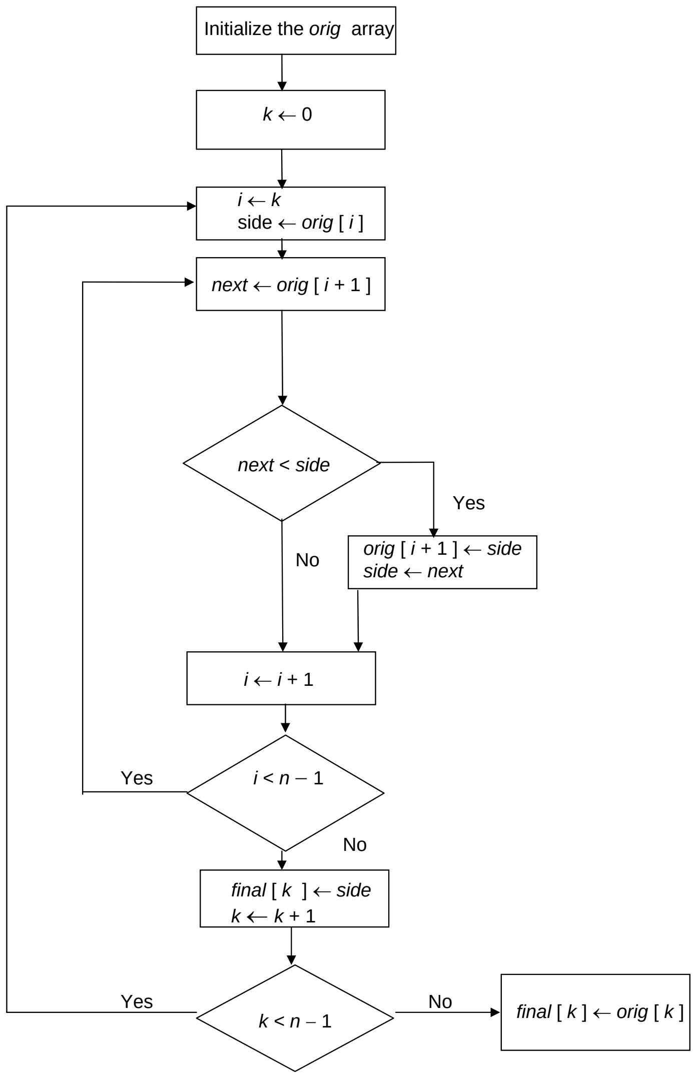

Figure 2.13 Algorithm to sort a set of integers.

Instead, most people resort to testing, sometimes called **debugging** (debugging is more than just testing; it also a procedure for removing the errors). Testing provides the means to find errors in the program; it does not provide the means of assuring that the program is “bug free”.

Testing usually starts by considering *all* possible inputs to the program and testing the result for each of these inputs. Since the number of inputs to the program is usually quite large (and sometimes even infinite), this is not a viable approach. Instead, one tries to cluster the inputs into a small number of representative cases, and perform the tests on this small set. Usually, in the case of integers, we divide the set of integers into 3 small representative sets—

```
#include <stdio.h>
main()
{

     int orig[10], final[10];
     int side, next;
     int i, k;

     /* Initialize the array orig */

     i = 0;
     while (i < 10)
         { scanf("%d", &orig[i]); i = i + 1; }

     k = 0;
     i = k;
     while (k < 10)
         { i = k; side = orig[i];
         while (i < 10)
             { next = orig[i+1];
             if (next < side)
                 { orig[i+1] = side; side = next;}
             i = i + 1; }
         final[k] = side;
         k = k + 1; }
     final[k] = orig[k];

     i = 0;
     while (i < 10)
         { printf("%d", final[i]); i = i + 1; }
}
```

Figure 2.14 C program to sort a set of 10 integers.

some negative integer (say -20), the integer "0", and some positive integer (say 33). This obviously need to be refind for your particular program.

Looking back at the program depicted in Figure 2.8, if you run that program three time with the inputs -20, 0, and 33, respectively, you will see that the output is indeed correct. For the program, shown in Figure 2.9, you should try several combinations of positive and negative integers for the two input integers to see that indeed the program is correct. Finally, for the program shown in Figure 2.12, please run it several times for the different values of N and make sure that the summation result is indeed correct.

Let us look a bit closer to the summation program of Figure 2.12. Note that if you run it with N being a negative integer (say -6), the result is "0", which seems reasonable (it is not entirely clear what should be the result if N is negative). Also, please run the program with N = 1,000,000 and see the result. If you run on a typical 32-bit laptop, the result you get is a negative integer, which is clearly wrong! (if you do not get a negative integer when you run your program, make N larger; for a sufficiently large N, you will indeed get as a result a negative integer). So why are we getting a negative integer? The answer lies in the fact that our computers have a finite number of bits to represent integers and when a positive integer becomes too large it turns into a negative integer (see Chapter 9).

2.7.2 Not all programs are created equal

Recall that there exists many different correct algorithms to accomplish the same task. This means that the corresponding programs will most likely vary too. Some may take less time to complete than others, may require less memory space than others, etc. These differences may be minor; but there are many cases where the differences are quite significant (e.g., execution time of 1 minutes versus 10 hours).

Let us illustrate this by revisiting the algorithm of Figure 2.12 to sum up the integers "1" to "N". The summation is accomplished by going through the loop consisting of the following two statements:

```
sum = sum + i;
i = i + 1;
```

N times.

Thus, if N = 10,000,000, the loop executes 20,000,000 statements before completing.

Now, it is a well known fact that:

$$1 + 2 + 3 + \ldots + N = N * (N + 1)/2$$

Using this fact, we can write another program to do the summation, which is shown in Figure 2.15. This program seems to be equivalent to the one depicted in Figure 2.12, but executes much faster.

It should not be difficult to convince yourself that this new program is faster. Indeed, the program does not have a single loop; it basically consists of one single statement:

```
#include <stdio.h>
main()
{

     int sum, n;

     scanf("%d", &n)
     sum = (n * (n + 1)) / 2;
     printf("%d", sum);

}
```

Figure 2.15 C program to sum up the integers 1 to N.

```
sum = (n * (n + 1)) / 2;
```

Compare it with the first program, which for N = 20,000,000, executes 40,000,000 statements (remember that the loop consists of two statements).

We claimed above that the programs depicted in Figures 2.12 and 2.15 are equivalent. But is it the case? We already discussed above that for the case program, running it with N = -6, the result is “0”. If you run the second program with N = -6, you will get as a result “15”. Clearly the two programs are not equivalent for all possible inputs; only for nonnegative integers we get the same results. The solution is to replace the statement:

```
sum = (n * (n + 1)) / 2;
```

with the statement:

```
if (n < 0)
   {sum = 0;}
    else {sum = (n * (n + 1)) / 2;}
```

2.8 Summary

- An **algorithm** is a finite set of instructions for accomplishing some specific task. It is a formal way of specifying a solution to a specific problem one wishes to solve. It provides the blue print for creating a computer program.
- The overall logical structure of an algorithm can be expressed graphically by a **flow chart**, which is built up from **rectangles** (which represent simple action steps), **diamonds** (which represent decision branching steps), and **arrowed lines** (which represent iteration steps and link some of the steps to some other steps).
- Integral to computer programming is the presence of data. Each data element in a computer system, in its simplest form, is a sequence of bits that represents some information.
- The data items in a computer program are stored in **variables**. A variable is a “datum container” that at any point in time has some single value.
- It is sometimes convenient to have a collection of variables be grouped under one unique name. For this purpose, most programming languages provide an **array** type data structure.
- A computer program consists of a data declaration part and a set of instructions that manipulate the data. The basic instructions sets consist of:
  - I/O statements
  - assignment statements
  - decision constructs
  - iteration constructs
- The C programming language provides many different instructions, including:

    - I/O—scanf and printf statements
    - decision—if-then-else statement
    - iteration—while statement
- After a program has been written one must test it to ensure that it is correct; that is, it does what it is suppose to accomplish.
- There exists many different correct programs to accomplish the same task. Some may take less time to complete than others, may require less memory space than others, etc.

Review Terms

- Algorithm
- Flow chart
- Variables
- Array
- Iteration construct
- Decision construct
- Infinite loop
- Debugging
- Boolean expression
- Basic statements
    - assignment
    - arithmetic
    - comparisons
- Logical operators
    - **and**
    - **or**
- I/O statements
    - scanf
    - printf
- Data declaration
- Comment statement
- Instruction set
    - if statement
    - if then else statement
    - while statement
- program testing

Exercises

2.1 Design an algorithm to sort an array of N integers in descending order (largest integer first).

2.2 Write a C program corresponding to the algorithm you have designed in Exercise 2.1.

Bibliographical Notes

Basic book on C Kernighan and Ritchie [1988].