– Institut für Mechanik und Mechatronik / Abteilung 6
– Intelligente Handhabung und Robotertechnik

Einführung in C(++)

VO 318.060

Part 1 C

o.Univ.-Prof. Dr. Dr.h.c.mult. Peter Kopacek

Oktober 2004

– Technische Universität Wien

1. The C programming language

1.1 History

Originally designed and implemented in the Bell Laboratories in 1972–1973 by Dennis Ritchie for the programming of the UNIX operating system.

Many of the important ideas of C stem from the language BCPL (Basic Combined Programming Language), proceeded through the language B.

First publication of "The C programming language" in 1978 by Brian Kernighan and Dennis Ritchie.

ANSI (American National Standard Institute) standardization committee working since 1983, "ANSI C" standard completed in 1988.

1.2 Characteristics

C is a general-purpose programming language.

C is not tied to any one operating system or machine.

BCPL and B are "typeless", C provides a variety of data types.

Pointers are provided for machine-independent address arithmetic.

C provides the fundamental control-flow constructions.

Functions in C may return values of any type.

Any function may be called recursively.

The functions may exist in separate source files that are compiled separately.

Variables may be internal to a function, external but known only within a single source file, or visible to the entire program.

A preprocessing step performs macro substitution on program text, inclusion of other files, and conditional compilation.

C is a relatively "low level" language. C provides no operations to deal directly with composite objects such as character strings, sets, lists, or arrays. C itself provides no input/output facilities; there are no read or write statements. All of these higher-level mechanisms must be provided by functions. Most C implementations have included a standard collection of such functions.

C offers only straightforward, single-thread control flow: test, loops, grouping, and sub-programs, but not multiprogramming.

2. Introduction – A Tutorial to C

In a first step we want to try to get to the point where you can write useful programs. Therefore we are concentrating on the basics: variables and constants, arithmetic, control flow, functions, simple input and output. We are leaving out of this chapter features of C that are important for writing bigger programs: pointers, structures, most of the operators, several control-flow statements, and the standard library. The whole story will be the content of advanced chapters.

2.1. Getting started

The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages:

Print the words "`hello, world!`" to the screen.

2.1.1. Creating a program step by step

Some of the steps depend on the computer system and the operating system you are using. But in general we have the following steps:

- Create the program text (⇒ text editor)
- Compile it successfully (⇒ compiler)
- Link it (⇒ linker)
- Load and run it (⇒ operating system)
- Find out where the output went (⇒ standard output, monitor)

2.1.2. "Hello, world!"

In C the program to print "`hello, world`" is

```
hello.c
      1   #include <stdio.h>

      2   main()
      3   {
      4     printf("hello, world!\n");
      5   }
```

A C **program** consists of **functions** and **variables**. A function consists of **statements**, that specify the computing operations to be done, and **variables**, that store values used during the computation. Normally you are at liberty to give functions whatever names you like, but "**`main`**" is special – your program begins executing at the beginning of **`main`**. This means that every program must have a **`main`** somewhere.

The function **`main`** will call other functions, some that you wrote, and others from libraries. The first line

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

tells the compiler to include information about the standard input/output library, the according header file "**stdio.h**".

One method of exchanging data between functions is arguments. The calling function provides a list of values, called **arguments**, to the function it calls. The argument list after the function name is surrounded by parentheses **()**.

In this example,

```
2   main()
```

is defined to be a function that expects no arguments, which is indicated by the empty list **()**.

The statements of a function are enclosed in braces **{}** (lines 3 and 5 in the example).

A function is called by naming it, followed by a parenthesized list of arguments. So the line

```
4     printf("hello, world!\n");
```

calls the function **printf** with the argument **"hello, world!\n"**. **printf** is a function provided by the standard library that prints output.

A sequence of characters in double quotes, like **"hello, world!\n"**, is called a **character string** or **string constant**. The sequence **\n** in the string is C notation for the **newline character**. As **printf** never supplies a newline automatically, you could write instead of line four

```
4a    printf("hello, ");
4b    printf("world!");
4c    printf("\n");
```

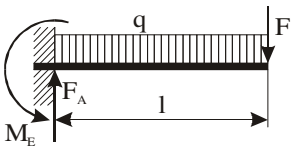
to produce identical output.

Notice that **\n** represents only one character. Some other **escape sequences** for (invisible) characters are: **\t** for tabulator, **\"** for double quote, **\b** for backspace, and **** for the backslash.

2.2. Variables and Arithmetic Expressions

Calculate the shear force and bending moment in a straight beam:
Length l = 10 m; Force F = 100 N; Distributed load q = 10 N/m

Figure 1
A straight beam



Print the following table of shear force and bending moment over x:

```
0       200     -1500
1       190     -1305
...     ...     ...
10      100     0
```

Shear force $Q(x) = F + q \cdot (l - x)$

Bending moment $M(x) = F \cdot (x - l) - \frac{q}{2} \cdot (l - x)^2$

beam.c

```
 1  #include <stdio.h>

 2  /* straight beam:
 3     print shear force and bending moment table  */

 4  main()
 5  {
 6    int x, l, q, F, M, Q;
 7    int diff;
 8    int lower, upper, step;

 9    l = 10;    /* length in [m]   */
10    F = 100;   /* force in [N]    */
11    q = 10;    /*        in [N/m] */

12    lower = 0;  /* lower limit of x */
13    upper = l;  /* upper limit of x = length of beam */
14    step = 1;   /* step size */

15    x = lower;
16    while(x <= upper) {
17      diff = l - x;
18      Q = F + q * diff;
19      M = -F * diff - q * diff * diff / 2;
20      printf("%d\t%d\t%d\n", x, Q, M);
21      x = x + step;
22    }
23  }
```

2.2.1. Comments

The two lines

```
 2  /* straight beam:
 3     print shear force and bending moment table  */
```

are a **comment**. Any characters between `/*` and `*/` are ignored by the compiler. They may be used freely to make a program easier to understand. Comments may appear anywhere a blank or tab or newline can.

2.2.2. Variables

All variables must be declared before they are used. A **declaration** announces the properties of variables; it consists of a type name and a list of variables, such as

```
6     int x, l, q, F, M, Q;
7     int diff;
8     int lower, upper, step;
```

The word **`int`** means that the variables listed are integers, by contrast with **`float`**, which means floating point. C provides several other basic data types, including:

- **`char`** character – a single byte
- **`short`** short integer
- **`long`** long integer
- **`double`** double-precision floating point

The sizes and the ranges of these objects are machine-dependent. For the storage of integer values usually 2 bytes (= 16 bits) are used. So values up to $2^{16} - 1 = 65\,535$ could be stored. But most of the time 1 bit is used for the sign (0 means + and 1 stands for –). Therefore all values between $-2^{15} = -32\,768$ and $2^{15} - 1 = +32\,767$ can be represented exactly.

Negative values are stored as "binary complement": "–x" = NOT (x) + 1 = $2^n - x$, where n is such that the 1 in 2^n is one place to the left of the most significant digit of x. To calculate you negotiate the positive binary number bit by bit (by exchanging the values 0 and 1), then add 1.

```
  0000 0001                    decimal 1 (8 bit integer)
  1111 1110                    NOT 1
  1111 1111                    -1
1 0000 0000                    2ⁿ where n = 8
```

Subtraction of integers therefore is equal to addition with the complement.

A **`float`** number is typically a 32-bit quantity, with at least six significant digits and magnitude between about 10^{-38} and 10^{+38}. Here floating point storage is used. For instance, $1\,700\,000 = 0.17 \cdot 10^7$, where 0.17 is called mantissa 10 the base, and 7 exponent. Only mantissa and exponent (m = 17 and e = 7, which are integers!) have to be stored, as there is an agreement saying that the base is 10 and the first digit behind the comma must not be 0. IEEE[1] floating-point standard defines (32 bit floating-point):

```
 1 bit   sign
 8 bits  exponent (including sign) with base 2 (2¹²⁷ ≈ 10³⁸)
23 bits  mantissa
32 bits
```

2.2.3. Assignments

The computation begins with the **assignment statements**

[1] IEEE = Institute of Electrical and Electronics Engineers, Inc. ("eye-triple-E")

```
 9     l = 10;    /* length in [m]  */
10     F = 100;   /* force in [N]   */
11     q = 10;    /*       in [N/m] */
12     lower = 0;  /* lower limit of x */
13     upper = l;  /* upper limit of x = length of beam */
14     step = 1;   /* step size */
15     x = lower;
```

which set the variables to their initial values.

Each line of the output table is computed the same way, so we use a loop that repeats once per output line:

```
16     while(x <= upper) {
        …
23     }
```

The **`while`** loop operates as follows: The condition in the parentheses is tested. If it is true (**`x`** is less than or equal to **`upper`**), the body of the loop (lines 17 to 21 enclosed in braces) is executed. Then the condition is re-tested, and if true, the body is executed again. Then the test becomes false the loop ends, and execution continues at the statement that follows the loop.

The body of a **`while`** loop can be one or more statements enclosed in braces, as in beam calculator, or a single statement without braces, as in

```
while(i < j)
  i = 2 * i;
```

Most of work gets done in the body of the loop. The difference, which is needed three times, is calculated and stored in the variable **`diff`**, then the force and moment is computed and assigned to the variables **`Q`** and **`M`**.

```
17       diff = l - x;
18       Q = F + q * diff;
19       M = -F * diff - q * diff * diff / 2;
```

This example shows how **`printf`** works as general-purpose output formatting function. Its first argument is a string of characters to be printed, with each **`&`** indicating where one of the other (second, third, …) arguments is to be substituted, and in what form it is to be printed. For instance, **`%d`** specifies an integer argument, so the statement

```
20       printf("%d\t%d\t%d\n", x, Q, M);
```

causes the values of the three integers **`x`**, **`Q`**, and **`M`** to be printed, with a tab (**`\t`**) between them.

Each **`%`** construction in the first argument of **`printf`** is paired with the corresponding second argument, third argument, etc.; they **must** match up properly by number and type, or you'll get wrong answers!

The output of our program is not very pretty because the numbers are not right-justified. To fix this problem we augment each **`%d`** in the **`printf`** with a width, the numbers printed will be right-justified in their fields. For instance, we might say

```
20a      printf("%3d %6d %6d\n", x, Q, M);
```

to print the first number of each line in a field three digits wide, and the others in fields six digits wide, like this:

```
0     200   -1500
1     190   -1305
2     180   -1120
3     170    -945
…       …       …
```

2.2.4. Format strings

As each `%` construction has to match the variable type you specified in the argument list (as mentioned above). So there must be a conversion specification for each variable type. Some conversion specification recognized by **`printf`**:

	integer
	integer, at least 6 characters wide, right-aligned
	point
	point, at least 6 characters wide, right-aligned
	point, 2 characters after decimal point, right-aligned
	point, at least 6 characters wide and 2 after decimal point
	ctal
	exadecimal
	r
	ng

A simple test program for **`printf`** statements:

printf.c

```
 1  #include <stdio.h>

 2  main()
 3  {
 4    char c;
 5    int i;
 6    float f;
 7    c = 64;
 8    i = 256;
 9    f = 3.141592653589;
10    printf("charcter c = %c - ASCII code decimal %3d, "
11      "octal %o, hexadecimal %x\n", c, c, c, c);
12    printf("integer  i = %3d = 0x%x\n", i, i);
13    printf("float    f = %12.10f is about %6.4f\n", f, f);
14  }
```

produces the output:

```
charcter c = @ - ASCII code decimal  64, octal 100,  hexadecimal 40
integer  i = 256 = 0x100
float    f = 3.1415927410 is about 3.1416
```

2.2.5. Arithmetic expressions

In our program we used **`int`**s, which is fine as all calculated values are integer values.

Use the formula $°C = \frac{5}{9} \cdot (°F - 32)$ to print a table of Fahrenheit temperatures and their centigrade or Celsius equivalents.

tempera1.c

```
 1  #include <stdio.h>
 2  #include <conio.h>

 3  main()
 4  {
 5    int fahr, celsius;
 6    int lower, upper, step;

 7    lower = 0;    /* lower limit of temperature table */
 8    upper = 300;  /* upper limit */
 9    step = 20;    /* step size */

10    clrscr();
11    printf("  °F   °C\n");
12    fahr = lower;
13    while(fahr <= upper) {
14      celsius = 5 * (fahr - 32) / 9;
15      printf("%4d %4d\n", fahr, celsius);
16      fahr = fahr + step;
17    }
18  }
```

This program is very similar to the last example. The line of interest is

```
14      celsius = 5 * (fahr - 32) / 9;
```

The reason for multiplying by **5** and then dividing by **9** instead of just multiplying by **5/9** is that in C, as in many other languages, integer division **truncates**: any fractional part is discarded. Since **5** and **9** are integers, **5/9** would be truncated to zero and so all the Celsius temperatures would be reported as zero.

The output of the program looks like this:

```
 °F   °C
  0  -17
 20   -6
 40    4
  …    …
280  137
300  148
```

The line

```
10    clrscr();
```

clears the screen. **clrscr** is defined in the header file **conio.h**.

The problem is that because we use integer arithmetic, the Celsius temperatures are not very accurate; for instance, 0°F is actually about –17.8°C, not –17°C. To get more accurate answers, we should use floating-point arithmetic instead of integer. This requires some changes in the program. Here is the second version:

tempera2.c

```
 1  #include <stdio.h>
 2  #include <conio.h>

 3  main()
```

```
 4  {
 5    float fahr, celsius;
 6    int lower, upper, step;

 7    lower = 0;    /* lower limit of temperature table */
 8    upper = 300;  /* upper limit */
 9    step = 20;    /* step size */

10    clrscr();
11    printf("  °F   °C\n");
12    fahr = lower;
13    while(fahr <= upper) {
14      celsius = (5.0/9.0) * (fahr - 32);
15      printf("%4.0f %6.1f\n", fahr, celsius);
16      fahr = fahr + step;
17    }
18  }
```

The only differences to the first version are that **fahr** and **celsius** are declared to be **float**, and the formula for conversion is written in a more natural way. We were unable to use **5/9** in the previous version because integer division would truncate to zero. A decimal point in a constant indicates that it is floating point, however, so **5.0/9.0** is not truncated.

If an arithmetic operator has integer operands, an integer operation is performed. If an arithmetic operator has one floating-point operand and one integer operand, however, the integer will be converted to floating point before the operation is done:
fahr = lower; while(fahr <= upper); fahr - 32

2.3. The For Statement

There are plenty or different ways to write a program for a particular task. Let's try a variation on the temperature converter.

tempera3.c

```
 1  #include <stdio.h>
 2  #include <conio.h>

 3  main()
 4  {
 5    int fahr;

 6    clrscr();
 7    printf("  °F   °C\n");

 8    for(fahr = 0; fahr <= 300; fahr = fahr + 20)
 9      printf("%4d %6.1f\n", fahr, (5.0/9.0) * (fahr - 32));
10  }
```

This produces the same answer, but it certainly looks different.

Most variables are eliminated, only **fahr** remains. The lower and upper limit and the step size appear only as constants in the **for** statement. The expression that computes the Celsius temperature now appears directly as third argument of the **printf** statement. In general it is permissible to replace a variable of some type by a more complicated expression of that type.

The **for** statement is a loop, a generalization of the **while**. Within the parentheses, there are three parts, separated by semicolons. The first part, the initialisation

```
fahr = 0
```

is done once, before the loop proper is entered. The second part is the test or condition that controls the loop:

```
fahr <= 300
```

This condition is evaluated and if it is true, the body of the loop (here a single **printf**) is executed. Then the increment step

```
fahr = fahr + 20
```

is executed, and the condition re-evaluated. The loop terminates if the condition has become false. As with the while, the body of the loop can be a single statement, or a group of statements enclosed in braces. The initialization, condition, and increment can be any expressions.

2.4. Symbolic Constants

It's bad practice using "magic numbers" like 300 and 20 in a program. They provide little information to someone who reads the program later, and they are hard to change in a systematic way. It is much better to give them meaningful names. A **#define** line defines a **symbolic name** or **symbolic constant** to be a particular string of characters.

```
#define name replacement-text
```

Thereafter, any occurrence of *name* (not in quotes and not part of another name) will be replaced by the corresponding *replacement text*. The *name* has the same form as a variable name: a sequence of letters and digits that begins with a letter. The *replacement text* can be any sequence of characters; it is not limited to numbers. As this functionality is provided by the preprocessor, the line begins with a **#** character and there is no semicolon at the end of a **#define** line.

tempera4.c

```
 1  #include <stdio.h>
 2  #include <conio.h>

 3  #define LOWER 0   /* lower limit of table */
 4  #define UPPER 300 /* upper limit */
 5  #define STEP  20  /* step size */

 6  main()
 7  {
 8    int fahr;

 9    clrscr();
10    printf("  °F   °C\n");

11    for(fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
12      printf("%4d %6.1f\n", fahr, (5.0/9.0) * (fahr - 32));
13  }
```

2.5. The If-Else Statement

The next program writes the word "ice!" in a third row if the temperature is below 0°C:

tempera5.c

```
 1  #include <stdio.h>
 2  #include <conio.h>

 3  #define LOWER 0   /* lower limit of table */
 4  #define UPPER 300 /* upper limit */
 5  #define STEP  20  /* step size */

 6  main()
 7  {

 8    int fahr;
 9    float celsius;

10    clrscr();
11    printf("  °F   °C\n");

12    for(fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP) {
13      celsius = (5.0/9.0) * (fahr - 32);
14      printf("%4d %6.1f  ", fahr, celsius);
15      if(celsius < 0)
16        printf("ice!");
17      printf("\n");
18    }
19  }
```

The body of the **`for`** loop consists now an **`if`** statement. It tests the parenthesized condition, and if the condition is true, executes the statement (or group of statements in braces) that follows.

The next version of our program also writes "water" and "steam" beside the temperatures:

tempera6.c

```
 1  #include <stdio.h>
 2  #include <conio.h>

 3  #define LOWER 0   /* lower limit of table */
 4  #define UPPER 300 /* upper limit */
 5  #define STEP  20  /* step size */

 6  main()
 7  {
 8    int fahr;
 9    float celsius;

10    clrscr();
11    printf("  °F   °C\n");

12    for(fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP) {
13      celsius = (5.0/9.0) * (fahr - 32);
14      printf("%4d %6.1f  ", fahr, celsius);
15      if(celsius < 0)
16        printf("ice!");
17      if(celsius == 0)
18        printf("freezing-point");
```

```
19      if(celsius == 100)
20        printf("boiling-point");
21      if(celsius > 100)
22        printf("steam");
23      if(celsius > 0 && celsius < 100)
24        printf("water");
25      printf("\n");
26    }
27  }
```

Each of the **if** statements evaluates its condition to decide if the following statement has to be executed.

The double equals sign == is the C notation for "is equal to" (like Pascal's single = or Fortran's **.EQ.**). This symbol is used to distinguish the equality test from the single = that C uses for assignment. Be careful: newcomers to C occasionally write = when they mean ==. As we will see, the result is a legal expression, so you will get no error.

The operator **&&** means AND, so the line

```
23      if(celsius > 0 && celsius < 100)
```

says: "if **celsius** is lager than zero and **celsius** is smaller than 100", which means "if **celsius** is between 0 an 100". There is a corresponding operator **||** for OR. The precedence of **&&** is just higher than **||**. Expressions connected by **&&** or **||** are evaluated left to right, and it is guaranteed that evaluation will stop as soon as the truth or falsehood is known. If **celsius** is smaller than zero, there is no need to test whether it is smaller than 100, as the whole condition cannot come true; so the second test is not made.

There are a lot of if statements but only one of them can be the right. So we can combine them using **else**.

tempera7.c

```
1   #include <stdio.h>
2   #include <conio.h>

3   #define LOWER 0   /* lower limit of table */
4   #define UPPER 300 /* upper limit */
5   #define STEP  20  /* step size */

6   main()
7   {
8     int fahr;
9     float celsius;

10    clrscr();
11    printf("  °F   °C\n");

12    for(fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP) {
13      celsius = (5.0/9.0) * (fahr - 32);
14      printf("%4d %6.1f  ", fahr, celsius);
15      if(celsius < 0)
16        printf("ice!");
17      else if(celsius == 0)
18        printf("freezing-point");
19      else if(celsius == 100)
20        printf("boiling-point");
```

```
21        else if(celsius > 100)
22          printf("steam");
23        else
24          printf("water");
25        printf("\n");
26      }
27    }
```

The general form is

```
if(expression)
  statement1
else
  statement2
```

One and only one of the two statements associated with an **`if-else`** is performed. If the *expression* is true, $statement_1$ is executed; if not, $statement_2$ is executed. Each *statement* can be a single statement or several in braces.

The pattern

```
if(condition1)
  statement1
else if(condition2)
  statement2
…
  …
else
  statementn
```

occurs frequently in programs as a way to express a multi-way decision. Te *conditions* are evaluated in order from the top until some *condition* is satisfied; at that point the corresponding *statement* part is executed, and the entire construction is finished. If none of the *conditions* is satisfied, the *statement* after the final **`else`** is executed if it is present.

2.6. Character Input and Output

The standard library supports a possibility of input and output that is very simple. Text input or output, regardless of where it originates or where it goes to, is dealt with as streams of characters. A **text stream** is a sequence of characters divided into lines; each line consists of zero or more characters followed by a newline character.

The standard library provides several functions for reading or writing one character at a time, of which **`getchar`** and **`putchar`** are the simplest. In Borland standard library **`getchar`** returns only as soon as a newline character occurs; therefore we will use the form **`getch`**. Each time it is called, **`getch`** reads the next input character from a text stream and returns it as its value. That is, after

```
c = getch()
```

the variable **`c`** contains the next character of input. The characters normally come from the keyboard, the standard input; input form files is discussed later in this lecture.

The function **`putchar`** prints a character each time it is called:

```
putchar(c)
```

prints the contents of the integer variable **c** as a character, usually to the screen, the standard output.

2.6.1. File Copying

Using these functions you can write a surprising amount of useful code without knowing anything more about input and output. The simplest example is a program that copies its input to its output one character at a time:

```
read a character
while (character is not ESC)
  output th character just read
  read a character
```

Converting this into C gives

copy1.c

```
1   #include <stdio.h>
2   #include <conio.h>

3   /* copy input to output; 1st version */

4   #define ESC '\x1B' /* character for termination */

5   main()
6   {
7     int c;

8     clrscr();
9     c = getch();
10    while(c != ESC) {
11      putchar(c);
12      c = getch();
13    }
14  }
```

The relational operator **!=** means "not equal to".

This program would be written more concisely by experienced C programmers. In C, any assignment, such as

```
c = getch()
```

is an expression and has a value, which is the value of the left hand side after the assignment. This means that an assignment can appear as part of a larger expression. For example,

```
int a, b, c;
a = b = c = 0;
```

sets all three integer variables **a**, **b**, and **c** to zero. This is a consequence of the fact that an assignment is an expression with a value and assignments associate form right to left. It's as if we had written

```
a = (b = (c = 0));
```

If the assignment of a character to **c** is put inside the test part of a **while** loop, the copy program can be written this way:

```
copy2.c
      1   #include <stdio.h>
      2   #include <conio.h>

      3   #define ESC '\x1B'

      4   main()
      5   {
      6     int c;

      7     clrscr();
      8     while((c = getch()) != ESC)
      9       putchar(c);

     10   }
```

The **while** gets a character, assigns it to **c**, and then test whether the character was an Escape character. If it was not, the body of the **while** is executed, printing the character. The **while** then repeats. When an Escape character is read, the **while** terminates and so does **main**.

This version centralizes the input – there is only one reference to **getch**. The resulting program is more compact, and, once the idiom is mastered, easier to read.

The parentheses around the assignment within the condition are necessary. The precedence of **!=** is higher than that of **=**, which means that in the absence of parentheses the relational test **!=** would be done before the assignment **=**. So the statement

```
c = getch() != ESC
```

is equivalent to

```
c = (getch() != ESC)
```

This has the effect of setting **c** to **0** or **1**, depending on whether or not the call of **getch** encountered an Escape character.

Each Boolean expression evaluates to **0** or **1**, where **0** always means **false** and any other value stands for **true**.

```
int booltest;
booltest = 3 == 4;    /* booltest is 0 now */
booltest = 3 != 4;    /* booltest is 1 now */
```

2.6.2. Character counting

The next program counts characters; it is similar to the copy program.

```
count1.c
      1   #include <stdio.h>
      2   #include <conio.h>

      3   #define ESC '\x1B'
```

```
 4   /* count characters in input */

 5   main()
 6   {
 7     long nc;  /* number of characters */

 8     clrscr();
 9     nc = 0;
10     while(getch() != ESC)
11       ++nc;
12     printf("%ld character(s)\n", nc);
13   }
```

The statement

```
11       ++nc;
```

presents a new operator, **++**, which means **increment by one**. You could instead write **nc = nc + 1** but **++nc** is more concise and often more efficient. There is a corresponding operator **--** to decrement by one. The operators **++** and **--** can be either prefix operators (**++nc**) or postfix (**nc++**). In both cases, the effect is to increment **nc**. But the expression **++nc** increments **nc** before its value is used, while **nc++** increments **nc** after its value has been used. If **nc** is **5**, then

```
x = nc++;
```

sets **x** to **5**, but

```
x = ++nc;
```

sets **x** to **6**. In both cases, **nc** becomes **6**. In a context where no value is wanted, just the increment effect, as in the count program, prefix and postfix are the same.

The character counting program accumulates its count in a **long** variable instead of an **int**, as **int** has a maximum value of 32 767, and it would take relatively little input to overflow an **int** counter. The conversion specification **%ld** tells **printf** that the corresponding argument is a **long** integer.

It may be possible to cope with even bigger numbers by using a **double**. We will also use a **for** statement instead of a **while**, to illustrate another way to write the loop.

count2.c

```
 1   #include <stdio.h>
 2   #include <conio.h>

 3   #define ESC '\x1B'

 4   /* count characters in input */

 5   main()
 6   {
 7     double nc;  /* number of characters */

 8     clrscr();
 9     for(nc = 0; getch() != ESC; ++nc)
10       ;
11     printf("%.0f character(s)\n", nc);
12   }
```

printf uses **%f** for both **float** and **double**; **%.0f** suppresses printing of the decimal point and the fraction part, which is zero.

The body of the **for** loop is empty, because all of the work is done in the test and increment parts. But the grammatical rules of C require that a **for** statement has a body. The isolated semicolon, called a **null statement**, is there to satisfy that requirement.

If the input contains no characters, the **while** or **for** test fails on the very first call of **getch**, and the program produces zero, the right answer. This is important. One of the nice things about **while** and **for** is that they test at the top of the loop, before proceeding the body. If there is nothing to do, nothing is done, even if that means never going through the loop body.

2.7. Arrays

To find the prime numbers between **2** and a given upper limit **MAX**, you can use an algorithm called the "Seven of Eratosthenes":

1. All natural numbers from **1** to **MAX** have to be listed.
2. Then all multiples of **2**, **3**, **5**, **7**, **11**, **13**, **17**, ..., **t** have to be deleted from the list, where **t** is the largest prime number $\leq \sqrt{\mathbf{MAX}}$.

Here is one version of the program:

prime1.c

```
 1  #include <stdio.h>
 2  #include <conio.h>
 3  #include <math.h>

 4  /* search primes using Eratosthenes */

 5  #define TRUE 1
 6  #define FALSE 0
 7  #define MAX 2000

 8  main()
 9  {
10    int prime[MAX], i, j;

11    clrscr();
12    for(i = 0; i < MAX; i++)
13      prime[i] = TRUE;
14    prime[0] = prime[1] = FALSE;
15    for(i = 2; i < sqrt(MAX); i++) {
16      j = i;
17      while((j += i) < MAX)
18        prime[j] = FALSE;
19    }
20    for(i = 0; i < MAX; i++)
21      if(prime[i])
22        printf("%5d", i);
23  }
```

The declaration

```
10    int prime[MAX];
```

declares **prime** to be an array of **MAX** (= 2000) integers. Array subscripts always start at zero in C, so the elements are **prime[0]**, **prime[1]**, …, **prime[MAX – 1]**. This is reflected in the **for** loops that initialize the array and print the output.

A subscript can be any integer expression, which includes integer variables like **i**, and integer constants.

The first version of the program does not use the knowledge about the multiples of the numbers that have already been deleted from the list.

prime2.c

```
1   #include <stdio.h>
2   #include <conio.h>
3   #include <math.h>

4   /* search primes using Eratosthenes */

5   #define TRUE 1
6   #define FALSE 0
7   #define MAX 2000

8   main()
9   {
10    int prime[MAX], i, j;

11    clrscr();
12    for(i = 0; i < MAX; i++)
13      prime[i] = TRUE;
14    prime [0] = prime[1] = FALSE;
15    for(i = 2; i < sqrt(MAX); i++) {
16      if(!prime[i])
17        continue;
18      j = i;
19      while((j += i) < MAX)
20        prime[j] = FALSE;
21    }
22    for(i = 0; i < MAX; i++)
23      if(prime[i])
24        printf("%5d", i);
25  }
```

The statement

```
17        continue;
```

is used to skip the multiples of numbers already deleted from the list. It causes the next loop iteration and is often used when reversion a test and indenting another level would nest the program too deeply. Lines 16 to 20 are equivalent to

```
16      if(prime[i]) {
17        j = i;
18        while((j += i) < MAX)
19          prime[j] = FALSE;
20      }
```

The **break** statement is related to **continue**; it provides an early exit from loops; a **break** causes the innermost enclosing loop to be exited immediately.

2.8. Character Arrays

The most common type of array in C is the array of characters, the **string**. The last character in a string is followed by a character `'\0'` to mark the end (the **null character**, whose value is zero; therefore this type of string is called the null terminated string). When a string constant like `"hello\n"` appears in a C program, it is stored as an array of characters containing the characters of the string and terminated with a `'\0'`.

Figure 2
Character array

'h'	'e'	'l'	'l'	'o'	'\n'	'\0'

The `%s` format specification in **`printf`** expects the corresponding argument to be a string represented in this form.

string1.c

```
1   #include <stdio.h>

2   main()
3   {
4     char name[5];  /* define a string of characters:
                        is an array of char */

5     name[0] = 'D';
6     name[1] = 'a';
7     name[2] = 'v';
8     name[3] = 'e';
9     name[4] = 0;    /* Null character - end of text */

10    clrscr();
11    printf("The name is %s\n", name);
12    printf("One letter is %c\n", name[2]);
13  }
```

There is a short form for the assignment of string constants, which automatically computes the size for the character array and puts the null character to its end.

string2.c

```
1   #include <stdio.h>

2   main()
3   {
4     char name[] = "Dave"; /* define a string of characters:
                               is an array of char */

5     clrscr();
6     printf("The name is %s\n", name);
7     printf("One letter is %c\n", name[2]);
8   }
```

2.9. Functions

In C, a function is equivalent to a subroutine or function in Fortran, or a procedure or function in Pascal. A function is used to modularize routines and to enable the programmer to reuse code as needed. With properly designed functions, it is possible to

ignore **how** a job is done; knowing **what** is done is sufficient. C makes the use of functions easy, convenient and efficient; you will often see a function defined and called only once, just because it clarifies some piece of code.

So far we have used only functions like **`printf`**, **`getch`**, and **`putchar`** that have been provided for us; now it's time to write a few of our own. Since C has no exponentiation operator like the **`**`** of Fortran, let us illustrate the mechanics of function definition by writing a function **`power(m, n)`** to raise an integer **`m`** to a positive integer power **`n`**. That is, the value of **`power(2, 5)`** is $2^5 = 32$. This function is not a practical exponentiation routine, since it handles only positive powers of small integers, but it's good enough for illustration. The standard library contains a function **`pow(x, y)`** that computes x^y.

Here is the function **`power`** and a main program to exercise it, so you can see the whole structure at once.

power1.c

```
 1  #include <stdio.h>

 2  int power(int m, int n);

 3  /* test power function */
 4  main()
 5  {
 6    int i;

 7    clrscr();
 8    for(i = 0; i < 10; i++)
 9      printf("%3d %7d %7d\n", i, power(2, i), power(-3, i));
10    return 0;
11  }

12  /* power: raise base m to n-th power; n >= 0 */
13  int power(int base, int n)
14  {
15    int i, p;

16    p = 1;
17    for(i = 1; i <= n; ++i)
18      p = p * base;
19    return p;
20  }
```

A function definition has this form:

```
return-type function-name(parameter declarations, if any)
{
  declarations
  statements
}
```

Function definitions can appear in any order, and in one source file or several, although no function can be split between files. For the moment, we will assume that all functions are in the same file. The function **`power`** is called twice by **`main`**, in the line

```
9       printf("%3d %7d %7d\n", i, power(2, i), power(-3, i));
```

Each call passes two arguments to **power**, which each time returns an integer to be formatted by **printf**. In an expression, **power(2, i)** is an integer just as **2** or **i**. (Not all functions produce an integer; a function may have any return type, as we will see later.)

The first line of **power** itself,

```
13  int power(int base, int n)
```

declares the parameter types and names, and the type of the result that the function returns. The names used by **power** for its parameters are local to **power**, and are not visible to any other function: other routines can use the same names without conflict. This is also true of the local variables **i** and **p**: the **i** in **power** is unrelated to the **i** in **main**. We will generally use **parameter** for a variable named in the parenthesized list in a function definition, and **argument** for the value used in a call of the function.

The value that **power** computes is returned to **main** by the **return** statement. Any expression may follow **return**:

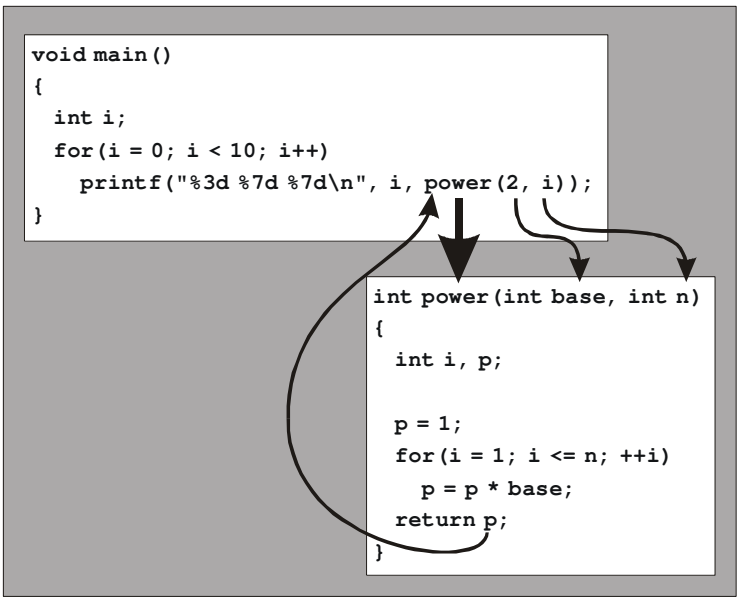
```
return expression;
```

A function need not return a value; a **return** statement with no expression causes the immediate termination of the function and return to the caller. And the calling function can ignore a value returned by a function.

You may have noticed that there is a **return** statement at the end of **main**. Since **main** is a function like any other, it may return a value to its caller, which is in effect the operating system. Typically, a return value of zero implies normal termination. In the interest of simplicity we have omitted **return** statements form our **main** functions up to this point.

If the return type of a function is omitted, **int** is assumed. The return type **void** explicitly states that no value is returned.

Figure 3
Function call and return value



The declaration

```
2   int power(int m, int n);
```

just before **main** says that **power** is a function that expects two **int** arguments and returns an **int**. This declaration, which is called **function prototype**, has to agree with the definition and uses of **power**. Parameter names need not agree and are optional. So for the prototype we could have written

```
2   int power(int, int);
```

A note of history: The biggest change between ANSI C and earlier versions is how functions are declared and defined. In the original definition of C, the **power** function would have been written like this:

historic.c

```
/* power: raise base m to n-th power; n >= 0; old style */
int power(base, n)
int base, n;
{
  int i, p;

  p = 1;
  for(i = 1; i <= n; ++i)
    p = p * base;
  return p;
}
```

The parameters are named between the parentheses, and their types are declared before the opening left brace; undeclared parameters are taken as **int**.

The declaration of power at the beginning of the program would have looked like this:

```
int power();
```

No parameter list was permitted, so the compiler could not readily check that **power** was being called correctly. Therefore we strongly recommend that you use the new form.

2.10. Arguments

2.10.1. Call by Value

In C, all function arguments are passed "by value". This means that the called function is given the values of its arguments in temporary variables rather than the originals. This leads to some different properties than are seen with "call by reference" languages like Fortran or with **`var`** parameters in Pascal, in which the called routine has access to the original argument, not a local copy.

The main distinction is that in C the called function cannot directly alter a variable in the calling function; it can only alter its private, temporary copy. Call by value usually leads to more compact programs, because parameters can be treated as conveniently initialized local variables in the called routine. For example, here is a version or power that makes use of this property.

```
power2.c
      12  /* power: raise base m to n-th power; n >= 0 */
      13  int power(int base, int n)
      14  {
      15    int p;

      16    for(p = 1; n > 0; --n)
      17      p = p * base;
      18    return p;
      19  }
```

The parameter **`n`** is used as a temporary variable, and is counted down until it becomes zero; there is no longer a need for the variable **`i`**. Whatever is done to **`n`** inside **`power`** has no effect on the argument that **`power`** was originally called with.

2.10.2. Call by Reference

When necessary, it is possible to arrange for a function to modify a variable in a calling routine. The caller must provide the **address** of the variable to be set (technically a **pointer** to the variable), and the called function must declare the parameter to be a

pointer and access the variable indirectly through it.

2.10.3. Arrays as Parameters

The story is different for arrays. When the name of an array is used as an argument, the value passed to the function is the location of address of the beginning of the array – there is no copying of array elements. By subscripting this value, the function can access and alter any element of the array.

We will use this to copy a string which is stored as character array.

```
strcopy1.c
        1  #include <stdio.h>
```

```
 2  void copy(char from[], char to[]);

 3  main()
 4  {
 5    char original[] = "String to copy";
 6    char clone[50] = "";

 7    clrscr();
 8    printf("The empty string (%s) ", clone);
 9    copy(original, clone);
10    printf("isn't empty any more: %s\n", clone);
11    return 0;
12  }

13  /* copy 'from' into 'to'; assume 'to' is big enough */
14  void copy(char from[], char to[])
15  {
16    int i;
17    for(i = 0; to[i] = from[i]; i++)
18      ;
19  }
```

The length of the arrays **`from`** and **`to`** is not necessary in **`copy`** since its size is set in main. **`copy`** writes the string from the source to the destination character array character by character. When the null character (which has the value zero) occurs, the **`for`** loop is terminated.

2.11. External Variables

The variables in **`main`**, such as **`original`**, **`clone`**, etc., are **private** or **local** to **`main`**. Because they are declared within **`main`**, no other function can have direct access to them. The same is true of the variables in other functions; for example, the variable **`i`** in **`copy`**. Each local variable in a function comes into existence only when the function is called, and disappears when the function is exited. This is why such variables are usually known as **automatic** variables.

Because automatic variables come and go with function invocation, the do not retain their values from one call to the next, and must be explicitly set upon each entry. If they are not set, they will contain garbage.

As an alternative to automatic variables, it is possible to define variables that are **external** to all functions. Because external variables are globally accessible, they can be used instead of argument lists to communicate data between functions. Furthermore, because external variables remain in existence permanently, they retain their values even after the function that set them have returned.

An external variable must be **defined**, exactly once, outside of any function; this sets aside storage for it. The variable must also be **declared** in each function that wants to access it; this states the type of the variable. The declaration may be an explicit **`extern`** statement or may be implicit from context. The extern statement can be omitted, if the definition of an external variable occurs in the source file before its use in a particular function.

`strcopy2.c`

```
 1  #include <stdio.h>

 2  char original[] = "String to copy";
 3  char clone[50] = "";

 4  void copy(void);

 5  main(void)
 6  {
 7    clrscr();
 8    printf("The empty string (%s) ", clone);
 9    copy();
10    printf("isn't empty any more: %s\n", clone);
11    return 0;
12  }

13  /* copy 'original' into 'clone' */
14  void copy(void)
15  {
16    int i;
17    for(i = 0; clone[i] = original[i]; i++)
18      ;
19  }
```

Since the specialized function **copy** has no arguments, logic would suggest that its prototype at the beginning of the file should be **copy()**. But for compatibility with older C programs the standard takes an empty list as an old-style declaration, and turns off all argument list checking; the word **void** must be used for an explicitly empty list.

Be careful using **extern** variable; it appears to simplify communications – argument lists are short and variables are always there when you want them. But external variables are always there even if you don't want them. Data connections are not all obvious, variables can be changed in unexpected ways, and the program is hard to modify. The second version of the string copy program is inferior to the first, partly for these reasons, and partly because it destroys the generality of the useful **copy** function by wiring into them the names of the variables they manipulate.

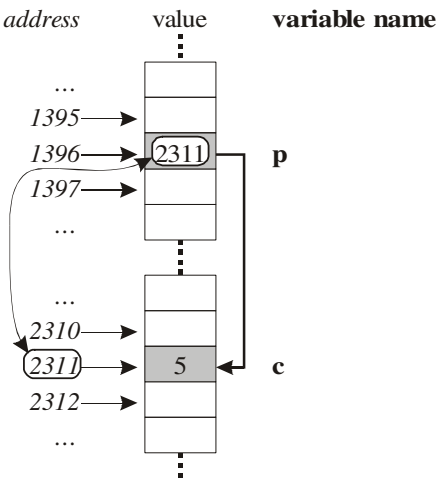
2.12. Pointers and Function Arguments

A pointer is a variable that contains the address of a variable.

2.12.1. Pointers and Addresses

A typical machine has an array of memory cells can be manipulated individually or in contiguous groups. Each memory cell can be addressed by an unique number. One common situation is that any byte can be a **char**; a pair of one-byte cells can be treated as **short** integer; and four bytes can be connected to a **long**. A pointer is a group of cells (often two or four) that can hold an address (16 bit or 32 bit addresses). So if **c** is a **char** and **p** is a pointer that points to it, we could represent the situation this way:

Figure 4
Address and value of variables



The unary operator `&` gives the address of an object (**address operator**), so the statement

```
p = &c;
```

assigns the address of `c` to the variable `p`, and `p` is said to "point to" `c`.

The unary operator `*` is the **indirection** or **dereferencing operator**; when applied to a pointer, it accesses the object the pointer points to.

The following artificial example shows how to declare a pointer and how to use `&` and `*`:

pointer1.c

```
1   #include <stdio.h>

2   int main(void)
3   {
4     int a, b, x[10];     /* integer variables */
5     int *ip;             /* ip is a pointer to int */

6     a = 250;
7     b = 300;
8     printf("\na = %d, b = %d", a, b);
9     ip = &a;
10    *ip = 1000;
11    printf("\na = %d, b = %d", a, b);
12    *ip = z[0];
13    return 0;
14  }
```

The **declaration** of the pointer `ip`,

```
int *ip;
```

is intended as mnemonic; it says that the expression `*ip` is an `int`. You should also note the implication that a pointer is constrained to a particular kind of object: every pointer points to a specific data type.

If `ip` points to the integer `a`, then `*ip` can occur in any context where `a` could, so

```
*ip = *ip + 10;
```

increments `*ip` by `10`.

The unary operators `*` and `&` bind more tightly than arithmetic operators, so the assignment

```
b = *ip + 1;
```

takes whatever `ip` points at, adds `1`, then assigns the result to `b`, while

```
*ip += 1;
```

increments what `ip` points to, as do

```
++*ip;
```

and

```
(*ip)++;
```

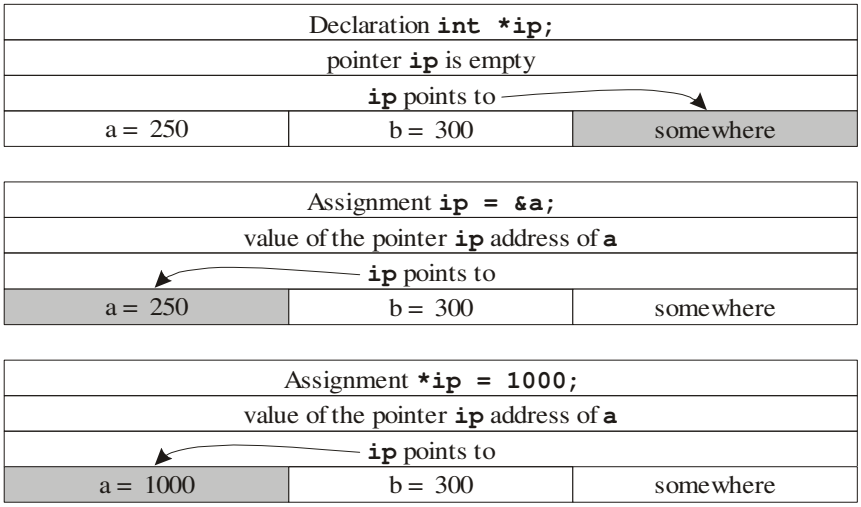
The parentheses are necessary in this last example; without them, the expression would increment `ip` instead of what it points to, because unary operators like `*` and `++` associate right to left.

Finally, since pointers are variables, they can be used without dereferencing. For example, if `iq` is another pointer to `int`,

```
iq = ip;
```

copies the contents of `ip` into `iq`, thus making `iq` point to whatever `ip` pointed to.

Figure 5
Example 10



2.12.2. Pointers and Function Arguments

Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function. For instance, a sorting routine might exchange two out-of-order elements with a function called `swap`. The left example will not work properly:

```
/* swap_not.c */
```

```
1  #include <stdio.h>
2  void swap(int a, int b);
3  void swap(int a, int b)
4  {
5    int t;
6    t = a;
7    a = b;
8    b = t;
9  }
10 int main(void)
11 {
12   int x, y;
13   x = 999;
14   y = 4;
15   printf("Before swap
   ");
16
     printf("%d  %d\n", x,
17 y);
18   swap(x, y);
19   printf("After swap
   ");
20
     printf("%d  %d\n", x,
21 y);
     return 0;
```

```
#include <stdio.h>
void swap(int *pa, int
*pb);
void swap(int *pa, int
*pb)
{
  int t;
  t = *pa;
  *pa = *pb;
  *pb = t;
}
int main(void)
{
  int x, y;
  x = 999;
  y = 4;
  printf("Before swap
");
  printf("%d  %d\n", x,
y);
  swap(&x, &y);
  printf("After swap
");
  printf("%d  %d\n", x,
y);
```

```
}
```

```
Before swap: 999  4
After swap:  999  4
```

```
Before swap: 999  4
After swap:  4  999
```

output:

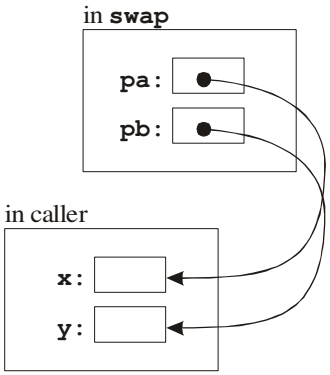
Because of call by value, `swap(int a, int b)` in the program `swap_not.c` can't effect the arguments `x` and `y` in the routine that called it. The function only swaps **copies** of `x` and `y`.

The way to obtain the desired effect is for the calling program to pass **pointers** to the values to be changed:

```
swap(&x, &y);
```

Since the operator `&` produces the address of a variable, `&x` is a pointer to `x`. In `swap` itself, the parameters are declared to be pointers, and the operands are accessed indirectly through them.

Figure 6
Example 11



2.12.3. Formatted Input `scanf`

The `scanf` function from standard library reads from the standard input (usually the keyboard) under control of format, and assigns converted values through subsequent arguments, **each of which must be a pointer**.

```
int scanf(const char *format, ...)
```

The format string may be built using the following conversion specifications:

`%d`	decimal integer; `int *`
`%i`	integer; `int *`; the integer may be octal (leading `0`) or hexadecimal
`%hd`	integer; `short *`
`%ld`	integer; `long *`
`%c`	characters; `char *`
`%s`	string of non-white space characters (not quoted); `char *`, pointing ıracters large enough to hold the string and a terminating `'\0'`.
`%f`	floating-point number; `float *`
`%lf`	floating-point number; `double *`

An input field is defined as a string of non-white space characters (blank, tab, newline, carriage return, vertical tab, and form feed).

scanf stops when it exhausts its format string, or when some input fails to match the control specification. It returns as its value the number of successfully matched and assigned input items. This can be used to decide how many items were found.

As a first example, a rudimentary calculator can be written:

calc1.c

```
1     #include <stdio.h>

2     int main(void)  /* rudimentary calculator */
3     {
4       double sum, x;
5       sum = 0;
6       while(scanf("%lf", &x) == 1)
7         printf("%20.2f\n", sum += x);
8       return 0;
9     }
```

Suppose we want to read input lines that contain dates of the form

```
25 Dec 2000
```

The **scanf** statement is

```
int day, year;
char monthname[20];
scanf("%d %s %d", &day, monthname, &year);
```

No **&** is used with **monthname**, since an array name is a pointer.

3. Types, Operators, and Expressions

Variables and constants are the basic data objects manipulated in a program. All variables must be declared, where their types and perhaps their initial values are specified. New values are produced in expressions, where operators specify what is to be done to variables and constants.

3.1. Variable Names

There are some restrictions on the names of variables and symbolic constants. Names are made up of **letters** and **digits**; the **first character** must be a letter. As letters count: `a`, `b`, …, `z`; `A`, `B`, …, `Z`; the underscore "_". Don't begin variable names with an underscore, since library routines often use such names. Upper case and lower case letters are distinct, so `x` and `X` are different names. Traditional C practice is to use lower case for variable names, and all upper case for symbolic constants.

At least the first **31** characters of an internal name are significant. For function names and external variables, the standard guarantees only for six characters and a single case, because external names may be used by assemblers and loaders.

You can't use the following 34 keywords as variable names (they are all lower case):

```
                    asm       auto      break     case      char
                    const
continue            default   do        double    else      enum
extern              float     for       fortran   goto      if
int                 long      register  return    short     signed
sizeof              static    struct    switch    typedef   union
unsigned            void      volatile  while
```

3.2. Data Types and Sizes

There are only a few **basic data types** in C:

- `char` is a single byte, capable of holding one character in the local character set.
- `int` is an integer, typically machine dependant.
- `float` single-precision floating point.
- `double` double-precision floating point.

In addition, there are a number of **qualifiers** that can be applied to these basic data types:

- `short` can be applied to `int`.
- `long` can be applied to `int` and `double`.
- `signed` and `unsigned` can be applied to `int` and `char`.

```
short int sh;
unsigned long int counter;
```

The word **int** can be omitted in such declarations.

short and **long** shall provide different lengths of integers. **int** will normally be the natural size for a particular machine. Each compiler is free to choose appropriate sizes for its own hardware, subject to the restrictions: **short** ≥ 16 bits, **int** ≥ 16 bits, **long** ≥ 32 bits, **short** ≤ **int** ≤ **long**.

The type **long double** specifies extended-precision floating point.

unsigned numbers are always positive or zero; if omitted the **int** will be declared **signed**.

The standard headers **<limits.h>** and **<float.h>** contain symbolic constants for the maximum and minimum values of all numeric data types. When using Borland C with MS DOS the limits are:

data type		minimum	maximum
unsigned char	8 Bits	0	255
char	8 Bits	−128	127
unsigned int	16 Bits	0	65 535
short int	16 Bits	−32 768	32 767
int	16 Bits	−32 768	32 767
unsigned long	32 Bits	0	4 294 967 295
long	32 Bits	−2 147 483 648	2 147 483 647
float	32 Bits	$3.4 \cdot 10^{-38}$	$3.4 \cdot 10^{+38}$
double	64 Bits	$1.7 \cdot 10^{-308}$	$1.7 \cdot 10^{+308}$
long double	80 Bits	$3.4 \cdot 10^{-4932}$	$1.1 \cdot 10^{+4932}$

3.3. Constants

3.3.1. Numerical Constants

The data type of a constant is determined by the compiler. Using suffixes you can force a specific type.

Floating-point constants contain a decimal point (**123.4**) or an exponent (**1e-2**) or both (**123.4E73**). Their type is double, unless suffixed: **f** or **F** indicate a **float**, **l** or **L** define a **long double**.

An **integer constant** will be **int**. An integer too big to fit into an **int** will be taken as **long**. The suffix **l** or **L** will produce a **long** constant, a terminal **u** or **U** indicate an **unsigned int**, and **ul** or **UL** are suffixes for **unsigned long**.

```
1234  /* int */
1234L /* long */
80000 /* long */
157ul /* unsigned long */
```

The value of an integer constant can be specified in octal or hexadecimal instead of decimal. A leading zero (`0`) on an integer constant means octal; a leading `0x` or `0X` means hexadecimal. For example, decimal `31` can be written as `037` in octal and `0x1f` or `0X1F` in hex. As all suffixes for integer constants may be used, `0XFUL` is an `unsigned long` constant with value `15` decimal.

A **character constant** is an integer value (type `char`), written as one character within single quotes (`'x'`). The value of a character constant is the value of the character in the machine's character set. For example, in the ASCII character set (which is used by MS DOS) the value of `'x'` is `48`.

Certain characters can be represented in character and string constants by **escape sequences**:

`\a` alert (bell) character	`\b` backspace	`\f` form feed
`\n` newline	`\r` carriage return	`\t` horizontal tab
`\v` vertical tab	`\\` backslash	`\?` question mark
`\'` single quote	`\"` double quote	

Any character constant can be specified by `'\`ooo`'`, where ooo is one to three octal digits (0 ... 7), or by `'\x`hh`'`, where hh is on or more hexadecimal digits (`0` ... `9`, `a` ... `f`, `A` ... `F`).

The ASCII character set specifies character 7 to be the bell character. So `'\a'`, `'\7'`, `'\007'` and `'\x7'` stand for the same character constant.

3.3.2. Constant Expressions

A **constant expression** is any expression that involves only constants. Such expressions can be evaluated during compilation rather than run-time. So they can be used in any place that a constant can occur.

```
#define MAXLINE 1000
char line[MAXLINE + 1]
```

3.3.3. String Constants

A **string constant**, or **string literal**, is a sequence of zero or more characters surrounded by double quotes, as in

```
"I am a string"
```

or

```
""  /* the empty string */
```

The quotes are not part of the string. The same escape sequences as in character constants can be used in string constants. String constants con be concatenated at compile time:

```
"Hello,"
```

```
    " world"
```

is equivalent to

```
    "Hello, world"
```

Technically, a string constant is an array of characters. At the end each string has a null character `'\0'`, so the physical storage required is one more than the number of characters written between the quotes.

You have to distinguish between a character constant and a string constant that contains only one character: `'x'` is an integer, whereas `"x"` is an array of characters that contains one character (the letter `x`) and a `'\0'`.

3.3.4. Enumerations

There is another kind of constant, the **enumeration constant**. An enumeration is a list of constant integer values, as in

```
    enum boolean { NO, YES };
```

The first name in an `enum` has value `0`, the next `1`, and so on, unless explicit values are specified.

```
    enum escapes { BELL = '\a', BACKSPACE = '\b',
  NEWLINE = '\n', RETURN = '\b', ONE = '\1' };

    enum months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG,
      SEP, OCT, NOV, DEC };     /* FEB is 2, MAR is 3, etc. */
```

Names in different enumerations must be distinct. Values need not be distinct in the same enumeration.

3.4. Declarations

All variables must be declared before use whereby the data type is specified.

```
    int lower, upper, step;
    char c, line[1000];
```

is equal to

```
    int lower;
    int upper;
    int step;
    char c;
    char line[1000];
```

A variable may be initialized in its declaration:

```
    char esc = '\\';
    int i = 0
    int limit = MAXLINE+1;
```

If a variable is not automatic (but external or static), the initialization is done once only, usually before the program starts; the initializer must be a constant expression. An explicitly initialized automatic variable is initialized each time the block it is in is entered, otherwise it has an undefined value (i.e., garbage); the initializer may be any expression. External and static variables are initialized to zero by default.

```
static int init_once = 10;
```

When the size of an array is omitted, the compiler will compute the length by counting the initializers. For example, the initialization of the array days with the number of days in each month:

```
int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30,
  31 };            /* equal to days[12] = ... */
```

Character arrays may be initialized using a string:

```
char line[] = "test";
```

is a shorthand for:

```
char line[] = { 't', 'e', 's', 't', '\0' };
```

In this case the array size is five (four characters plus the terminating `'\0'`).

The qualifier const in a variable declaration specifies that its value will not be changed:

```
const double e = 2.71828182845905;
const char msg[] = "warning: ";
```

3.5. Arithmetic Operators

The binary arithmetic operators are `+`, `–`, `*`, `/`, and the modulus operator `%`. The binary operators `+` and `–` have the same precedence, which is lower then the precedence of `*`, `/`, and `%`, which is in turn lower than unary `+` and `–` (e.g. `–5 * 20` evaluates as `–100`).

3.6. Relational and Logical Operators

The relational operators are `>`, `>=`, `<`, and `<=`; they all have the same precedence. Just below them in precedence are the equality operators `==` (equal) and `!=` (not equal).

Relational operators have lower precedence than arithmetic operators, so an expression like `i < lim–1` is taken as `i < (lim – 1)`, as would be expected.

The logical operators are `&&` and `||`. Expressions connected by `&&` or `||` are evaluated left to right, and evaluation stops as soon as the truth or falsehood of the result is known. For example:

```
for (i=0; i<lim-1 && (c=getchar()) != '\n' && c!=EOF; ++i)
  s[i] = c;
```

Before reading a new character it is necessary to check that there is room to store it in the array `s`, so the test `i<lim–1` **must** be made first. Moreover, if this test fails, we **must not** go on and read another character. Similarly, it would be unfortunate if `c` were tested against `EOF` before `getchar` is called. The precedence of `&&` is higher than that of `||`, and both are lower than relational and equality operators, so no extra parentheses is needed. But since the precedence of `!=` is higher than assignment, parentheses are needed in

```
(c=getchar()) != '\n'
```

to achieve the desired result of assignment to `c` and then comparison with `'\n'`.

By definition, the numeric value of a relational or logical expression is `1` if the relation is `true`, and `0` if the relation is `false`.

The unary operator `!` converts a non-zero operand into 0, and a zero operand into 1. A common use of `!` is in constructions like

```
if(!valid)
```

which reads nicely ("if not valid") rather than

```
if(valid == 0)
```

3.7. Type Conversions

When an operator has operands of different types, they are converted to a common type, which is also the type of the result, according to a small number of rules. In general, the only automatic conversions are those that convert a "narrower" ("smaller") operand into a "wider" ("larger") one without losing information, such as converting an integer to floating point in an expression like `f + i`. Expressions that might lose information, like assigning a longer integer type to a shorter, may draw a warning, but are not illegal.

Explicit type conversions can be forced ("coerced") in any expression, with a unary operator called a **cast**. In the construction

```
(type-name) expression
```

the *expression* is converted to the named type. So if `n` is an integer, we can use

```
sqrt((double) n)
```

to convert the value of `n` to `double` before passing it to `sqrt`. If arguments are declared by a function prototype, as they normally should be, the declaration causes automatic coercion of any arguments when the function is called. Thus, given a function prototype for `sqrt`:

```
double sqrt(double);
```

the call

```
root2 = sqrt(2);
```

coerces the integer `2` into the `double` value `2.0` without any need for a cast.

3.8. Increment an Decrement Operators

The increment operator `++` adds `1` to its operand, while the decrement operator `--` subtracts `1`. `++` and `--` may be used as prefix operators (before the variable, as in `++n`), or postfix (after the variable: `n++`). In both cases, the effect is to increment `n`. But the expression `++n` increments `n` before its value is used, while `n++` interments `n` after its value has been used. This means that in a context where the value is being used, not just the effect, `++n` and `n++` are different. If `n` is `5`, then

```
x = n++;
```
sets **x** to **5**, but

```
x = ++n;
```
sets **x** to **6**. In both cases, **n** becomes **6**.

3.9. Bitwise Operators

C provides six operators for bit manipulation, which may be applied to integral operands:

- **&** bitwise AND
- **|** bitwise inclusive OR
- **^** bitwise exclusive OR
- **<<** left shift
- **>>** right shift
- **~** one's complement (unary)

The bitwise AND operator **&** is often used to mask off some set of bits; for example,

```
n = n & 0x7F;
```
sets to zero all but the low-order 7 bits of **n**.

The bitwise OR operator **|** is used to turn bits on:

```
n = n | SET_ON;
```
sets to one in **x** the bits that are set to one in **SET_ON**.

One must distinguish the bitwise operators **&** and **|** from the logical operators **&&** and **||**, which imply left-to-right evaluation of a truth value. For example, if **x** is **1** and **y** is **2**, then **x & y** is zero while **x && y** is one.

The shift operators << and >> perform left and right shifts of their operand by the number of bit positions given by the right operand, which must be positive. Thus **x << 2** shifts the value of **x** left by two positions, filling vacated bits with zero; this is equivalent to multiplication by **4**.

The unary operator **~** yields the one's complement of an integer; that is, it converts each 1-bit into a 0-bit and vice versa. For example,

```
x = x & ~ 0x3F;
```
sets the last six bits of **x** to zero.

The following function **getbits(x, p, n)** returns the (right adjusted) **n**-bit field of **x** that begins at position **p**. For example, **getbits(x, 4, 3)** returns the three bits in bit position **4**, **3**, and **2**, right adjusted.

```
/* getbits: get n bits from position p*/
unsigned getbits(unsigned x, int p, int n)
{
  return (x >> (p+1-n)) & ~(~0 << n);
```

```
}
```

3.10. Assignment Operators and Expressions

Expressions such as

```
i = i + 2;
```

in which the variable on the left hand side is repeated immediately on the right, can be written in the compressed form

```
i += 2;
```

The operator **+=** is called an assignment operator. Most binary operators have a corresponding assignment operator *op*=, where *op* is one of:

```
+   -   *   /   %   <<   >>   &   ^   |
```

If $expr_1$ and $expr_2$ are expressions, then

expr₁ op= expr₂

is equivalent to

expr₁ = (expr₁) op (expr₂)

except that $expr_1$ is only computed once. Notice that the parentheses around $expr_2$:

```
x *= y + 1;
```

means

```
x = x * (y + 1);
```

rather than

```
x = x * y + 1;
```

3.11. Conditional Expressions

The statements

```
if(a > b)
  z = a;
else
  z = b;
```

compute in **z** the maximum of **a** and **b**. The conditional expression, written with the ternary operator "**?:**", provides an alternate way to write this and similar constructions. In the expression

expr₁ ? expr₂ : expr₃

the expression $expr_1$ is evaluated first. If it is non-zero (**true**), then the expression $expr_2$ is evaluated, and that is the value of the conditional expression. Otherwise $expr_3$ is evaluated, and that is the value. Only one of $expr_2$ and $expr_3$ is evaluated. Thus to set **z** to the maximum of **a** and **b**,

```
z = (a > b) ? a : b;    /* z = max(a, b); */
```

Another good example is

```
printf("You have %d item%s.\n", n, n==1 ? "" : "s");
```

3.12. Precedence and Order of Evaluation

Table 1 summarizes the rules for precedence of all operators, including those that we have not discussed yet. Operators in the same line have the same precedence; rows are in order of decreasing precedence.

The "operator" `()` refers to function call. The operators `->` and `.` are used to access members of structures; `sizeof` yields the size of an object.

Table 1

•	`()`	`[]`	`->`	`.`							
•	`!`	`~`	`++`	`--`	`+`	`-`	`*`	`&`	(*type*)	`sizeof`	(unary)
•	`*`	`/`	`%`								(binary)
•	`+`	`-`									(binary)
•	`<<`	`>>`									
•	`<`	`<=`	`>`	`>=`							
•	`==`	`!=`									
•	`&`										
•	`^`										
•	`|`										
•	`&&`										
•	`||`										
•	`?:`										
•	`=`	`+=`	`-=`	`*=`	`/=`	`%=`	`&=`	`^=`	`|=`	`<<=`	`>>=`
•	`,`										

4. Control Flow

4.1. Statements and Blocks

An **expression** such as `x = 0` of `i++` or `printf(…)` becomes a **statement** when it is followed by a semicolon, as in

```
        x = 0;
          i++;
    printf(…);
```

In C, the demicolon is a statement terminator.

Braces `{` and `}` are used to group declarations and statements together into a **compound statement**, or **block**, so that they are syntactically equivalent to a single statement. There is no semicolon after the right brace that ends a block.

4.2. If-Else

The general form is

```
if(expression)
     statement1
          else
     statement2
```

One and only one of the two statements associated with an `if-else` is performed. If the *expression* is true, $statement_1$ is executed; if not, $statement_2$ is executed. Each *statement* can be a single statement or several in braces. By the way, notice that there is a semicolon after the expression in the if case. This is because grammatically, a statement follows the `if`, and a statement is always terminated by a semicolon.

The pattern

```
 if(condition1)
     statement1
else if(condition2)
     statement2
               …
               …
          else
     statementn
```

expresses a multi-way decision.

4.3. Switch

The `switch` statement is a multi-way decision that tests whether an expression matches one of a number of **constant** integer values, and branches accordingly.

```
switch(expression) {
case const-expr:
     statements
               …
```

```
            …
      default:
    statements
            }
```

Each case is labeled by one or more integer-valued constants or constant expressions. If a case matches the expression value, execution starts at that case. All case expressions must be different. The case labeled **default** is executed if none of the other cases are satisfied. A **default** is optional; if it isn't there and if none of the cases match, no action at all takes place. Cases and the **default** clause can occur in any order.

The **break** statement causes an immediate exit from the switch. Because cases serve just as labels, after the code for one case is done, execution falls through to the next unless you take explicit action to escape! With the exception of multiple labels for a single computation, fall-throughs should be used sparingly, and commented, because they are not very robust, being lost when the program is modified. As a matter of good form, put a **break** after the last case even though it's logically unnecessary. Some day when another case gets added at the end, this bit of defensive programming will save you.

The following example program waits for user inputs and counts the digits, white spaces and all other characters.

switch.c

```
1#include <stdio.h>
2#include <conio.h>

3/* count digits, white spaces, others */

4int main(void)
5           {
6  const char ESC = '\x1B';
7  int ndigits, nwhites, nothers;
8      char c;
9  enum digits {ONE = '1', TWO, THREE, FOUR, FIVE, SIX,
10               SEVEN, EIGHT, NINE};
11  ndigits = nwhites = nothers = 0;
12  while((c = getch()) != ESC) {
13    switch(c) {
14    case '0': case ONE: case TWO: case THREE: case FOUR:
15    case FIVE: case SIX: case SEVEN: case EIGHT: case NINE:
16      ndigits++;
17      break;
18    case ' ':
19    case '\n':
20    case '\r':
21    case '\t':
22      nwhites++;
23      break;
24    default:
25      nothers++;
26      break;
27           }
28           }
29  printf("%d digits, %d whitespaces, and %d other "
30         "characters\n", ndigits, nwhites, nothers);
31   return 0;
32           }
```

4.4. Loops

4.4.1. For

The **for** statement

for(*initialisation*; *condition*; *increment step*)
**   *statement***

is a loop. Within the parentheses, there are three parts, separated by semicolons. The first part, the *initialisation* is done once, before the loop proper is entered. The second part is the *test* or *condition* that controls the loop. It is evaluated and if it is true, the body of the loop (the *statement*) is executed. Then the *increment step* is executed, and the *condition* re-evaluated. The loop terminates if the *condition* has become false. The body of the loop can be a single *statement*, or a *group of statements* enclosed in braces. The *initialization*, *condition*, and *increment* can be any expressions.

The function **strcount** calculates the length of a string:

```
strcount.c
1int strcount(char str[])
2            {
3       int i;
4  for(i = 0; str[i]; i++)
5            ;
6    return i;
7            }
```

Any of the three parts, *initialisation*, *condition*, and *increment step*, can be omitted, although the semicolons must remain. If *initialisation* or *increment step* is omitted, it is simply dropped form the expansion. If the *test* is not present, it is taken as permanently true, so

```
    for(;;) {
            …
            }
```

is an "infinite" loop, presumably to be broken by other means, such as a **break** or **return**.

4.4.2. While

In

while(*expression*)
**   *statement***

the *expression* is evaluated. If it is non-zero, *statement* is executed and *expression* is re-evaluated. This cycle continues until *expression* becomes zero, at which point execution resumes after *statement*. The body of the loop can be a single *statement*, or a *group of statements* enclosed in braces. So,

```
   while(1) {
            …
            }
```

is an "infinite" loop

4.4.3. Do-While

As we discussed, the **while** and **for** loops test the termination condition at the top. By contrast, the third loop in C, the **do-while**, tests at the **bottom after** making each pass through the loop body; the boy is always **executed at least once**.

The syntax of the **do** is

```
          do
     statement
while(expression);
```

The *statement* is executed, then *expression* is evaluated. If it is true (non-zero), *statement* is executed again, and so on.

We will use this to copy a string which is stored as character array. Please mention, that also the terminating 0 will be copied.

```
1/* copy 'from' into 'to'; assume 'to' is big enough */
2void copy(char from[], char to[])
3           {
4   int i = 0;
5           do
6    to[i++] = from[i];
7  while(from[i]);
8           }
```

4.4.4. Break and Continue

It is sometimes convenient to be able to exit from a loop other than by testing at the top or bottom. The **break** statement provides an early exit from **for**, **while**, and **do**, just as from **switch**. A **break** causes the innermost enclosing loop or **switch** to be exited immediately.

The following function, **trim**, removes trailing blanks, tabs, and newlines from the end of a string, using a **break** to exit from a loop when the rightmost non-blank, non-tab, non-newline is found.

```
trim.c
1int trim(char s[])
2           {
3       int n;

4  for(n = strlen(s)-1; n >= 0; n--)
5    if(s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
6       break;
7  s[n+1] = '\0';
8  return n + 1;
9           }
```

strlen returns the length of the string.

The **continue** statement is related to **break**. It causes the next iteration of the enclosing **for**, **while**, or **do** loop to begin. In the **while** and **do**, this means that the *test* part is executed immediately; in the **for**, control passes to the *increment step*.

As an example, this fragment processes only the non-negative elements in the array **a**; negative values are skipped.

```
for(i = 0; i < 1000; i++) {
  if(a[i] < 0) /* skip negative elements */
     continue;
  … /* do positive elements */
             }
```

5. Functions and Program Structure

5.1. Basics of Functions

Each **function definition** has the form

```
return-type function-name(parameter declarations, if any)
{
   declarations
   statements
}
```

Various parts may be absent; a minimal function is

```
dummy() {}
```

which does nothing and returns nothing. If the return type is omitted, `int` is assumed.

The functions can occur in any order in the source file, and the source program can be split into multiple files, so long as no function is split. If a name that has not been previously declared occurs in an expression and is followed by a left parenthesis, it is declared by context to be a function name, the function is assumed to return an `int`, and nothing is assumed about its arguments; all parameter checking is turned off! Therefore you should declare a function which does not return a value and has no parameters in the following way (**function declaration**):

```
void function-name(void);
```

A function which has no return value but uses parameters should be declared as follows:

```
void function-name(parameter declarations);
```

A function call has the general form:

```
return-type var;
var = function-name(parameters);
```

A program is just a set of definitions of variables and functions. Communication between the functions is by arguments and values returned by the functions, and through external variables. The `return` statement is the mechanism for returning a value from the called function to its caller. Any expression can follow `return`:

```
return expression;
```

The calling function is free to ignore the returned value. Furthermore there need be no expression after `return`; in that case, no value is returned to the caller. Control also returns to the caller with no value when execution "falls off the end" of the function by reaching the closing right brace.

5.2. Program Structure

The basic program structure looks like this:

```
/* comment */
```

```
#include <header.h>
#include "another_header.h"
#define identifier token-sequence
external variables
function declarations (prototypes)
function definitions
int main()
{
  ...
}
```

More information you will find in chapter "**Fehler! Verweisquelle konnte nicht gefunden werden.**", p. **Fehler! Textmarke nicht definiert.**.

6. Structures and Type Definitions

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. Structures help to organize complicated data, because they permit a group of related variables to be treated as a unit instead of separate entities.

One example of a structure is an entry in an address database: a contact is described by a set of attributes such as name, address, email address, etc. Another example, more typical for C, comes from graphics: a point is a pair of coordinates, a rectangle is a pair of points, and so on.

6.1. Basics of Structures: Definition and Use

The two coordinates of a point, both integer, can be placed in a structure declared like this:

```
struct point {
        int x;
        int y;
            };
```

The keyword **`struct`** introduces a structure declaration, which is a list of declarations enclosed in braces. The *structure-tag* in

```
struct structure-tag {
   declarations
            };
```

is optional; this tag names the structure and can be used subsequently as a shorthand for the part of the declaration in braces. The variables named in a structure are called **members**.

A **`struct`** declaration defines a type. The right brace that terminates the list of members may be followed by a list of variables, just as for any basic type. Using the syntactical analogy from

```
  int x, y, z;
```

we can write:

```
struct { … } x, y, z;
```

in the sense that each statement declares **`x`**, **`y`**, and **`z`** to be variables of the named type and causes space to be set aside for them.

If the declaration is tagged, as in the declaration of **`point`** above, the tag can be used later in definition of instances (variables) of the structure:

```
struct point pt;
```

defines a variable **`pt`** which is a structure of type **`struct point`**. A structure can be initialized by following its definition with a list of initializers, each a constant expression, for the members:

```
struct point maxpt = { 320, 200 };
```

A member of a particular structure is referred to in an expression by a construction of the form

structure-name*.*member

The structure member operator "." connects the structure name and the member name. To print the coordinates of the point pt, for instance,

```
printf("(%d/%d)", pt.x, pt.y);
```

or to compute the distance from the origin (0/0) to **pt**,

```
  double dist;
dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```

Of course it is possible to define a pointers to structures, for example a pointer to the **point** structure declared above:

```
struct point *ppt;
    ppt = &pt;
```

To use **ppt**, we might write, for example:

```
(*ppt).x = 75;
```

The parentheses are necessary in **(*ppt).x** because the precedence of the structure member operator . is high than *. The expression ***ppt.x** means ***(ppt.x)** which is illegal here because **x** is not a pointer. Pointers to structures are so frequently used that an alternative notation is provided as a shorthand:

pointer_to_structure-name*->*member

refers to the particular member. So we could write instead:

```
 ppt->y = 150;
```

C provides a compile-time unary operator called **sizeof** that can be used to compute the size of any object. The expressions

```
  sizeof object
```

and

```
sizeof(type-name)
```

yield an integer equal to the size of the specified object or type in bytes. Don't assume however, that the size of a structure is the sum of the sizes of its members. Because of alignment requirements for different objects, there may be unnamed "holes" in a structure. Thus, for instance, if a **char** is one byte and an **int** four bytes, the structure

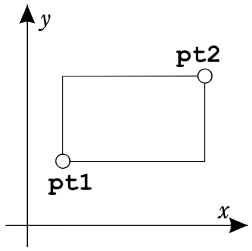
```
struct { char c; int i; };
```

might well require eight bytes, not five.

6.2. Nested Structures

Structures can be nested. One representation of a rectangle (which sides are parallel to the coordinate axes) is a pair of points that denote the diagonally opposite corners:

Figure Rectangle 7




```
struct rect {
  struct point pt1;
  struct point pt2;
            };
```

The **rect** structure contains two **point** structures. If we declare **rectangle** as

```
struct rect rectangle;
```

then

```
rectangle.pt1.x = 50;
```

refers to the **x** coordinate of the **pt1** member of **rectangle**.

An initialization with constant values can be done following the definition:

```
struct rect screen = {
                        {0, 0},
                        {640, 480}
                      };
```

6.3. Arrays of Structures

To store coordinates of a polygon we need an array to store the x and an array to store the y coordinates. One possibility is to use two parallel arrays, **x** and **y**, as in

```
int x[MAX_POINTS];
int y[MAX_POINTS];
```

But the very fact that the arrays are parallel suggests a different organization, an array of structures. Each related x and y coordinates entry is a point, and there is an array of points:

```
struct point {int x; int y;} point_tab[MAX_POINTS];
```

This line declares a structure type **point**, defines an array **point_tab** of **MAX_POINTS** structures of this type, and sets aside storage for them. Each element of the array is a structure.

Code to assign values to the members looks like

```
point_tab[n].x = 45;
```

where **n** is an integer.

The structure initialization is analogous to earlier ones – the definition is followed by a list of initializers enclosed in braces:

```
struct point {
        int x;
        int y;
} quadrat[] = {
     {10, 10},
     {10, 20},
     {20, 20},
     {20, 10},
      {10, 10}
           };
```

To get the number of entries in this array (which is counted at compile-time) you can use the **sizeof** operator, as the number of entries is just the (size of **quadrat** / size of **struct point**). This computation is used in a **#define** statement to set the value of **N_POINTS**:

```
#define N_POINTS (sizeof quadrat / sizeof(struct point))
```

Another way to write this is to divide the array size by the size of a specific element:

```
#define N_POINTS (sizeof quadrat / sizeof quadrat[0])
```

6.4. Structures and Functions

The only legal operations on a structure are copying it or assigning to it as a unit, taking its address with **&**, and accessing its members. Copy and assignment include passing arguments to functions and returning values from functions as well. Structures may not be compared. A structure may be initialized by a list of constant member values; an automatic structure may also be initialized by an assignment.

There are at least three possible approaches to pass structures form a function to another: pass components separately, pass an entire structure, or pass a pointer to it.

In the following example we will use a point structure for "named points", as follows:

struct.c

```
1#include <stdio.h>
2#define NAME_LEN 50

3/* structure point */
4struct point {
5  int x;       /* x coordinate */
6  int y;       /* y coordinate */
7  char name[NAME_LEN];
8           };
```

The first function, **makepoint**, will take two integers and a name and return a **point** structure:

```
9struct point makepoint(int x, int y, char name[])
10          {
11  point temp;

12  temp.x = x;
13  temp.y = y;
14  strcpy(temp.name, name);
15  return temp;
16          }
```

Notice that there is no conflict between the argument name an the member with the same name; indeed the re-use of the names stresses the relationship. **makepoint** can now be used to initialize any structure dynamically, or to provide structure arguments to a function:

```
struct point pt1, pt2, pt3, middle;
pt1 = makepoint(0, 0, "origin");
pt2 = makepoint(XMAX, YMAX, "maximum");
    pt3 = pt2;
middle = makepoint(pt2.x / 2, pt2.y / 2, "middle");
```

Very practical for testing purposes is a function to print a "point" to the screen:

```
17void printpoint(struct point p)
18          {
19  printf("(%d / %d) %s\n", p.x, p.y, p.name);
20          }
```

The next step is a set of functions to do arithmetic on points. For instance,

```
21struct point addpoint(struct point p1, struct point p2)
22          {
23  /* only a to addpoint local copy of p1 is changed! */
24  p1.x += p2.x;
25  p1.y += p2.y;
26  /* append names, but be carful about maximum length */
27  strncat(p1.name, " + ", NAME_LEN - strlen(p1.name) -1);
28  strncat(p1.name, p2.name, NAME_LEN - strlen(p1.name) -1);
29  return p1;
30          }
```

Here both the arguments and the return value are structures. We incremented the components in **p1** rather than using a temporary variable to emphasize that structure parameters are passed by value like any others.

```
pt3 = addpoint(pt1, pt2);
printpoint(pt1);
printpoint(pt2);
printf("--------"\n);
printpoint(pt3);
printf("========"\n);
```

If a large structure has to be passed to a function, it is generally more efficient to pass a pointer than to copy the whole structure.

```
31struct point *movepoint(struct point *pp, int dx, int dy)
32          {
33  (*pp).x += dx; /* note: is equivalent to: pp->x += dx; */
34  pp->y += dy;
35  return pp;
36          }
```

In **movepoint** the point is given as a parameter by reference, so we can change its value. The return value is a pointer to the given point which can be used to produce compact code sometimes.

```
printpoint(*movepoint(p1, 10, 15));
```

6.5. Type Definitions

C provides a facility called **typedef** for creating new data type names. For example, the declaration

```
typedef int Length;
```

makes the name **Length** a synonym for **int**. The type **Length** can be used in declarations, casts, etc,. In exactly the same ways that the type **int** can be:

```
Length len, maxlen, lengths[20], *plen, *plengths[];
```

Similarly, the declaration

```
typedef int *String;
```

makes **String** a synonym for **char*** or character pointer, which may then be used in declarations and casts.

Notice that the type being declared in a **typedef** appears in the position of a variable name, not right after the word **typedef**:

typedef ***type typedef-name***;

We can use **typedef** to declare a structure as a new type, for instance:

```
typedef struct point *PointPtr;
typedef struct point {
  int x;       /* x coordinate */
  int y;       /* y coordinate */
  char name[NAME_LEN];
     } Point;
```

This creates two new type keywords called **Point** (a structure) and **PointPtr** (a pointer to the structure). Then the functions to handle points may be declared as follows:

```
9aPoint makepoint(int x, int y, char name[]);
17avoid printpoint(Point p);
21aPoint addpoint(Point p1, Point p2);
31aPointPtr movepoint(PointPtr pp, int dx, int dy);
```

A variable point is defined by:

```
   Point pt1;
```

A pointer to a point can be defined using

```
   Point *pp;
```

which is equivalent to

```
 PointPtr pp;
```

6.6. Bit-fields

When storage space (or communication time) is at a premium, it may be necessary to pack several objects into a single machine word; one common use is a set of single-bit flags in digital control using programmable logic controllers (PLCs). Here you often require the ability to get at pieces of a word (for example one single bit).

Imagine a control program that wants to test, if a single digital input is set or not. The usual way this is done is to define a set of "masks" corresponding to the relevant bit positions, as in

```
#define START_BUTTON 01
#define STOP_BUTTON  02
#define OIL_SENSOR   04
#define MOTOR_RUN    08
```

or

```
enum { START_BUTTON=01, STOP_BUTTON=02, OIL_SENSOR=04 };
```

The numbers must be powers of two. Then accessing the bits becomes a matter of "bit-fiddling" with the shifting, masking and complementing operators, for example

```
if(flags & START_BUTTON) /* START_BUTTON is pressed */
  flags |= MOTOR_RUN;    /* start the engine        */
if(flags & STOP_BUTTON)  /* STOP_BUTTON is pressed  */
  flags &= ~MOTOR_RUN;   /* stop the engine         */
```

C offers the capability of defining and accessing fields within a word directly rather than by bitwise logical operators. A bit-field is a set of adjacent bits within a single storage unit that we will call a "word". The syntax of field definition and access is based on structures. For example, the symbol table **#define**s above could be replaced by the definition of four fields:

```
     struct {
unsigned int is_start_button : 1;
unsigned int is_stop_button  : 1;
unsigned int is_oil_sensor   : 1;
unsigned int set_motor_run   : 1;
     } flags;
```

This defines a variable called **flags** that contains four 1-bit fields. The number following the colon represents the field width in bits. The fields are declared **unsigned int** to ensure that they are unsigned quantities.

Individual fields are referenced in the same way as other structure members. Thus the previous examples may be written more naturally as:

```
if(flags.is_start_button)  /* START_BUTTON is pressed */
  flags.set_motor_run = 1; /* start the engine        */
if(flags.is_stop_button)   /* STOP_BUTTON is pressed  */
  flags.set_motor_run = 0; /* stop the engine         */
```

Fields may be declared only as **int**s. They are not arrays, and the do not have addresses, so the **&** operator cannot applied to them.

7. Literature

Kernighan/Ritchie, "Programmieren in C", Hanser, ISBN 3-446-15497-3