1. A Crash Course in C

This appendix gives a crash course in the C programming language. The story of C began in the late sixties with the Multics project, which was a predecessor to the UNIX operating system. The first two versions of Multics were written in the assembly language. However, since programming in assembly languages was rather cumbersome, an appropriate high-level language was called for.

The researchers Dennis Ritchie and Ken Thompon created the language B by simplifying the research language BCPL[[1]](#footnote-1) in 1969. The new versions, called New B, were introduced in 1972. The name did soon evolve into C. In 1978, Brian Kernighan and Dennis Ritchie wrote the book The C Programming Language, which introduced the informal K&R C standard, named after the authors. In 1983, a working group was formed by the American National Standard Institute (ANSI) in order to define a C standard, which was finally adopted in 1989. The ANSI C standard is described in an appendix of the second edition of The C Programming Languages.

## The Compiler and the Linker

The source code is the actual text the programmer writes. The compiler is the program that translates the source code into object code, which the computer can read. The linker puts several compiled files into an executable file

Let us say we have a C program in the source code file Program.c and a routine used by the program in Routine.c. Furthermore, the program calls a function in the standard library. In this case, the compiler translates the source code into object code and the linker joins the code into the executable file Program.exe.

Compiler

Program.c

Linker

Program.o

Routine.c

Routine.o

Compiler

Standard Library

Program.exe

If the compiler reports an error we refer to it as compile-time error, and if an error occurs during the execution of the program we call it a run-time error.

## The Hello-World Program

In the first edition of The C Programming Language the hello-world example was introduced; that is, a small program that prints the text “Hello, World!”[[2]](#footnote-2) on the screen. To write a hello-world programoften means to write a small program to test that the compiler and linker work properly. The execution of the program does always start with the function main.

#include <stdio.h>

void main(void) {

puts("Hello, World!");

}

When executing, the program above generates the following printout.



## Comments

In C, it is possible to insert comments to describe and clarify the meaning of the program. The comments are ignored by the compiler[[3]](#footnote-3). There are two types of comments: line comments and block comments. Line comments start with two slashes and ends at the end of the line.

printf("Hello, World!"); // Prints "Hello, World!".

Block comments begin with a slash and an asterisk and ends with an asterisk and a slash. A block comment may range over several lines.

/\* This is an example of a C program. It writes the text

"Hello, World!" at the screen. \*/

#include <stdio.h>

void main(void) {

printf("Hello, World!");

}

Block comments cannot be nested. The following example will result in a compile-time error.

/\* A block comment cannot be /\* nested inside \*/ another

block comment. \*/

A piece of advice is that you use the line comments for regular comments, and save the block comments for situations when you need to comment a whole block of code for debugging purposes. Actually, there is a third level of comments if you use conditional programming, see macros (Section 13.31) for details.

#if 0

This is an example of a C program. It writes the text

"Hello, World!" at the screen.

#endif

#include <stdio.h>

void main(void) {

printf("Hello, World!");

}

## Types and Variables

There are several types in C. They can be divided into two groups: simple and compunded. The simple types can be further classified into integral, floating, and logical types. The compunded types are arrays, structures, unions, and pointers. They all (directly or indirectly) composed by simple types. We can also define a type with our own integer values, called the enumeration type.

### Simple Types

There are four simple types intended for storing integers: char, short int, int, and long int. They are called the integral types. The types short int and long int may be abbreviated to short and long, respectively. As the names implies, they are designed for storing characters, small integers, normal integers, and large integers, respectively. The exact limits of the values possible to store varies between different compilers. Such property that is not defined in the standard, but is rather defined by the compiler, linker, or operating system, is said to implementation dependent.

Furthermore, the integral types may be signed or unsigned. An unsigned type must not have negative values. If the word signed or unsigned is left out, a short int, int, and long int will be signed. Weather a char will be signed or unsigned is implementation dependent. However, a char is always one byte long, which means that it always holds a single character, regardless whether it is unsigned or not.

The next category of simple types is the floating types, which are used to store decimal numbers. The types are float, double, and long double; where float stores the smallest value and long double the largest one. How large values each type can store is implementation dependent. A floating type cannot be unsigned.

In many languages, there is a logical type (named bool, boolean, or logical) that stores logical value true or false. In C, however, there is no such type[[4]](#footnote-4). Instead, int is used as a logical type, where zero represents false and all other values represent true.

### Variables

A variable can be regarded as a box in the memory. In almost every case, we do not need to know its exact memory address. A variable always has a name, a type, and a value. We define a variable by simple writing its type and name. If we want to, we can initialize the variable; that is, assign it a value. If we do not, the variable’s value is undefined[[5]](#footnote-5). In this chapter, that value is denoted by a question mark (?).

int i = 123, j;

double d = 3.14;

char c = 'a';

123

i

?

j

3.14

d

'a'

c

We can transform values between the types by stating the new type within parentheses. The use of stating the type is not strictly necessary. However, if omitted the compiler may give warnings[[6]](#footnote-6).

int i = 123;

double x = 1.23;

int j = (int) x;

double y = (double) i;

### Constants

As the name implies, a constant is a variable whose value cannot be altered since it has been initialized. Unlike variables, constants must always be initialized and are often written in capital letters.

const double PI = 3.14;

### Output

As noted in the first section of this chapter, the standard function for output is printf (formatted print) writes to the standard output device (normally a text window).

The first argument is the format strings. With it, we can write every kind of value. However, there are many format codes to keep track of. A format code begins with the per cent character (%) followed with optional digits and signs ending with a character. We can write all the types we have went through so far. The character '\n' represents a new line. Below follows a list of the most common format codes.

|  |  |
| --- | --- |
| Code | Type |
| d | decimal integer |
| o | octal integer |
| x | hexadecimal integer |
| f | double |
| lf | long double |
| c | char |
| s | char \* |
| p | Prints the address of the variable in hexadecimal form. |

In the program below, *i* represents an integer, *f[[7]](#footnote-7)* a double, and *c* a character. Note that it is the responsible of the programmer to match the values following the format string with values of the correct types. There is no built-in error handling and mistakes would likely result in a run-time error.

#include <stdio.h>

void main() {

char c = 'a';

int i = 123;

double d = 1.23;

printf("char %c, int %d, double %f\n", c, i, d);

}

There is also the puts function that prints a string followed by a new-line. The following two lines are equivalent.

printf("Hello, World!\n");

puts("Hello, world!");

### Input

The standard functions for input is and scanf (formatted scan), it reads from the standard input device (normally the keyboard).

Similar to printf, its first argument is the format string. The format codes mostly agree with those of printf, but not completely. See the table at the end of the chapter for a list of format codes. For instance, scanf expects lf to read a double value, in contrast to printf that expects f*[[8]](#footnote-8)*.

Another difference is that while printf wants the values corresponding to the format codes, scanf wants the addresses of the variables that are going to store the values. This is accomplished by using the ampersand (&), see the Section 10.1.1 on pointers. Below follows a list of the most common format codes.

|  |  |
| --- | --- |
| Code | Type |
| d | decimal integer |
| o | octal integer |
| x | hexadecimal integer |
| f | float |
| lf | double |
| ld | long double |
| c | Char |
| S | char \* |

In the program below, note that scanf expects lf to represent a double float, in contrast to printf that expects f.

#include <stdio.h>

void main() {

char c;

int i;

double d;

printf("char: ");

scanf("%c", &c);

printf("int: ");

scanf("%d", &i);

printf("double: ");

scanf("%lf", &d);

printf("char %c, int %d, double %f\n", c, i, d);

}

There is also the gets function that reads a string until it reaches a new-line, which differs from scanf that reads a string until it reaches a white-space character[[9]](#footnote-9). It the user prints “Hello World” followed by a new-line twice when executing the code below, gets will place “Hello world” is s, while scanf will place “Hello” in t. Note that we do not give the addresses of s or t, since they are pointer to characters in themselves.

char s[20], t[20];

gets(s);

scanf("%s", t);

### Enumerations

An enumeration is a way to create our own integral type. We can define which values a variable of the type can store. In practice, however, enumerations are essentially an easy way to define integer constants[[10]](#footnote-10).

enum Cars {FORD, VOLVO, TOYOTA, VOLKSWAGEN};

Unless we state otherwise, the constants are assigned to zero, one, two, three, and so on. In the example above, FORD is an integer constant with the value zero; VOLVO has the value one, TOYOTA three, and VOLKSWAGEN four.

We do not have to name the enumeration type. In the example above, Cars can be omitted. We can also assign an integer value to some (or all) of the constants. In the example below, TOYOTA is assigned the value 10. The constants without assigned values will be given the value of the preceding constant before, adding one. This implies that VOLKSWAGEN will be assigned the value 11.

enum {FORD, VOLVO, TOYOTA = 10, VOLKSWAGEN};

### Arrays

An array is a variable compiled by several values of the same type. The values are stored on consecutive locations in memory. The array size may be stated explicitly of implicitly by a list of values. In the following example, a is given the size three, b is given size two and c is given size four, even though only its first two values are defined, which may cause the compiler to emit a warning. Naturally, d is given the size three.

int a[3] = {11, 12, 13};

double b[] = {1.2, 3.4};

char c[4] = {'a', 'b'}, d[3];

d

11

a

b

c

12

13

1.2

3.4

'a'

'b'

?

?

?

?

?

0

1

2

0

1

0

1

2

3

0

1

2

A value of an array can be accessed by index (brackets) notation.

int i = a[2];

double x = b[0];

char t = c[1];

13

i

1.2

x

'b'

t

### Characters and Strings

As stated in Section 1.1.1, a char is a small integer type intended to store exactly one character. A string stores a (possible empty) sequence of characters. There is no built in type for describing a string. Instead, a string is defined by an array of characters. Note that characters are enclosed by single quotations while strings are enclosed by double quotations.

char c = 'a';

char s[] = "Hello, World!";

'a'

c

"Hello, World!"

s

When defining a string, the compiler adds a special zero character '\0' at the end. Its task is to mark the end of the string. If the code above, the character array s has a size of fourteen (thirteen regular characters, including space, comma, and exlamation mark as well as the finishing zero character). The two following definitions are equivalent, they both define character arrays of size four (three regular characters and the finishing zero character).

char s1[] = "abc";

char s2[] = {'a', 'b', 'c', '\0'};

### The ASCII Table

So far we have assigned characters to char variables. Technically, the compiler translates a character literal into an integer in accordance to the ASCII table (see Appendix Fel! Hittar inte referenskälla.), where character has a corresponding integer value. For instance, ‘a’ corresponds to 97, ‘b’ to 98, and so on. The line above could have been written as the following code instead. The possibility to write character and string literal is for convenience only.

char s2[] = {97, 98, 99, 0};

### Pointers

A pointer is a variable containing the address of value. For example, let us say that the integer i has the value 123 that is stored on the memory address 10,000, and that the size of each value is four bytes. As p is a pointer to i, it holds the value 10,000.

10000

p

123

10000

9992

9996

10004

In almost all cases, we do not really need to know the address of the value. Therefore, a simpler (and clearer) way to illustrate the same thing is to draw an arrow from the pointer to the value.

p

123

i

The following code gives rise to the diagram above[[11]](#footnote-11), where the ampersand (&) denotes the address of the variable.

int i = 123;

int \*p = &i;

If we want to access the value pointed at, we use the asterisk (\*), which dereferenceers the pointer (“follows the arrow”). As the address operator (&) gives the address of a variable (“against the arrow”), they address and dereferenceerring operators can be regarded as each other reverses. Note that the asterisk is used on two occasions, when we define a pointer variable and when we dereferenceer a pointer. The asterisk is in fact used on a third occasion, when multiplying two values. The ampersand is used on two occasions: address and bitwise and[[12]](#footnote-12).

int i = 999;

int \*p = &i; // Pointer definition.

int j = \*p; // Dereferenceereeing of a pointer.

int a = 1, b = 2;

int c = a \* b; // Multiplication.

### Pointers and Dynamic Memory

Pointers can also be used to allocate dynamic memory. There is a section of the memory called the heap that is used for dynamically allocated memory blocks. The standard functions malloc, calloc, realloc and free used to allocate, reallocate, and deallocate the memory, respectively. Memory not dynamically allocated is referred to static memory.

p

123

int \*p = malloc(sizeof(int));

\*p = 123;

free(p);

We can also allocate memory for a whole array. Even though p is a pointer in the example below, we can use the array index notation to access a value of the array in the allocated memory block.

p

123

124

125

0

1

2

int \*p = calloc(3, sizeof(int));

p[0] = 123;

p[1] = 124;

p[2] = 125;

free(p);

Once we have allocated a memory, we can allocate a block of different size by calling realloc, which copies to contents of the memory block to new address, is necessary.

int \*p = malloc(sizeof(int));

p[0] = 123;

p = realloc(2 \* sizeof(int));

p[1] = 456;

free(p);

p

123

456

0

1

The difference between malloc and calloc is that calloc resets the bytes of the memory blocks to zero, which malloc do not. Otherwise, the following two calls are equivalent.

int \*p = calloc(3, sizeof(int));

int \*q = malloc(3 \* sizeof(int));

The predefined constant NULL holds the pointer equivalence of the zero value. We say that the pointer is set to null. In a diagram, we can simple write null.

p

NULL

int \*p = NULL;

Sometimes, the electric ground symbol is used to symbolize a null pointer. For this reason, a null pointer is sometimes said to be a grounded pointer.

p

There is a special type void, which actually marks the absence of a type. We can define a pointer to void; we cannot, however, dereferenceer the pointer. It is useful in low-level application where we want to exam a specific location in memory.

void\* voidPtr = (void\*) 10000;

The void type is useful on one further occasion: to mark that a function does not return a value or misses parameters, see the function section later in this chapter.

In the example below, the memory block has been deallocated, but p has not been set to null. It has become a dangling pointer; it is not null and does not really points at anything. In spite of that, we try to access the value p points at. This is a dangerous operation and would most likely result in a run-time error.

p

dangling pointer

int \*p = malloc(sizeof(int));

\*p = 1;

free(p);

\*p = 2; // Undefined behavor

In the example below, we allocate memory for two pointers, p and q. Then we assign p to q, by doing so we have created a memory leak. There is no way we can access or deallocate the memory block that was pointed at by p. In fact, we deallocate the same memory block twice as both pointers by now points at the same memory block. This dangerous operation will most likely also result in a run-time error.

p

(a)

q

1

2

p

(b)

q

1

2

int \*p = malloc(sizoef(int)); // (a)

int \*q = malloc(sizeof(int));

\*p = 1;

\*q = 2;

p = q; // (b)

free(p); // Undefined behavor. Deallocates the same memory block twice,

free(q); // as p and q points at the same memory block. Also memory leak,

// since the memory block holding 1 cannot be accessed any more.

### Defining our own types

It is possible to define our own type with typedef, which is a great tool for increasing the readability of the code. However, too many defined types tend to make the code less readable. Therefore, a piece of advice would be to use typedef with care.

int i = 1;

typedef unsigned int unsigned\_int;

unsigned\_int u = 2;

typedef int\* int\_ptr;

int\_ptr ip = &i;

typedef unsigned\_int\* uint\_ptr;

uint\_ptr up = &u;

### The Type Size

The operator sizeof gives us the size of a type[[13]](#footnote-13), either by taking the type surrounded by parentheses or a value of the type. The size of a character is always one byte and the signed and unsigned forms of each integral type always have the same size. Otherwise, the sizes are decided by the compiler. In these cases, when a property is not specified by the standard by rather the compiler, we say the property is implementation dependant. The operator returns a value of the type size\_t, which exact definition is also implementation dependant. However, it is often an unsigned integer.

#include <stdio.h>

void main() {

int intSize1 = sizeof (int);

int intSize2 = sizeof intSize1;

int\* intPtr = &intSize1;

int ptrSize = sizeof intPtr;

int array[3] = {1, 2, 3};

int arraySize = sizeof array;

printf("Integer sizes: %d and %d.\n", intSize1, intSize2);

printf("Pointer size: %d\n", ptrSize);

printf("Array size: %d\n\n", arraySize);

}

## Expressions and Operators

The operations of C are divided into the arithmetic, relational, logical, and bitwise operators as well as simple and compound assignment. Moreover, there is the conditional operator.

In the figure below, + is an operator, 1 and 2 are operands and the whole term is an expression.



Operator

Operand

Operand

### Arithmetic Operators

The arithmetic operators are addition (+), subtraction (-), multiplication (\*), division (/), and modulo (%). The first four operators are equivalent to the four fundamental rules of arithmetic. The operators can take operands of integral and floating types. The last operator, modulo, gives the remainder of integer division. If we mix integral and floating types in the expression, the result will have floating type. The modulo operator, however, only works with integral operands. The last assignment in the following code may give rise to a compiler warning as the result of the division is a double and is converted into an integer.

int a = 10, b = 3, c;

c = a + b; // 13

c = a - b; // 7

c = a \* b; // 30

c = a / b; // 3, integer division

c = a % 3; // 1, remainder

double d = 3.0, e;

e = a / d; // 3.333, floating type

### Pointer Arithmetic

The addition and subtraction operators are also applicable on pointers. It is called pointer arithmetic. An integral value can be added to or subtracted from a pointer. The value of the pointer is then changed by the integral value times the size of the type the pointer points at. As the void type is not really a type, but rather the absence of a type, it has no size. Therefore, we cannot perform pointer arithmetic on void pointers.

In the code below, let us assume that number is stored at memory location 10,000 and that the integer type has the size of four bytes. Then the pointer numberPtr will assume the values 10,000, 10,004, 10,008, and 10,012, not the values 10,000, 10,001, 10,002, and 10,003, as pointer arithmetic always takes the size of the type into consideration.

numberPtr

102

100

101

103

10008

10000

10004

10012

int number = 100;

int\* numberPtr = &number;

numberPtr = numberPtr + 1;

\*numberPtr = number + 1;

numberPtr = numberPtr + 1;

\*numberPtr = number + 2;

numberPtr = numberPtr + 1;

\*numberPtr = number + 3;

There is also possible to subtract two pointers pointing at the same type. The result will be the difference in bytes between their two memory locations divided with the size of the type. Again, pointer arithmetic always considers the type size.

int array[] = {1, 2, 3};

int\* p1 = &array[0];

int\* p2 = &array[2];

int diff = p2 - p1; // 2

The index notation for arrays is equivalent to the dereferenceereeing of pointers together with pointer arithmetic. The second and third lines of the following code are by definition interchangeable.

int array[] = {1, 2, 3};

array[1] = array[2] + 1;

\*(array + 1) = \*(array + 2) + 1;

### Increment and decrement

There are two rather special operators: increment (++) and decrement (--)[[14]](#footnote-14). They add or subtract one from its operand. The operator can be placed before (prefix) or after (postfix) its operand.

int a = 1, b = 1;

++a; // 2, prefix increment

b++; // 2, postfix increment

However, there is a difference between prefix and postfix increment or decrement. In the prefix case below, the subtraction occurs first and the new value is returned; in the postfix case, after the subtraction the original value is returned.

int a = 1, b = 1, c, d;

c = --a; // c = 0, prefix decrement

d = b--; // d = 1, postfix decrement

### Relational Operators

There are six relational operators: equal to (==), not equal to (!=), less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=)[[15]](#footnote-15). Observe that the equal to operator is constituted by two equal signs rather than one. The operators give a logical value, true (an integer value not equal zero) or false (zero). The operands shall be of integral of floating type.

int i = 3;

double x = 1.2;

int b = i > 0; // true (an integer value not equal to zero, usually one)

int c = (x == 2); // false (always zero)

### Logical Operators

There are three logical operators: not (!), or (||), and and (&&). They take and return numerical values interpreted as logical values.

int i = 3;

int b, c, d, e;

b = (i == 3); // true (not zero)

c = !b; // false (zero)

d = b || c; // true (not zero)

e = b && c; // false (zero)

C applies lazy evaluation (also called short-circuit evaluation), which means that an expression will not be evaluated to a further extent than necessary to find its value. In the following example, the evaluation of the expression is completed when the left expression (i != 0) is evaluated to false. If the left expression is false, the whole expression must also be false because it needs both the left and right sub expressions to be true for the whole expression to be true. This gives that the right expression (1 / i == 1) will never be evaluated and the division with zero will never occur.

int i = 0;

int b = (i != 0) && (1 / i == 1); // false (zero);

### Bitwise Operators

An integer value can be viewed as a bit pattern. Our familiar decimal system has base 10; it can be marked with an index 10.



An integer value can also be viewed with the binary system, which has base 2. A single digit is called a bit, and the integer value is called a bit pattern. A bit may have the values one and zero.



There are four bitwise operations in C: inverse (~), and (&), or (|), and exclusive or (^). Exclusive or means that the result is one if one of its operand bits, but not both, is one. They all operate on integral values on bit level; that is, they examine each individual bit of an integer value.

101010102 101010102 101010102

& 100101102 | 100101102 ^ 100101102 ~ 100101102

----------- ------------ ------------ ------------

= 100000102 = 101111102 = 001111002 = 011010012

int a = 170; // 101010102

int b = 150; // 100101102

int c = a & b; // 100000102 = 13010

int d = a | b; // 101111102 = 19010

int e = a ^ b; // 001111002 = 6010

int f = ~b; // 011010012 = 10510

An integer value can also be shifted to the left (<<) or to the right (>>). Each left shift is equivalent with doubling the value, and each right shift is equivalent with (integer) dividing the value with 2. Overflowing bits are dropped for unsigned values; the behaviour of signed values is implementation dependent. Therefore, I recommend you only perform bitwise operations on unsigned values.

unsigned char a = 172; // 101011002

unsigned char b = a << 2; // 101100002 = 17610

unsigned char c = 166; // 101001102

unsigned char d = c >> 2; // 001010012 = 4110

### Assignment

There are two kinds of assignment operators: simple and compound. The simple variant is quite trivial, one or several variables are assigned the value of an expression. In the example below, a, b, and c are all assigned the value 123.

int a, b, c, d = 123;

a = d;

b = c = d;

The compound variant is more complicated. Let us start with the additional assignment operator. In the example below, the value of a is increased by the value of b; that is, a is given the value 4.

int a = 2, b = 4, c = 2;

a += c; // 4, equivalent to a = a + c.

b -= c; // 2, equivalent to a = a - c.

In a similar manner, there are operations -=, \*=, /=, %=, |=, &=, ^=, <<=, and >>=. Note the difference between a -= 2 and a =- 2. In the former case, a is decreased by 2. In the latter case, a is assigned -2.

### The Condition Operator

The condition operator resembles the if-else statement of the next section. It is the only C operator taking three operands. The first expression is evaluated. If it is true, the second expression is evaluated and its value is returned. If the first expression instead is false, the third expression is evaluated and its value is returned.

int a = 1, b = 2, max;

max = (a > b) ? a : b; // The maximal value of a and b.

Too frequent use of this operator tends to make the code compact and hard to read. A piece of advice is that you restrict your use of the operator to the trivial cases.

### Precedence and Associatively

Given the expression 1 + 2 \* 5, what is its value? It is 11 because we first multiply two with five and then add one. We say that multiplication has a higher precedence than addition.

What if we limit ourselves to one operator? Let us pick subtraction. What value has the expression 8 – 4 – 2? As we first subtract 4 from 8 and then subtract 2, the result is 2. As we evaluate the value from left to right, we say that subtraction is left associative.

Below follows a table over the prorities and associative of the C operators. The first operator in the table has the highest precedence.

|  |  |  |
| --- | --- | --- |
| Group | Operators | Associatively |
| Brackets and fields | () [] -> . | Left to Right |
| Unary operator | ! ~ ++ -- + - (type) sizeof | Right to Left |
| Arithmetic operators | \* / %  + - | Left to Right  Left to Right |
| Shift operators | << >> | Left to Right |
| Relation operators | < <= > >=  == != | Left to Right |
| Bitwise operators | &  ^  | | Left to Right |
| Logical operators | &&  || | Left to Right |
| Conditional operator | ?: | Right to Left |
| Assignment operators | = += -= \*/ /= %= &= ^= |= <<= >>= | Right to Left |
| Comma operator | , | Left to Right |

Note that unary +, -, &, and \* have higher precedence than their binary forms. Note that we always can change the evaluation order of an expression by inserting parantheses at appropriate posititions. The expression (1 + 2) \* 5 has the value 15.

## Statements

There are four kinds of statements in C: the selection, iteration, jump, and expression statements.

|  |  |
| --- | --- |
| Kind | Statements |
| Selection | if, if-else, switch |
| Iteration | for, while, do-while |
| Jump | break, continue, goto, return |
| Expression | expression ; |

### Selection statements

The if statement needs in its simplest form a logical expression to decide whether to execute the following statement. The example below means that the text will be output if i not equal zero.

if (i != 0) {

puts("i does not equal zero");

}

As all values not equal to zero are interpreted as true, the following statement is equivalent with the previous one.fexacltly

if (i) {

puts("i does not equal zero");

}

We can also attach an else part, which is executed if the expression is false.

if (i > 0) {

puts("i is greater than zero");

}

else { puts("i is not greater than zero");

}

Between the if and else part we can insert one or more else if parts.

if (i > 0) {

puts("i is greater than zero");

}

else if (i < 0) {

puts("i is less than zero");

}

else {

puts("i is equal to zero");

}

In the examples above, there is not strictly necessary to surround the output statements with brackets. However, in this book brackets are always used for sake of clarity. Moreover, it is always necessary in case of several statements. The brackets and the code in between are called a block.

if (i > 0) { int j = i + 1;

printf("j is %d\n", j);

}

A warning may be in order. In an if statement, it is perfectly legal to use one equal sign instead of two signs when comparing two values. As one equals sign is used for assignment, not comparison, the variable i in the following code will be assign the value one, and the expression will always be true.

if (i = 1) { // Always true.

// ...

}

One way to avoid the mistake above is to swap the variable and the value. As a value can be compared but not assigned, the compiler will issue an error message and the error will be recognized.

if (1 = i) { // Compile-time error.

// ...

}

The switch statement is simpler than the if statement, but not as powerful. It evaluates the switch expression and jumps to a case statement with the matching (constant) value. If no value matches, it jumps to the default statement, if present. The break statement is used to jump out of a switch or iteration statements. The switch expression must have an integral or pointer type and two case statements cannot hold the same value. We can only have one default statement, and it can be omitted. However, it needs not to be placed at the end of the switch statement, even though I recommend you to do so.

switch (i) { case 1:

puts("i is equal to 1");

break;

case 2:

puts("i is equal to 2");

break;

case 3:

puts("i is equal to 3");

break;

default:

puts("i is not equal to 1, 2, or 3.");

break;

}

It is important to remember the break statement. Otherwise, the execution would simple continue with the code attached to the next case statement[[16]](#footnote-16). However, in C we can use the fact that an omitted break statement makes the execution continue with the next statement to group several case statements together.

switch (i) { case 1:

case 2:

case 3:

printf("i is equal to 1, 2, or 3");

break;

// ...

}

### Iteration statements

Iteration statements, also called loops, iterate one or several statements as long as a certain condition is true; while is the simplest loop. It repeats one statement or a block of statements as long as the given expression is true. The example below uses the while statement to write the numbers one to ten.

int i = 1;

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

++i;

}

The same thing can be done with a do-while statement.

int i = 1;

do { printf("%d\n", i);

++i;

}

while (i <= 10);

However, the do-while statement is less powerful. If the expression is false to begin with, the while statement just skips the repetitions altogether, but the do-while statement must always execute the iteration statement at least once in order to reach the continuation expression at the end.

We can also use the for statement, which is a more compact variant of while. It takes three expressions, separated by semicolons. The first expression initializes the variable, the repetition continues as long as the second expression is true and the third expression is executed at the end of each repetition.

int i;

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

}

Similar to the switch statement, an iteration statement can be interrupted by the break statement. Below is an infiniteloop (remember that every non-zero value is regarded as true) that is stopped by the break statement.

int i = 1;

while (1) { printf("%d\n", i);

++i;

if (i > 10) { break;

}

}

The same effect can be archived by omitting the second expression of the for statement.

int i;

for (i = 1; ; ++i) { printf("%d\n", i);

if (i == 10) { break;

}

}

An iteration statement can also include a continue statement. It skips the rest of the current repetition. The following example prints the numbers from 1 to 10 with the exception of 5.

int i;

for (i = 1; i <= 10; ++i) { if (i == 5) { continue;

}

printf("%d\n", i);

}

The following example, however, will not work as intended. Because the continue statement will skip the rest of the while block, i will never be updated and we will be stuck in an infiniteloop. Therefore, I recommend that you use continue with care, or avoid it altogether.

int i = 1;

while (i <= 10) { if (i == 5) { continue;

}

printf("%d\n", i);

++i;

}

### Jump statements

We can jump from one location to another inside the same function by marking the latter location with a label inside the block with the goto statement. The following code is yet another example of how to print the numbers from one to ten, inclusive. As you can, it is not as clear as the previous examples.

int i = 1;

label: printf("%d\n", i);

++i;

if (i <= 10) { goto label;

}

The goto statement is considered to give rise to unstructured code, so called “spagetti code.” I strongly recommended that you omit goto altogether. To put it bluntly: now when you seen it, forget it.

### Expression statements

An expression can form a statement.

a = b + 1; // Assignment operator.

puts("Hello, World!"); // Function call.

In the above examples, we are only interested in the side effects; that a is assigned a new value and that a text is written. We are allowed to write expression statements without side effects; even thought it has no meaning and is likely to be erased by the compiler.

a + b \* c;

### Volatile

A variable can be volatile, which has no functional effect but is rather a notification to the compiler that the variable shall not be subjected to optimization. Let us say we that the memory location 10000 is connected to an input port in a low-level application, and that we want to wait until we receive a signal through the port. If p is not marked as volatile, there is a risk that the compiler would replace the for-loop with another statement since the value of \*p does not seem to change inside the loop.

volatile int\* p = (void\*) 10000;

while (\*p != 0) {

// Wait.

}

## Functions

A function can be compared to a black box. We send in information (input) and we receive information (output). In C, the input values are called parameters and the output value is called the return value.

Input (Parameters)

Function

Output (Return Value)

To start with, let us try the function Square; it takes an integer and returns its square.

int Square(int n) {

return n \* n;

}

void main() {

int i = Square(2); // Square returns 4.

}

2

Square

4

In the example above, the parameter n in the Square definition is called a formal parameter, and the value 3 in the Square call in main is called an actual parameter or an argument.

Now, let us try a more complicated function: SquareRoot takes a value of double type and returns its square root. The idea is that the function iterates and calculates increasingly better root values by taking the mean value of the original value divided with the current root value and the previous root value. The process continues until the difference between two consecutive root values has reached an acceptable tolerance. Just like main[[17]](#footnote-17), a function can have local variables; root and oldRoot hold the current and previous value of the root, respectively.

#include <stdio.h>

double SquareRoot(double value) {

const double EPSILON = 1e-12;

double root = value, oldRoot = value;

while (1) { root = ((value / root) + root) / 2;

if ((oldRoot - root) <= EPSILON) { return root;

}

oldRoot = root;

}

}

void main() {

double input = 16;

printf("SquareRoot of %f: %f\n", input,

SquareRoot(input));

}

### Void Functions

A function does not have to return a value; in that case, we set void as the return type. As mentioned above, void is used to state the absence of a type rather than a type in itself. We can return from a void function by just stating return without a value.

void PrintSign(int value) {

if (value < 0) { puts("Negative.");

return;

}

if (value > 0) { puts("Positive.");

return;

}

puts("Zero");

}

There is not a problem if the execution of a void function reaches the end of the code, it just jump back to the calling function. However, a function not returning void shall always return a value before reaching the end of the function. Otherwise, the compiler often gives a warning.

### Local and global variables

There are two kinds of variables: local and global. A global variable is defined outside a function and a local variable is defined inside a function.

#include <stdio.h>

int global = 1;

void main() {

int local = 2;

printf("Global variable: %d, Local variable: %d.\n",

global, local);

}

A global and a local variable can have the same name. In that case, the name inside the function refers to the local variable. We cannot access the global variable in the inner scope[[18]](#footnote-18) when a local variable with same name is present.

#include <stdio.h>

int number = 1;

void main() {

int number = 2;

printf("Variable: %d.\n", number); // 2

}

One way to distinguish them is to precede the global variable name with ’g\_’.

#include <stdio.h>

int g\_number = 1;

void main() {

int number = 2;

printf("Global variable: %d, Local variable: %d.\n",

g\_number, number); // 1, 2

}

Global variables can only be initialized with constant values.

int g\_number1 = 1; // Right.

int g\_number2 = g\_number1 + 1; // Wrong.

### Inner Blocks with Local Variables

In C, you are allowed to use the bracket notation when initializerializing structures, unions, and arrays, but not when assigning[[19]](#footnote-19). Therefore, the following code will not work.

INT\_PAIR Block(int\* p) {

INT\_PAIR result;

if (p != NULL) {

result = {\*p - 1, \*p + 1}; // Wrong.

return result;

}

else {

result = {0, 0}; // Wrong.

return result;

}

}

One solution would be to a more clumsy code.

INT\_PAIR Block(int\* p) {

INT\_PAIR result;

if (p != NULL) {

result.a = \*p - 1;

result.b = \*p + 1;

return result;

}

else {

result.a = 0;

result.b = 0;

return result;

}

}

However, a more elegant solution would be to use inner blocks with local variables. Each block can define its own variable, which is especially valuable when initializerializing structures.

struct IntPair {

int a, b;

};

struct IntPair Block(int\* p) {

if (p != NULL) {

struct IntPair result = {\*p - 1, \*p + 1};

return result;

}

else {

struct IntPair empty = {0, 0};

return empty;

}

}

An alternative way is to use typedef.

typedef struct {

int a, b;

} INT\_PAIR;

INT\_PAIR Block(int\* p) {

if (p != NULL) {

INT\_PAIR result = {\*p - 1, \*p + 1};

return result;

}

else {

INT\_PAIR empty = {0, 0};

return empty;

}

}

### Call-by-Value and Call-by-Reference

Say we want to write a function for switching the values of two variables.

#include <stdio.h>

void Swap(int number1, int number2) {

int temp = number1; // (a)

number1 = number2; // (b)

number2 = temp; // (c)

}

void main() {

int num1 = 1, num2 = 2;

printf("Before: %d, %d\n", num1, num2); // 1, 2

Swap(num1, num2);

printf("After: %d, %d\n", num1, num2); // 1, 2

}

Unfortunately, it will not work; the variables keep their values. The explanation is that the values of firstNum and secondNum in main are copied into num1 and num2 in Swap. Then num1 and num2 exchange values with the help of temp. However, their values are not copied back into firstNum and secondNum in main.

1

2

main:

1

firstNum

2

Swap:

num2

1

temp

secondNum

num1

2

2

num2

1

temp

num1

2

1

num2

1

temp

num1

(a)

(b)

(c)

The problem can be solved with reference calls. Instead of sending the values of the actual parameters, we send theirs addresses by define the formal parameters as pointers to the type.

#include <stdio.h>

void Swap(int\* numberPtr1, int\* numberPtr2) {

int temp = \*numberPtr1; // (a)

\*numberPtr1 = \*numberPtr2; // (b)

\*numberPtr2 = temp; // (c)

}

void main() {

int num1 = 1, num2 = 2;

printf("Before: %d, %d\n", num1, num2); // 1, 2

Swap(&num1, &num2);

printf("After: %d, %d\n", num1, num2); // 2, 1

}

In this case, we do not send the values of firstNum and secondNum, but rather their addresses. Therefore, num1 and num2 in Swap does in fact contain the addresses of firstNum and secondNum on main. As in the reference section above, we illustrate this with dashed arrows. Therefore, when num1 and num2 exchanges values, in fact the values of firstNum and secondNum are exchanged.

1

2

firstNum

num2

1

temp

secondNum

num1

(a)

2

2

firstNum

num2

1

temp

num1

(b)

2

1

firstNum

num2

1

temp

num1

(c)

### Static, Extern, and Register Variables

In the function below, s\_count (the prefix ‘\_s’ is used for static variables) is a static local variable, which means that it is initialized when the execution of the program starts rather when the function is called. If s\_count was a regular local variable (without the keyword static), the function would, at every call, write that the function has been called once, as s\_count would be initialized to zero at every call.

void KeepCount() {

static int s\_count = 0;

s\_count++;

printf("This function has been called %d times.", s\_count);

}

If we define a global variable in one file, it will be accessable in another file if we declare[[20]](#footnote-20) it as extern. If we omit the extern keyword in the second file, the linker would complain about us defining two global variables with the same name.

File1.c

int g;

File2.c

extern int g;

Finally, a variable can also be marked with the keyword register, which is a notification to the compiler that the variable is suitable to place in a process register rather them the memory. The only function difference is that we cannot get the address of a register variable, since it might be located in a register.

register int i;

int\* p = &i; // Wrong

### Recursion

A function may call itself; it is called recursion. In the following example, the mathematical function factorial (n!) is implemented. It can be defined in two ways. The first definition is rather straightforward. The result of the function applied to a positive integer n is the product of all positive integers up to and including n.



int Factorial(int n) {

int result = 1, i;

for (i = 1; i <= n; ++ i) { result \*= i;

}

return result;

}

An equivalent definition involves a recursive call that is easier to implement.



int Factorial(int n) {

if (n == 1) {

return 1;

}

else { return n \* Factorial(n - 1);

}

}

### Definition and Declaration

It important to distinguish between definition and declaration. For a function, its definition generates code while the declaration is merely an item of information to the compiler. A function declaration is also called a prototype.

When it comes to mutual recursion (two functions calling each other), at least the second of them must have a prototype to avoid compiler warnings. I recommend that you put prototypes for all functions at the beginning of the file (or in a header file, see Section 13.29). In the following example, we use two functions to decide whether a given non-negative integer is even or odd according to the formulas below.





Note that it is not necessary to name the parameters in a function prototype. If names after all are given, they will be ignored by the compiler, which means that the parameters do not need to have the same names in the function declaration and definition. The only restriction is that two parameters cannot have the same name.

int IsEven(int);

int IsOdd(int n);

As C does not include a logical type, we use int to represent true (1) or false (0).

int IsEven(int num) {

if (num == 0) {

return 1;

}

else { return IsOdd(num - 1);

}

}

int IsOdd(int num) {

if (num == 0) {

return 0;

}

else {

return IsEven(num - 1);

}

}

One peculiar thing about prototypes in C is that they can have an unspecified parameter list. If the parameter list is completely omitted in a function prototype, every parameter list is allowed in the call. The following code is will not result in any compile-time errors. However, there is likely to be run-time errors.

void Print();

void main(void) {

Print();

Print(1);

Print(1, 2);

Print(1, 2, 3);

}

The obvious way to avoid the run-time errors is to always state the parameter list in function prototypes. If the function does not have any parameters, it can be stated by the void type. In that case, the function can only be called without parameters.

void Print(void);

### Higher-Order Functions

A function that takes another function as a parameter is called a higher-order function. Technically, it does not take the function itself as a parameter, but rather a pointer to the function. However, the pointer marker (\*) may be omitted. The parameter function is also called a callback function. The following example takes an array of the given size and applies the given function to each integer in the array. ApplyArray a higher order function and Apply is a callback function.

#include <stdio.h>

void ApplyArray(int intArray[], int size, int Apply(int)) {

int index;

for (index = 0; index < size; ++index) { intArray[index] = Apply(intArray[index]);

}

}

int Double(int number) {

return 2 \* number;

}

int Square(int number) {

return number \* number;

}

void PrintArray(int intArray[], int size) {

int index;

for (index = 0; index < size; ++index) { printf("%d ", intArray[index]);

}

printf("\n");

}

void main() {

int numberArray[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int arraySize = sizeof numberArray / sizeof numberArray[0];

PrintArray(numberArray, arraySize);

// Doubles every value in the array.

ApplyArray(numberArray, arraySize, Double);

PrintArray(numberArray, arraySize);

// Squares every value in the array.

ApplyArray(numberArray, arraySize, Square);

PrintArray(numberArray, arraySize);

}

An alternative way is to define the type of the callback function. In the following code segment, the APPLY\_FUNC type is a pointer to a function that takes an integer parameter and returns an integer value.

typedef int (\*APPLY\_FUNC)(int);

void ApplyArray(int intArray[], int size,

APPLY\_FUNC pApplyFunc) {

int index;

for (index = 0; index < size; ++index) { intArray[index] = pApplyFunc(intArray[index]);

}

}

One additional point of the example above is how to of find the size of an array: we divide the size of the array with the size of its first value. This function does only work on static arrays, not on dynamically allocated arrays or array given as parameters to functions. A parameter array is in fact converted to a pointer to the array type. The following two function definitions are (by definition) equivalent. This gives that the function must have an extra integer parameter stating the array size.

void PrintArray(int intArray[], int arraySize) {

// ...

}

void PrintArray(int\* intArray, int arraySize) {

// ...

}

## Structures and Linked Lists

A struct is a compound type. Like arrays, it holds several values, called fields. However, in contrast to arrays, the values can hold different types.

A pointer may point at a struct value as well as a simple value, and a struct may have a pointer to another struct value as a field, which in turn points at another struct value, and so one. In this way, a linked list can be constructed. The list must end eventually, so the last pointer points at null. A pointer to the next cell in the list is called a link.

struct Cell {

int value;

struct Cell\* nextPtr;

};

struct Cell cell3 = {3, NULL};

struct Cell cell2 = {2, &cell3};

struct Cell cell1 = {1, &cell2};

1

value

nextPtr

2

value

nextPtr

3

value

nextPtr

NULL

### Stacks and Linked Lists

A stack is very valuable in a number of applications and it can be implemented with a linked list. We can add a value on top of the stack, we can inspect or remove the topmost value, and we can check whether the stack is empty. However, we cannot do anything to the values that are not on top. The function adding a new value on top of the stack is called push and the function removing it is called pop. Let us say that we push our stack three times with the values 1, 2, and 3. Then we can only access the topmost value, 3, and not the two below, 1 or 2.

1

2

3

push 1

push 2

push 3

Cell.h

struct Cell { int m\_value;

struct Cell\* m\_nextPtr;

};

typedef struct Cell CELL;

void CellInitializer(CELL\* cellPtr, int value, CELL\* nextPtr);

void CellSetValue(CELL\* cellPtr, int value);

int CellGetValue(CELL\* cellPtr);

void CellSetNext(CELL\* cellPtr, CELL\* nextPtr);

CELL\* CellGetNext(CELL\* cellPtr);

Cell.c

#include "cell.h"

void CellInitializer(CELL\* cellPtr, int value, CELL\* nextPtr) {

cellPtr->m\_value = value;

cellPtr->m\_nextPtr = nextPtr;

}

void CellSetValue(CELL\* cellPtr, int value) {

cellPtr->m\_value = value;

}

int CellGetValue(CELL\* cellPtr) {

return cellPtr->m\_value;

}

void CellSetNext(CELL\* cellPtr, CELL\* nextPtr) {

cellPtr->m\_nextPtr = nextPtr;

}

CELL\* CellGetNext(CELL\* cellPtr) {

return cellPtr->m\_nextPtr;

}

Main.c

#include <stdlib.h>

#include "cell.h"

void main() {

CELL\* cellPtr1 = malloc(sizeof(CELL));

CELL\* cellPtr2 = malloc(sizeof(CELL));

CELL\* cellPtr3 = malloc(sizeof(CELL));

CellInitializer(cellPtr3, 3, NULL);

CellInitializer(cellPtr2, 2, cellPtr3);

CellInitializer(cellPtr1, 1, cellPtr2);

}

The following struct is created by main above.

1

m\_value

m\_nextPtr

2

m\_value

m\_nextPtr

3

m\_value

m\_nextPtr

NULL

cellPtr1

cellPtr2

cellPtr3

However, there is one more thing to think about. What happens if we run out of dynamic memory or try to access the topmost value of an empty stack? We can deal with the problem in some different ways, everything from ignoring it to abort the execution. In this book, I have limited the error handling of memory shortage to the use of assert, which is a macro (see Section 13.31) that takes a logical value and aborts the execution if the value is false, adding an error message with information about the file name and the code line number. To keep things simple, let us use that function in the stack structure too.

Stack.h

struct Stack { CELL\* m\_firstCellPtr;

};

typedef struct Stack STACK;

void StackInitializer(STACK\* stackPtr);

void StackClear(STACK\* stackPtr);

void StackPush(STACK\* stackPtr, int value);

void StackPop(STACK\* stackPtr);

int StackTop(STACK\* stackPtr);

int StackIsEmpty(STACK\* stackPtr);

Stack.c

#include <stdlib.h>

#include <assert.h>

#include "cell.h"

#include "stack.h"

void StackInitializer(STACK\* stackPtr) {

stackPtr->m\_firstCellPtr = NULL;

}

void StackClear(STACK\* stackPtr) {

CELL\* cellPtr = stackPtr->m\_firstCellPtr;

while (cellPtr != NULL) { CELL\* tempCellPtr = cellPtr;

cellPtr = CellGetNext(cellPtr);

free(tempCellPtr);

}

}

void StackPush(STACK\* stackPtr, int value) {

CELL\* newCellPtr= malloc(sizeof(CELL));

assert(newCellPtr!= NULL);

CellInitializer(newCellPtr, value, stackPtr->m\_firstCellPtr);

stackPtr->m\_firstCellPtr = newCellPtr;

}

void StackPop(STACK\* stackPtr) {

CELL\* tempCellPtr;

assert(stackPtr->m\_firstCellPtr != NULL);

tempCellPtr = stackPtr->m\_firstCellPtr;

stackPtr->m\_firstCellPtr = CellGetNext(stackPtr->m\_firstCellPtr);

free(tempCellPtr);

}

int StackTop(STACK\* stackPtr) {

assert(stackPtr->m\_firstCellPtr != NULL);

return CellGetValue(stackPtr->m\_firstCellPtr);

}

int StackIsEmpty(STACK\* stackPtr) {

return (stackPtr->m\_firstCellPtr == NULL);

}

Main.c

#include <stdlib.h>

#include "cell.h"

#include "stack.h"

void main() {

STACK stack;

StackInitializer(&stack);

StackPush(&stack, 1);

StackPush(&stack, 2);

StackPush(&stack, 3);

StackClear(&stack);

}

The stack in main above gives rise to the following struct. However, when StackClear is called, it deallocates the allocated memory. All memory allocated with malloc or calloc must (or at least should) be deallocated with free.

m\_nextPtr

3

2

m\_value

m\_nextPtr

1

m\_value

m\_nextPtr

NULL

m\_value

m\_firstCellPtr

### Unions

A union is a compound type like struct. The difference is that while a struct places each of its members on different memory addresses in order for them to not intervene with each other, a union locates each of its members on the same address. This implies that when one member is assigned a value, it overwrites the other values. Unions are mostly used in low-level applications. In the code below, the two structures overlap each other.

struct \_Bits8Registers { // A character always has the size of 1 byte (8 bits).

unsigned char ah, al, bh, bl, ch, cl, dh, dl;

};

struct \_Bits16Registers { // Assuming short has a size of 2 bytes (16 bits).

unsigned short ax, bx, cx, dx;

};

union \_Registers {

struct \_Bits8Registers bits8Registers;

struct \_Bits16Registers bits16Registers;

};

### Bitfields

A bitfield is a struct with the possibility to define the size of the members in bits, which is useful in low-level applications. Only integral types can have bits, and the total number of bits must not exceed the number of bits an integer (eight times sizeof (int)). In the code below, the whole struct shares one byte (eight bits). The second field does not have a value; it only marks that the second bit shall be ignored.

struct Communicate {

int alert : 1;

int :1;

int read : 3;

int write : 3;

};

## Structures with function pointers

A struct can hold pointers to functions as well as values. However, in order to obtain the specific struct value, we have to add its address a pointer parameter[[21]](#footnote-21) in the function call.

#include <stdio.h>

struct Person { char name[100];

int age;

void (\*Print)(struct Person\*);

};

void PrintNameAndAge(struct Person\* personPtr) {

printf("Name: %s\n", personPtr->name);

printf("Age: %d\n\n", personPtr->age);

}

void PrintOnlyName(struct Person\* personPtr) {

printf("Name: %s\n\n", personPtr->name);

}

void PrintOnlyAge(struct Person\* personPtr) {

printf("Age: %d\n\n", personPtr->age);

}

void main() {

struct Person adam = {"Adam", 10, &PrintNameAndAge};

struct Person bertil = {"Bertil", 20, &PrintOnlyName};

struct Person ceasar = {"Ceasar", 30, &PrintOnlyAge};

adam.Print(&adam); // Prints the name and age.

bertil.Print(&bertil); // Prints only the name.

ceasar.Print(&ceasar); // Prints only the age.

}

## The Preprocessor

The preprocessor is a tool that precedes the compiler in interpreting the code. The #include directive is one of its parts. It opens the file and includes its text. So far, we have only included system header files, whose names are surrounded by arrow brackets (for instance, <stdio.h>). Later on, we will include our own header files. Then we will use parentheses instead of arrow brackets. The difference is that the pre-processor looks for the system header files in a special system directory while it looks for our header files in the local directory (usually the source code directory).

Another part of the pre-processor is the macros. They acts like functions with the difference that they do not perform any type checking, they just replace text. We have already used the assert macro. A macro is introduced with the #define directive and is often written with capitals.

#define ADD(a, b) ((a) + (b))

printf("%d\n", ADD(1 + 2, 3 \* 4)); // 15

Unlike regular C code, if we add a page-break we have to mark it by a backslash.

#define ADD(a, b) ((a) + \

(b))

It is also possible to perform conditional programming by checking the value of macros. In the following example, we define a system integer according to the underlying operational system.

#ifdef WINDOWS

#define SYSINT int

#endif

#ifdef LINUX

#define SYSINT unsigned int

#endif

#ifdef MAC

#define SYSINT long int

#endif

SYSINT iOpData = 0;

Condition programming can also be used as a third level of comments (above line and block comments, see Section 13.31). In the following code the whole segment between the #if and #endif directives will be omitted, regardless of the comments, as zero always is interpreted as false.

#if 0

int Square(int value)

{ return value \* value; // Square.

}

#endif

## The Standard Library

The C standard library hold around 200 functions and some macros divided into several sections.

### File Management

We can open, write to, read from, and close files with the help of file pointers. A file pointer is a pointer to a value of the FILE type; it can be considered a connection to the file. The program below reads a series of integers from the text file Input.txt and writes their squares to the file Output.txt. The function feof returns zero when there is no more value to be read from the file. Finally, we must not forget to close the file.

#include <stdlib.h>

#include <assert.h>

#include <stdio.h>

void main(void) {

FILE\* inFile = fopen("Input.txt", "r");

FILE\* outFile = fopen("Output.txt", "w");

assert(inFile != NULL);

assert(outFile != NULL);

while (!feof(inFile)) { double value;

fscanf(inFile, "%lf", &value);

fprintf(outFile, "%f\n", value \* value);

}

fclose(inFile);

fclose(outFile);

}

The text files are written in plain texts and can be viewed by the editor.

Input.txt

1

2

3

4

5

Output.txt

1

4

9

16

25

We can also read and write binary data. The program below writes the number one to ten to the file Numbers.bin and then reads the same series of values from the file. The functions write and read takes the address of the value to be read or written and the size of the value in bytes. They return the number of bytes actually read or written. When reading, we can check whether we have reached the end of the file by counting the number of read bytes; if it is zero, we have reached the end.

BinaryFile.c

#include <stdlib.h>

#include <assert.h>

#include <stdio.h>

void main(void) {

int index, value;

FILE \*outFile = fopen("Number.bin", "w"), \*inFile;

assert(outFile != NULL);

for (index = 1; index <= 10; ++index) { fwrite(&index, 1, sizeof index, outFile);

}

fclose(outFile);

inFile = fopen("Number.bin", "r");

assert(inFile != NULL);

while (fread(&value, 1, sizeof value, inFile) > 0) { printf("%d\n", value);

}

fclose(inFile);

}

The values are stored in compressed form in the binary file Numbers.bin, why they are not readable in the editor. Here is a screen dump of the file.



### Program Parameters

It possible to start the program execution with parameters. In main, the argc parameter holds the number of input strings, and argv holds the string themselves.

#include <stdio.h>

void main(int argc, char\* argv[]) {

int index;

printf("argc: %d\n", argc);

for (index = 0; index < argc; ++index) {

printf("argv[%d]: %s\n", index, argv[index]);

}

}

The parameters are given when the program executes. Note that the value of argv[0] always is the program path.



### Environment Functions

There is a set of functions that communicates with the surrounding environment: exit quits the execution and sends an integer value to the environment; atexit states a function that is called when exit is called; getenv reads the value of an environment variable, and system executes a system command. There are two macros [EXIT\_SUCCESS](http://www.cplusplus.com/reference/clibrary/cstdlib/EXIT_SUCCESS/) and [EXIT\_FAILURE](http://www.cplusplus.com/reference/clibrary/cstdlib/EXIT_FAILURE/) that can used with the exit function.

#include <stdio.h>

#include <stdlib.h>

void ExitFunction(void) {

printf("The Program is exiting.\n");

}

void main() {

char\* path = getenv("PATH");

system("dir \*.c");

atexit(&ExitFunction);

if (path != NULL) {

printf("Path: %s.\n", path);

exit(EXIT\_SUCCESS);

}

else {

printf("Path not found.\n");

exit(EXIT\_FAILURE);

}

}

Another way to send an exit message to the surrounding environment is to define int as main’s return type. It has the same effect as calling exit in main (except that the exit function will not be called).

#include <stdio.h>

#include <stdlib.h>

int main() {

char\* path = getenv("PATH");

if (path != NULL) {

printf("Path: %s.\n", path);

return EXIT\_SUCCESS;

}

else {

printf("Path not found.\n");

return EXIT\_FAILURE;

}

}

### Searching and Sorting

The bsearch function performs a binary search through a sorted list of values. We need to add a pointer function parameter specifying a function comparing two values.

#include <stdlib.h>

int CompareIntegers(const int\* intPtr1, const int\* intPtr2) {

return (\*intPtr1 < \*intPtr2) ? -1 : (((\*intPtr1 > \*intPtr2) ? 1 : 0));

}

void main() {

int intArray[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10},

arraySize = sizeof intArray / sizeof intArray[0],

value = 6;

int\* resultPtr = bsearch(&value, intArray, arraySize,

sizeof (int), &CompareIntegers);

if (resultPtr != NULL) {

int index = ((int\*) resultPtr) - intArray;

printf("Index %d.\n", index);

}

else {

printf("Not found.\n");

}

}

There is also the qsort function that performs a fast sorting.

#include <stdlib.h>

void main() {

int index, intArray[] = {7, 2, 5, 9, 3, 1, 10, 8, 4, 6},

arraySize = sizeof intArray / sizeof intArray[0],

key = 5;

qsort(intArray, arraySize, sizeof (int), &CompareIntegers);

for (index = 0; index < arraySize; ++index) {

printf("%d\n", intArray[index]);

}

}

### Functions with Variable Number of Parameters

In C, there is possible to define functions with a variable number of parameters (such as printf and scanf). One condition is that the function has at least one regular parameter in order for the va\_start macro to hook up to. The va\_arg macro reads the arguments and the va\_end macro ends the reading. Note that the macros do not give any information about the number or types of the arguments. In the following code, we use the first argument count to count the number of arguments thereafter.

#include <stdio.h>

#include <stdarg.h>

void Print(int count, ...) {

va\_list arg\_list;

va\_start(arg\_list, count);

while (count > 0) { int value = va\_arg(arg\_list, int);

printf("%d ", value);

--count;

}

printf("\n");

va\_end(arg\_list);

}

void main(void) {

Print(1, 1);

Print(2, 1, 2);

Print(3, 1, 2, 3);

}

### String Management

There are several functions dealing with strings: strcpy copies a string; strcat adds a string; strlen gives the length of a string; strcmp compares two strings; strchr and strrchr gives the first and last occurrence of a character in a string. All functions look for the finishing null character (‘\0’), strcmp returns a value less than zero if the first string is smaller than second string, a value greater than zero if the second string is larger, and zero if they are equals. Note that the comparison continues until the finishing zero character has been found, even though the character arrays may be longer. As strchr and strrchr return a pointer to the found character (or null if it was not found), we can calculate the array index by subtracting the pointers.

int FirstIndexOf(char s[], char c) {

char\* p = strchr(s, c);

return (p != NULL) ? (p - s) : -1;

}

int LastIndexOf(char s[], char c) {

char\* p = strrchr(s, c);

return (p != NULL) ? (p - s) : -1;

}

void main(void) {

char s1[] = "Hello", s2[] = "World";

char t1[20], t2[20];

strcpy(t1, s1);

strcat(t1, s2);

sprintf(t2, "%s, %s!", s1, s2);

printf("t1: \"%s\", length of t1: %d, t2: \"%s\"\n",

t1, strlen(t1), t2);

printf("Comparing t1 and t2: %d\n", strcmp(t1, t2));

printf("First Index of 'l' in t2: %d, last index of 'l' in t2: %d.\n",

FirstIndexOf(t2, 'l'), LastIndexOf(t2, 'l'));

}

It is also possible to write to a string with sprintf and read from a string with sscanf. These function come in handy when to cast between strings and numerical values.

void main() {

int i;

double x;

char s1[20], s2[20];

sprintf(s1, "%d", 123);

sprintf(s2, "%f", 123.456e-3);

printf("s1 = \"%s\", s2 = \"%s\"\n", s1, s2);

sscanf(s1, "%d", &i);

sscanf(s2, "%lf", &x);

printf("i = %d, x = %f\n", i, x);

}

The functions return the number of values written or read, which gives that we can check that the conversion worked.

if (sscanf(s2, "%lf", &x) == 1) {

printf("x = %f\n", x);

}

else {

printf("Could not cast the string \"%s\".\n", s2);

}

Moreover, we can also count the number of read character and decide whether to whole string has been read.

sscanf(s2, "%lf%n", &x, &charCount);

if (charCount == strlen(s2)) {

printf("x = %f\n", x);

}

else {

printf("Could not cast the string \"%s\".\n", s2);

}

### Memory Management

There is also a set of function for general memory management: memset sets each byte of a memory block to a given value; memcmp compares the memory blocks; memcpy and memmove copies a memory block. The difference between them is that memcpy copies the bytes of the block directly when memmove uses a buffer, which gives that memcpy is faster and memmove is safer when copying overlapping blocks.

These functions work similar to their string counterparts. However, note that they do not look for a finishing character. Instead, the sizes of the function blocks are given as a parameter.

#include <stdlib.h>

void main() {

int a1[3], a2[3];

memset(a1, 1, sizeof a1);

memcpy(a2, a1, sizeof a2);

printf("Comparing a1 and a2: %d\n", memcmp(a1, a2, sizeof a2));

}

### Mathematical Functions

C standard library includes the trigonometric cos, sin, tan, acos, asin, and atan, and atan2; the hyperbolic functions cosh, sinh, and tanh; ceil and floor that rounds a floating value to the closest higher and lower value, respectively; exp returning the natural exponent; log returning the natural logarithm; log10 returning the logarithm of ten; sqrt returning the square root, and pow returning the power of two values. All of these functions takes and returns a double value (except atan2 and pow that takes two values). The trigonometric functions deal with angles in radians.

When converting a floating value to an integral value, it will always be truncated; that is, rounded to the integer value closest to zero. One way to find the closest integer value is to add 0.5 to the double value before converting it.

printf("double %f, rounded to the closest integer value: %d\n",

x, ((int) (x + 0.5)));

### Long Jumps

Regular jumps with goto can only occur in the same function body; long jumps with the standard function setjmp and longjump can, on the other hand, occur throw call sequences; setjmp saves the state of the program execution and longjmp uses that information to restore that state, which gives the illusion of a backwards jump.

In the program below, the user inputs a number, if it does not equal zero its inverse is return. If the number does equal zero, a long jump occurs. The call to setjump returns zero when the jump is set and a non-zero value when a long jump occurs[[22]](#footnote-22).

#include <stdio.h>

#include <setjmp.h>

jmp\_buf env;

double Divide(double numerator, double denominator) {

if (denominator == 0) { longjmp(env, -1); // -1 to represents an error state.

}

return numerator / denominator;

}

double Inverse(double number) {

return Divide(1.0, number);

}

void main() {

if (setjmp(env) == 0) { double value;

printf("Value: ");

scanf("%lf", &value);

printf("Inverse: %f\n", Inverse(value));

}

else {

puts("Division by zero.");

}

}

### Time

The time function gives the number of seconds since January 1, 1970. That value can then be used to fill a tm struct with values regarding local date and time by calling the localtime function. If we call instead call gmtime, the [Greenwich Mean Time](http://sv.wikipedia.org/wiki/Greenwich_Mean_Time) (Coordinated Universal Time) will be given. Note that the tm\_year field gives the year starting from 1900.

#include <time.h>

void main() {

time\_t t = time(NULL);

struct tm time;

char \*months[] = {"Jan", "Feb", "Mar", "Apr",

"May", "Jun", "Jul", "Aug",

"Sep", "Oct", "Nov","Dec"};

char \*days[] = {"Sun", "Mon", "Tue", "Wed",

"Thu", "Fri", "Sat"};

localtime\_s(&time, &t);

printf("%02d:%02d:%02d\n", time.tm\_hour, time.tm\_min, time.tm\_sec);

printf("%s %s %d, %d\n", days[time.tm\_wday], months[time.tm\_mon],

time.tm\_mday, 1900 + time.tm\_year);

printf("Day of year: %d.\n", time.tm\_yday);

printf("Daylight saving time: %s.\n", time.tm\_isdst ? "Yes" : "No");

}

It is also possible to measure the period between two events.

#include <stdio.h>

#include <time.h>

void main() {

time\_t t1 = time(NULL);

int index;

for (index = 0; index < 100000; ++index) {

printf("%d\n", index);

}

{ time\_t t2 = time(NULL);

printf("The loop took %f seconds.\n", difftime(t2, t1));

}

}

### Random Numbers

It is possible to generate a sequence of pseudo-random numbers with the rand function. In order to initialize the sequence we need to sow a random seed with the srand function. One way to generate different seeds is to use the time function. The [RAND\_MAX](http://www.cplusplus.com/reference/clibrary/cstdlib/RAND_MAX/) macro gives the largest possible random numbers; however, usually we want random numbers in a certain interval. The following code generates five sequences of ten numbers between 1 and 100.

#include <stdlib.h>

void main() {

int iIndex1, iIndex2;

for (iIndex1 = 0; iIndex1 < 5; ++iIndex1) {

srand((unsigned int) time(NULL));

for (iIndex2 = 0; iIndex2 < 10; ++iIndex2) {

int randomNumber = rand() % 100 + 1;

printf("%d\n", randomNumber);

}

printf("\n");

}

}

### Limits of Integral and Floating Types

Since the type sizes are implementation dependent, there are a set of macros defining the minimum and maximum values of the integrated and floating types. The minimum values of unsigned integral types are always zero.

#include <stdio.h>

#include <limits.h> // The integral type limit constants.

#include <float.h> // The floating type limit constants.

void main() {

printf("Minimum signed char: %d\n", SCHAR\_MIN);

printf("Maximum signed char: %d\n", SCHAR\_MAX);

printf("Minimum signed short int: %d\n", SHRT\_MIN);

printf("Maximum signed short int: %d\n", SHRT\_MAX);

printf("Minimum signed int: %d\n", INT\_MIN);

printf("Maximum signed int: %d\n", INT\_MAX);

printf("Minimum signed long int: %ld\n", LONG\_MIN);

printf("Maximum signed long int: %ld\n\n", LONG\_MAX);

printf("Maximum unsigned char: %u\n", UCHAR\_MAX);

printf("Maximum unsigned short int: %u\n", USHRT\_MAX);

printf("Maximum unsigned int: %u\n", UINT\_MAX);

printf("Maximum unsigned long int: %lu\n\n", ULONG\_MAX);

printf("Minimum float: %e\n", FLT\_MIN);

printf("Maximum float: %e\n", FLT\_MAX);

printf("Minimum double: %e\n", DBL\_MIN);

printf("Maximum double: %e\n", DBL\_MAX);

printf("Minimum long double: %le\n", LDBL\_MIN);

printf("Maximum long double: %le\n", LDBL\_MAX);

}

### Character Functions

There are a set of functions for classifying a character.

if (isdigit(c)) {

printf("%c is a digit.", c);

}

Below follows a table of the character functions. The ASCII codes are stated within brackets.

|  |  |
| --- | --- |
| Function | Checks |
| **isalpha** | Letter: a-z and A-Z. |
| **islower** | Lower case letter: a-z. |
| **isupper** | Upper case letter: A-Z. |
| **isdigit** | Digit: 0-9. |
| **Isxdigit** | Hexadecimal digit: a-f, a-f, and 0-9. |
| **Isalnum** | Alphanumeric character: letter or digit. |
| **Isspace** | White-space character: space (0x20), horizontal tab (0x09), vertical tab (0x0B), new-line (0x0A), form-feed (0x0D), or carriage return (0x0C). |
| **Iscntrl** | Control character: characters with ASCII codes 0x00 to 0x0F, inclusive, and delete (0x1F). |
| **Isprint** | Printable character: any character that is not a control character. |
| **Isgraph** | Graphical character: any printable character that is not a white-space character. |
| **ispunct** | Punctuation character: any graphical character that is not an alphanumeric character. |

There is also the tolower and toupper functions that returns the character in lower and upper case, respectively.

printf("%c in lower case: %c, and in upper case: %c.",

c, tolower(c), toupper(c));

## Further Reading

I recommend *The C Programming Language* by Brian Kernighan and Dennis Ritchie (2nd Edition, Prentice Hall, 1988). It describes ANSI C and the standard library in a short and consist way, and holds an appendix about the ANSI C standard.

If you want to learn more about C programming, I recommend *C How to Program* by P. J. [Deitel](http://www.amazon.com/s/ref=ntt_athr_dp_sr_1?_encoding=UTF8&sort=relevancerank&search-alias=books&ie=UTF8&field-author=Paul%20J.%20Deitel) (6th Edition, Prentice Hall, 2009), which which discusses in depth theory as well as practice of programming in C, with many clear and comprehensive examples. C in a Nutshell by [Peter Prinz, Tony Crawford](http://shop.oreilly.com/product/9780596006976.do#tab_04) (O'Reilly Media, 2005) and Practical C Programming (3th edition, O'Reilly Media, 1997) are also good choices.

When you have mastered the basic of C programming, I recommend Expert C Programming by P. Van Der Linden (Prentice Hall, 1994). Do not worry; you do not have to be an expert to read it.

# Foreword

The compiler is made up of several components:

#### The Type System

Every programming language has a type system, which determines the properties of its types.

#### The Symbol Table

In many languages, the source code can hold a definition of a variables that is later referred to. In that case the, the variable (its name, type, and potential value) needs to be stored in a symbol table.

#### Scanning

The scanner takes a stream of characters and put them together into a sequence of *tokens*; that is, the least meaningful parts of the source code. For instance, the characters ‘i' and ‘f’ is put together into the keyword **if**, and the characters ‘!’ and ‘=’ is put together into the operator **not\_equal**.

#### Parsing and Middle Code Generation

The syntax of a programming language is often defined by a grammar, and the parser checks whether the token sequence generated by the scanner is accepted by the grammar. Moreover, the parser does also generate middle code.

#### Middle Code Optimization

The middle code generated by the parser often holds unnecessary instructions that needs to be removed.

#### Target Code Generation

When the middle code has been generated and optimized, it is time to generate the target code. In this book, we actually generate target code for two formats: Intel x86 assembly source code and executable code in the 16-bits COM-format.

#### Register Allocation

When the target code has been generated, the registers need to be allocated. We use a graph coloring algorithm to do that.

#### Linking

When each source file has been completely compiled, they need to be linked together into a executable file format.

**The Preprocessor**

The C programming language comes with a preprocessor that needs to be.

**The Standard Library**

The C programming language also comes with a standard library.

3. I have another book proposal for you, this time about compiler technology. The book will describe the design and implementation of a C++ compiler (including the preprocessor, linker, and standard library), with all code given.

So far, I have written a C compiler and I plan to extend it to a C++ compiler in the near future. Even though C++ is a much larger language than C, it not too much extra job. Most of the job has already been done with the C compiler.

The idea of the book is basically the same as the Windows book: I explain the details and provide all source code of the compiler. The source code written in Java with CUP and JLex and generates both assembly code and executable code.

The books below are the latest editions of three classic books. They describe in detail the basic features of compiler construction and to some extent advanced features. The structure of these books is similar to mine, one can say that I have followed these books. However, while these books give many small examples, my book describes in each chapter the features necessary to be included in a C++ compiler.

Alfred V. Aho, Lam, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compilers – Principles, Techniques, and Tools, 2nd edition. Prentice Hall, 2006.

Keith Cooper, Linda Torczon. Engineering a Compiler, 2nd edition. Morgan Kaufman, 2011.

Charles N. Fischer, Ron K. Cytron, Richard J. LeBlanc. Crafting a Compiler. Pearson Education, 2009.

The following books describe compiler construction in Java, C, and ML. They describe a compiler for a smaller language with excerpt from the code given in the book (the complete code is downloadable). However, these books are briefer than the books above, and describe the compiler features in less detail.

Andrew W Appel. Modern Compiler Implementation in Java, 2nd edition. Cambridge University Press, 2002.

Andrew W Appel. Modern Compiler Implementation in C, 2nd edition. Cambridge University Press, 2004.

Andrew W Appel. Modern Compiler Implementation in ML, 2nd edition. Cambridge University Press, 2004.

The following book could be said to be closely related to mine, as it also described the code for a C compiler. However, this book is more or less unreadable. It presents the code, which (in my option) is unstructured, and it does not include much text describing the code. Hopefully, my book will describe the topic in a clearer way. I also plan to describe theory of a compiler, not just the code to implement it. Moreover, this book does not include the preprocessor, linker, or standard library, and the code is written in C, not Java.

Christopher W. Fraser, David R. Hanson. A Retargetable C Compiler : Design and Implementation. Benjamin Cummings, 1995.

This books deals with the more advanced parts of compiler construction; that is, compiler optimization. Even though I will include some optimization at the end of my book, it does not compete with this book.

Steven S. Muchnick. Advanced Compiler Design & Implementation. Morgan Kaufmann, 2003.

Here is a suggestion for the chapters:

1. Introduction. Introduces the compiler phases described in the following chapters by demonstrating a compiler for a small toy language generating MIPS-code executable in the SPIM simulator.

2. The Scanner. The scanner is a relatively small part of the compiler. Its task is to put together characters into tokens (the smallest significant parts of the source code). Examples are as key words, operators, and numerical values. The scanner is written in JLex, a lexical generator for Java, based on Lex for C.

3. The Parser. The parser, on the other hand, is a large part of the compiler. Its task is to confirm that the given tokens (generated by the scanner) agree with the syntax of the programming language, which is represented by a set of grammatical rules. The parser is defined in CUP, a syntax generator for Java, based on Yacc for C. Each rule can also be equipped with code dealing with type checking and target code generation. However, I will have try to omit as much as possible of the code in this chapter. Most of the code is made up of calls to methods defined in later chapters.

4. Declarations and the Symbol Table. C++ has a rather complicated declaration system with aggregated types such as classes, structures and arrays with a corresponding complicated syntax. All defined variables and functions are stored in the symbol table, which is a hierarchical structure matching the program structure.

5. Type Checking. C++ has a rather large set of operators with complicated rules that need to be checked.

6. Intermediate Code Generation. When the types of expressions and statements are checked, three-address-code are generated, which is a simple intermediate language used to represent the code internally. Type conversation is also included in this chapter.

7. Static and Dynamic Initialization. In C++, variables can be initialized. If the variable is static, the data shall be generated and placed in the static area of the final target code. If it is dynamic, the initialization will result in a series of assignments. One thing that complicates the issue is that it is possible to initialize hierarchical structures made up by structures and arrays with one flat list.

8. Intermediate Code Optimization. The intermediate code can be optimized. For instance, code that is never reached and assignment of variables that are never used shall be removed.

9. Target Code Generation. The target code of the C++ compiler of this book is Intel x86, which is harder to deal with than the MIPS code the first chapter. It holds a few registers and registers of different sizes overlap. Therefore, the register allocation process needs to closely keep track on which variable values that are currently stored in the registers.

10. The Preprocessor. Before the actually compilation starts, the source code has been traversed by the preprocessor that replaces macros with text, includes header files, and provides conditional programming.

11. The Linker. When the target code has finally been generated, it becomes stored into an object file. As the source code can be distributed over several files, the target code need to be merged into one executable file. This is the task of the linker, it merges the code and data area, resolve the static and extern references and generate the executable file.

12. The C Standard Library, made up by functions and macros, which will be included by the linker in the final executable file.

13. The C++ Standard Library, made up by classes, to a certain extent is based on C standard library.

The compiler is made up of several phases:

Preprocessor

Source Code

Parser

Processed Code

Syntax Tree Optimister

Syntax Tree

Middle Code Generator

Optimized Syntax Tree

Middle Code Optimizer

Middle Code List

Optimized Middle Code List

Scanner

Symbol Table

Preprocessor

Source Code

Parser

Processed Code

Syntax Tree Optimister

Syntax Tree

Middle Code Generator

Optimized Syntax Tree

Middle Code Optimizer

Middle Code List

Optimized Middle Code List

Scanner

Symbol Table

1. BCPL stands for Basic Command Programming Language and was developed at join effort at London University and Cambridge University. Later on, BCPL has jokely been said to be an acronym for the Before C Programming Language. [↑](#footnote-ref-1)
2. Actually, their program wrote "hello, world\n" without a capital letter or an exclamation mark. Their example was inherited from the internal 1974 memorandum Programming in C: A Tutorial by [Brian Kernighan](http://en.wikipedia.org/wiki/Brian_Kernighan). The oldest known instance of the usage of the words ”hello” and ”world” together occurred in Kernighan’s 1972 tutorial *Introduction to the Language* [B](http://en.wikipedia.org/wiki/B_(programming_language)), even though the syntax was different from C. [↑](#footnote-ref-2)
3. Technically, the compiler replaces every block comment with a single space character, which makes it unwise to place a comment inside a name. [↑](#footnote-ref-3)
4. Usually, the problem is solved by the following macros (see Section 14.29):

   #define BOOL int

   #define TRUE 1

   #define FALSE 0 [↑](#footnote-ref-4)
5. Technically, it is given the value that happened to be stored on the memory address. [↑](#footnote-ref-5)
6. Depending of the compiler and its warning level settings. [↑](#footnote-ref-6)
7. Actually, f represent both float or double. The reason is that a float value is always converted to double when used as input to printf (or any other function). [↑](#footnote-ref-7)
8. As we inputs pointers to the variables rather than the variables themselves when calling scanf, no conversation from float to double occur. That is why f represents float and lf represents double. That lf (long float) represents double in scanf is an emergency solution since d represents an integer in decimal notation in both printf and scanf. [↑](#footnote-ref-8)
9. A white-space character is a space (‘ ‘), horizontal tabulator (‘\t’), vertical tabulator (‘\v’), new-line (‘\n’), form-feed (‘\f’), or carriage return (‘\r’). [↑](#footnote-ref-9)
10. Technically, each enumeration value holds the type unsigned int. [↑](#footnote-ref-10)
11. However, we have no knowledge about the actual address; it may be 10,000 or anything else. [↑](#footnote-ref-11)
12. In C++, the ampersand is also used to define references variables, which are not included in C. [↑](#footnote-ref-12)
13. More accurately, the number of bytes needed to hold a value of the type, [↑](#footnote-ref-13)
14. They were added to C in order to match two specific assembler instructions. Since then, they have stuck. More recent languages, such as C++, Java, and C#, do also support increment and decrement. [↑](#footnote-ref-14)
15. Some programmers prefer to denote equal to and not equal to as comparison operators. [↑](#footnote-ref-15)
16. In C#, the break statement is mandatory. [↑](#footnote-ref-16)
17. Technically, main is a regular function. The only difference, compared to other functions, is that the linker generates code that starts the program execution by calling main. Therefore, the program must include exactly one **main** function. [↑](#footnote-ref-17)
18. In C++, it is possible to access the global variable by prefixing colons, like ::number. [↑](#footnote-ref-18)
19. There is no rational explanation, just accept it. [↑](#footnote-ref-19)
20. Note the distinction of variable definition and declaration: a definition reserves memory for the variable while a declaration is simple a notification to the compiler. [↑](#footnote-ref-20)
21. In object-oriented languages the pointer is hidden and referred to as the this pointer. [↑](#footnote-ref-21)
22. This construction performs the same task as exception handling does in object-oriented languages. [↑](#footnote-ref-22)