My notes from K.N. King's "C Programming A Modern Approach" 2nd version

Piotr Marendowski

March 2023

Chapter 1

Note

In this material I will go over everything from book, trying to summarize every note-worthy subject. I will do it, while learning Latex, so good luck to me.

Contents

Chapter 2

C Fundamentals

2.1 Steps of Executing a C Program

Automated proccess:

1. **Preprocessing** - Preprocessor is executing directives (they begin with #).
2. **Compiling** - Compiler translates program into machine instructions (object code).
3. **Linking** - Linker combines object code and code needed for execution of the program.

2.2 The Genereral Form of a Simple Program

Simple C programs have this form:

directives
```
int main(void)
{
    statements
}
```

Directives - Begin with '#' symbol, they state what headers include to program.
Functions - They are segments of code that take arguments, and returns (or not) a value. Only `main` function is required.
Statements - Commands to execute, mostly end with semicolon.

String literal - Series of characters enclosed in double quotation marks, e.g. `"Hello world!"`.
New-line character - `\n` is an escape sequence, which advances to the next line of output.
Comments - Are ommited in program execution, can be used to comment single line e.g. `/* Comment */`, or block of lines. From C99 we can use one line comments e.g. `// Comment`.

2.3 Variables and Assigments

Variable - Place to store calculation's output, for using in future. Variable's characteristics:

- **Types** - For now, there are two types of variables:
  - `int` - Integer types, can store quite big whole number, but that depends on your computer's architecture.
  - `float` - Can store bigger numbers, as well as digits after decimal point.

- **Declarations** - To use a variable, we first need to declare it. It means that we need to specify variable's type, and name. We can chain declarations with the same type e.g. `int i, sum, x;`. In C99 they can now be declared after statements, not like in C89.
- **Assignment** - We assign value to a variable. Variable is on the left side, while value, expression, formula etc. is on the right side. To assign something to a variable, we first need to declare it. Examples:

```
int i;
float f;
i = 1;
f = 1.5;
```

Initialization

At the default most variables are uninitialized, which means that they have some random - garbage value assigned to them, if we did not. In declaration we can assign value to a variable, making it an **initializer**, e.g. `int i = 0;`.

2.4 Reading Input

For reading input we need to use `scanf` function, which needs a format string and value to read, e.g. `scanf("%d", &i);`.

2.5 Defining Names for Constants

To define a constant, we need to use a **macro definition**, which is interpreted by the preprocessor e.g. `#define WIDTH 20`.

2.6 Identifiers

Names in C are called **identifiers**. They can begin with the lower-case or upper-case letters or underscores e.g. `times10 my_var _done`. They cannot begin with a number e.g. `10times`. They cannot contain minus signs e.g. `my-var`.

Keywords

There are number of keywords, which are prohibited from using as identifiers.

2.7 Layout of the C Program

We can slice C statements into **tokens**:

```
printf   (   "Height:   %d\n"   ,   height   )   ;
  1      2      3          4     5     6      7   8
```

Tokens 1 and 2 are identifiers, token 3 is a string literal and tokens 2, 4, 6, and 7 are punctuation.
In most cases we can put many spaces between them. But we cannot put spaces within tokens e.g. `fl oat f;`.

Chapter 3

Formatted Input/Output

3.1 The `printf` Function

Needs a format string and arguments to insert there. There is no limit to these arguments. Format string could have conversion specifications, which are supplied by its arguments, to insert into format string e.g. `printf("Value:  %d", i);` where `%d` is a conversion specification and `i` is a value to be supplied to the format string.

3.2 Conversion Specifications

Outside of letter(s) specifying which type to covert to, they consist of the **Minimal Field Width (m)** and **Precision (p)**. They have the form of: `%m.pX`.

Minimal field width

Specifies the minimum number of characters to print. If the number of characters to print is less than specified, the number is right justified with spaces added. If the number of characters is greater than specified it will automatically expand to display all of characters.

Precision

Depends on the type to be displayed, reference the book for more detailed preview.

Conversion specifications:

- `d` - Displays an integer in a decimal (base 10) form. p indicates the minimum number of digits to display.
- `e` - Displays a floating-point number in exponential format. p indicates the number of digits after the decimal point.
- `f` - Displays a floating-point number without an exponent. p has the same meaning as previous.
- `g` - Displays a floating-point number in exponential format or fixed (without an exponent). p indicates the maximum number of significant digits to be displayed. It depends on the size of the number.

Escape sequences

They are characters, that would introduce problems in compilation or have some action to do e.g. insert new line into output. Few of them are:

- `\a` - alert (bell),
- `\b` - backspace,
- `\n` - new line,
- `\t` - tab,
- `\"` - qoute character,
- `\\` - slash character.

3.3 The `scanf` Function

This funcion handles input from stdin stream (keyboard), have a format string and may contain conversion specifications. The scanf call may look like that: `scanf("%d", &i);`. Scanf when reading an input ignores the white-space characters. It only matches input to the provided variables. If a reading error occurres, scanf will return immediately, ignoring the rest of the format string. It does not read the new-line character at the end of the input. If character cannot be read, function puts it back for the next variable and adds it to that.

Ordinary Characters in Format Strings

We can put a white-space characters into the format string, then scanf will read any number of white-space characters and discard them. When it encounters a non-white-space character it is trying to match it with an inputted character. If it fails, it returns.

Chapter 4

Expressions

4.1 Arithmetic Operators

Unary	Binary	
	Additive	Multiplicative
+ unary plus - unary minus	+ addition - subtraction	* multiplication / division % reminder

Operator Precendence and Associativity

Operator precendece is in what order C calculates expressions. The arithmetic operators have the following relative precendence:

Highest: + - (unary)
* / %
Lowest: + - (binary)

Associativity decides in what order operators with the same precendece are calculated. The binary operators are all left associative, whilst the unary operators are all right associative.

4.2 Assignment Operators

Are used to store a computed value of the expression.

Simple Assignment

Evaluates an expression, which then assigns into the variable, expression can be a *constant* (always has the same value). If they do not have the same type, the value of an expression is converted to the type of the variable. Assignments can be chained together e.g. `i = j = k = 0;`. The `=` operator is right associative.

Lvalues

Assignment operator requires that on its left side can only be a variable, not an expression e.g. `i + j = 0;` is wrong.

Compound Assignment

We can shorten statements e.g. `i += 2;` is equivalent to `i = i + 2;`. It works with the other operators including the following: `-= *= /= %=`, they all work in the same way and are right associative.

4.3 Increment and Decrement Operators

Used to even more shorten a compound addition and subtraction by 1. E.g.

```
i = i + 1;
j = j - 1;
```

Are the same as:

```
i += 1;
j -= 1;
```

Which are the same as:

```
i++;
j--;
```

They contain side effects - after adding or subtracting 1, value of their operands is modified. There are two types of these operators: **prefix** (`++i` or `--i`) which increments the variable first, then assigns value to `i`, and **postfix** (`i++` or `i--`) which first assings value to `i` and increments `i` after this statement.

Chapter 5

Selection Statements

We could group most statements in this three categories:

- **Selection statements** - Test provided condition, and execute code within their borders, e.g. `if` and `switch` statements.
- **Iteration statements** - Iterate over and over again, until their condition is not true, e.g. `for, while`, and `do while` statements.
- **Jump statements** - They control the flow of the program, can stop iterations, skip through them or jump to any place in the program, e.g. `break`, `continue` and `goto` statements.

5.1 Logical Expressions

Relational Operators

They are used to compare expressions, yelding 0 if statement are not true and 1 if statement are true.

Symbol	Meaning
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Equality Operators

Symbol	Meaning
==	equal to
!=	not equal to

Logical Operators

Symbol	Meaning	Operation
!	logical negation	Inverse - if false, returns 1
&&	logical end	If both expressions are true, returns 1
||	logical or	If either one of them them is true, returns 1

5.2 The `If` Statement

Has the form of: `if ( expression ) statement`. If evaluated expression has a non-zero value, then statement after parentheses is executed.

Compound Statements

We can "stack" multiple statements between the parentheses.

The `else` clause

If we want to execute statements when our expression is not true, we need to use a `else` clause. It has the following form: `if ( expression ) statement else statement`.

Cascaded `if` Statements

Thanks to them we can test multiple conditions. They have the following form:
`if ( expression ) statement else if ( expression ) statement else statement`.

Conditional Expressions

Has the following form: `epr1 ? epr2 : epr3`. Tests whether the first expression is true, if it is then executes the second expression, otherwise executes the third expression.

5.2.1 Boolean Values

C89

There is not a boolean type in C89, but we could declare a macro definition named `TRUE or FALSE`, e.g. `#define TRUE 1`.

C99

With the arrival of C99 we could declare `_Bool` type, e.g. `_Bool flag = true;`. For using this type we must declare a `#include <stdbool.h>`.

5.3 The `switch` statement

Switch statement can be a better-looking and faster alternative to the cascaded `if`. It has the following form:

```
switch ( expression ) {
    case constant-expression : statements
    ...
    case constant-expression : statements
    default : statements
}
```

Components of the `switch` statement:

- **Controlling expression** - could be an `int` or a `char`, but not `float` and strings.
- **Case lebels** - Form: `case constant-expression :` Cannot contain variables and function calls.
- **Statements** - Comes after each case label.

`Switch` statement does not need a `default` case. We use a `break` statement to stop at one choice, otherwise switch would execute all remaining cases.

Chapter 6

Loops

Loop is a repeatedly executing statements until the controlling expression is not true.

6.1 The `while` Statement

It has the following form: `while ( expression ) statement`. First it tests the controlling expression, then executes the loop body, if the expression is false, the loop terminates. It is possible that the loop body will not be executed at all, because it could be a false from the beginning.

6.2 The `do` Statement

It resembles the `while` loop, but it tests the controlling expression after each execution of the loop body, which makes it execute always at least once. It has the following form: `do statement while ( expression ) ;` .

6.3 The `for` Statement

Is used in a variety of ways, is ideal for counting loops. Has the following form: `for ( expr1 ; expr2 ; expr3 ) statement`. Every expression has its own role, in first you initialize or assign values to iterating variables. In the second you place a condition, which is checked every execution and while it is true loop will execute. In the third you put an operation which is performed at the end of each loop iteration. We could make more expressions (separated by the comma), but would need to place them in place according to their characteristics. Most of the `for` loops can be replaced by the `while` loop. We could omit any expression, but we need to compensate for it before or later. From C99 we could declare variables in the loop body, which will not be used as the variables already declared.

6.4 Exiting from a Loop

6.4.1 The `break` Statement

Is used to jump out of the loop body, terminating it. Can escape only one level of nesting.

6.4.2 The `continue` Statement

Only used in loops. Transfers control to the end of the loop, but it stayes in loop for the next execution. Can be considered a "skip to the end" statement.

The `goto` Statement

Jumps (transfers control) from its declaration to the label in the other part of the program. Label has the following form: `identifier : statement`. The `goto` statement has this form: `goto identifier ;`. Nowadays it is not commonly used, in favour of the `break, continue` and `return` statements.

6.5 The `null` Statement

We could omit the loop body (moving the loop body into expressions) or one of its expressions (which will create an infinite loop), e.g. `for (i = 0; ; i++) ...` .

Chapter 7

Basic Types

7.1 Integer Types

They are the whole numbers. We can divide them into two categories: `signed` and `unsigned`. The `signed` integers can be both positive and negative, but because of this they are two times less than `unsigned` type, which cannot contain negative numbers, thus making it one bit bigger. Sometimes we need bigger numbers, then we would use a `long` integer which on the most machines is double the `int` type. We could also store smaller integer values in the `short` type. Every type can be `signed` or `unsigned`, e.g. `unsigned short int`. We could abbreviate names by dropping the word `int`. Types are not required to have a certain bounds or maximum values, which varies from one architecture to another. For our machine's ranges we could see the `<limits.h>` header.

Integer Types in C99

From C99 we could now declare type `long long int`, which can be even two times the `long int` type.

Integer Constants

Constant numbers are numbers that cannot change. Can be of the base 10, 8 and 16:

- **Decimal** - Contain digits 0 through 9, cannot begin with a zero.
- **Octal** - Contain digits 0 through 7, must begin with a zero.
- **Hexadecimal** - Containg digits 0 through 9 and letter a to f, always begin with a 0x. Letters can be either lower case or upper case.

We could mix together any of these without any repercussions. As well as we could indicate that constant is for example `long int` - by adding `L or l` to the number. To indicate that constant is a `unsigned int` - add `U or u`. Both can be added together. In C99 we could specify the type `long long int` by adding `LL or ll` to the end.

If the result of the arithmetic operation is more than what desired type can store, we say that occured an **Integer Overflow**.

Reading and Writing Integers

For reading/writing operations we need to use the conversion specification for Unsigned types:

- **Unsigned** - `U` in decimal.
- **Octal** - `O`.
- **Hexadecimal** - `X`.

We use could use them with type conversion specifications by appending this letter to the front:

- **Short** - `H`.
- **Long** - `L`.
- **Long Long** (C99) - `LL`.

7.2 Floating Types

Numbers in the decimal form. There are three of them:

- **Float** - Single precision.
- **Double** - Double precision.
- **Long double** - Extended precision.

Mostly are stored according to The IEEE Floating-Point Standard.

Floating Constants

A floating constant must contain a decimal point and/or an exponent. The exponent (if present) must be preceded by the letter E or e. We could put the letter F or f (for `float`) or LF or lf (for `double`) at the end of the number to set the desired type.

Reading and Writing Floating-Point Numbers

In order to read a `double` value we use `lf, le, lg` (for `printf` without the l character). For the `long double` we used `Lf, Le, Lg`.

7.3 Character Types

For one character we use the type `char` which on the most machines corresponds to the ASCII table.

Operations on Characters

Because of the fact that `char` type is a really short int, we could take characters from the ASCII table and print them according to their number e.g. number 97 will be the character 'A'. We could also do some calculations on characters, for example by adding 1 to previous character we would get 'B'. Because of this characteristic we could compare characters and check if a number is for example greater than 'A' and less then 'Z'.

Signed and Unsigned Characters

C standard does not states that the `char` type is explicitly a `signed` or an `unsigned` type. Most of the times we do not care about it.

Escape Sequences

We use them for non-printable characters. To get them all we need to use a numeric escape which we could write in octal or hexadecimal:

- **Octal** - Does not begin with a zero, but with a backslash, after it comes a number from the ASCII character set in octal.
- **Hexadecimal** - Consists of the \x followed by the hexadecimal number.

An escape sequence must be enclosed in single qoutes e.g. `'\33'`.

Operations on Character types

To read or write on the character type, we need to use the `%c` conversion specification. To skip white spaces before reading a variable in the `scanf` function we would need to write it like that: `" %c"`.

We could also use the `putchar` function in order to write a single character. As well as to `getchar` to read one character. They are generally faster than `scanf` or `printf` function, because they are designed to only read one variable type and thus are smaller. If first there is `scanf` it will leave peaked (not assigned) variables and than calling `getchar` will read them.

7.4 Type Conversion

If we mix different types in an arithmetic expression, the compiler will convert them to the same type and then calculate on them. These conversions are called **implicit expressions**, because they are handled automatically. There are also **explicit expressions** which we could work with using the cast operator.

Casting

We could state that variable must convert to the stated type. It has the following form: `( type-name ) expression`. We use this to avoid overflowing, when one variable is converted to the smaller type than it can handle.

7.5 Type Definitions

Example: `typedef int Bool;` would make a `Bool` type which has the same characteristics as the `int` type.

7.6 The `sizeof` Operator

`sizeof ( type-name )` represents the number of bytes required to store a value belonging to type-name. Has type `size_t` which is unsigned integer type.

Chapter 8

Arrays

8.1 One-Dimensional Array

Is a data structure containing number of elements of the same type. Elements are arranged in a single row one after another, beginning with a zero element and ending with a n-1 element because of that. `int a[10];` declares one-dimensional array with 10 `int` elements. We use an array textbfsubscripting or **indexing** in order to access a particular element of the array, e.g. `a[0] = 10;`.

Array Initialization

`int a[3] = 1, 2, 3` If we initialize less than total number of elements, the rest will be zeroed. Thanks to that we could initialize all elements to zero: `int a [3] = 0` . We could also omitt the length of the array if the initializer is present.

8.2 Multidimensional Arrays

We could create arrays with any number of dimensions:

```
int m[2][2] = {{1, 2}
               {3, 4}};
```

Which will have two rows and colums. They are stored in a row-major order (one row after another in a continuous block of memory). We could also declare them to be constant with a keyword `const` to not permitt any modification of them.

8.3 Variable-Length Arrays

C99 feature, which can be used to supply not constant number to the declaration. For example we could first ask to specify a length and then declare array with that length.

Chapter 9

Functions

9.1 Defining and Calling Functions

```
double average(double a, double b)
{
    return (a + b) / 2;
}
```

Where `double` is the **return type** of the function, `a` and `b` are **parameters** supplied from the calling function to be used in this function. The `return` statement is in loop **body** which will be executed. To call this function we need to enter a function name followed by the list of **arguments**: `average(x, y);`.

9.1.1 Function Definitions

General function definition:

```
return-type function-name ( parameters )
{
    declarations
    statements
}
```

Functions cannot return arrays. If `return-type` is `void` function doesn't returns a value. If there is not return value, type is `int` in C99, while in C99 it is illegal. If function has zero parameters it should contain the keyword `void`. Void function's body can be empty.

Function Calls

Consists of a function name and parameters enclosed in parentheses (if there are not any parameters, we we need to write parentheses without anything inside).

9.2 Function Declarations

Is a first line of the function and is used for providing information about function which could be written as a whole below the `main` function. It must be consistent with the function's definition. They are known as **function prototypes**.

9.3 Arguments

Parameters are in function definitions and arguments are in function calls. Arguments are passed by value which means that they won't be affected after function execution.

Array Arguments

When supplying an array compiler does not know its length so we need to add a length argument. In function calls we only pass the array name as an argument. In contradiction we can modify an array when passing it into the function.

Variable-Length Array Parameters

To use VLAs as parameters we need to first specify parameter which will go into this VLA.

Compound Literals

```
total = sum_array((int []){3, 0, 3, 4, 1}, 5);
```

Makes this array "on the fly" to be supplied into `sum_array` function.

9.4 The `return` Statement

`return expression ;`
If there is for example a `double` variable in the `return` statement, it will be converted to the function's return type. For `void` functions `return` statement is not necessary.

9.5 Program Termination

The `main` function should return 0 if terminated successfully.

The `exit` Function

`exit`0 `or exit`$EXIT_SUCCESS$ - normal termination, `exit`1 `or exit`$EXIT_FAILURE$ indicates abnormal termination. This function and macros are declared in `stdlib.h` header.

9.6 Recursion

A function is recusive if it calls itself.

Chapter 10

Program Organization

10.1 Local Variables

Variable declared in the body of a function is said to be local to the function, which means that it cannot be seen by other functions and used by them. It has an Automatic storage duration which means that it will be automatically allocated and deallocated at the function's return. It also has a block scope which means that it can only be referenced inside this function.

Static Local Variables

Putting the word `static` causes it to have static storage duration which means that it retains its value throughout the whole program execution. But it is still hidden from other functions.

10.2 External Variables

Are declared outside the body of any function. Have the static storage duration and the file scope. There are many dangers of using it e.g:

- If we would change its type we would need to check every function using it.
- If it will have assigned an incorrect value there will be a problem to identify a guilty function.
- Functions using it are hard to reuse in other programs.

10.3 Blocks

We could declare a variable in a e.g. an `if` statement, then this variable will have the block scope and will not be able to be referenced outside this scope.

10.4 Scope

In a C program, the same identifier may have several different meanings. When a declaration inside a block names an identifier that's already visible, the new declaration temporarily "hides" the old one, and the identifier takes on a new meaning. At the end of the block, the identifier regains its old meaning. If you go deeper, the most relevant variable will be used. There are file and block scopes.

10.5 Organizing a C Program

Rules to organize a program:

- A preprocessing directive does not take effect until the line on which it appears.

- A type name cannot be used until it is been defined.
- A variable cannot be used until it is declared.

Chapter 11

Pointers

11.1 Pointer Variables

Every byte has a unique memory address. If a variable consists of more than one byte, then its address is the first byte occupied by it. Pointer variables "point" (store) to this address in memory.

Declaring Pointer Variables

We declare it by adding an asterisk by the name: `int *p;` which points only to the `int` variables. Pointers can point to any type or a block in memory.

11.2 The Address and Indirection Operators

If `x` is a variable, then its address is `&x`. To gain access to the object (value) that a pointer points to, we use the (indirection) operator.

The Address Operator

We could initialize pointer in declaration to point to some value e.g. `int i, p = &i;`.

The Indirection Operator

We could access the value pointed to e.g. `printf("%d\n", p);` will display the value of `i` not its address. By changing the value of `p` we will change the value `i`.