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 didn't. 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 doesn't 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 don't 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