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

1 Note **2**

2 C Fundamentals **6**
2.1 Steps of Executing a C Program . . . 6
2.2 The General Form of a Simple Program . . . 6
2.3 Variables and Assigments . . . 6
2.4 Reading Input . . . 7
2.5 Defining Names for Constants . . . 7
2.6 Identifiers . . . 7
2.7 Layout of the C Program . . . 7

3 Formatted Input/Output **8**
3.1 The `printf` Function . . . 8
3.2 Conversion Specifications . . . 8
3.3 The `scanf` Function . . . 9

4 Expressions **10**
4.1 Arithmetic Operators . . . 10
4.2 Assignment Operators . . . 10
4.3 Increment and Decrement Operators . . . 11

5 Selection Statements **12**
5.1 Logical Expressions . . . 12
5.2 The `If` Statement . . . 12
5.2.1 Boolean Values . . . 13
5.3 The `switch` statement . . . 13

6 Loops **14**
6.1 The `while` Statement . . . 14
6.2 The `do` Statement . . . 14
6.3 The `for` Statement . . . 14
6.4 Exiting from a Loop . . . 14
6.4.1 The `break` Statement . . . 14
6.4.2 The `continue` Statement . . . 14
6.5 The `null` Statement . . . 15

7 Basic Types **16**
7.1 Integer Types . . . 16
7.2 Floating Types . . . 17
7.3 Character Types . . . 17
7.4 Type Conversion . . . 18

7.5 Type Definitions 18
7.6 The `sizeof` Operator 18

8 Arrays **19**
8.1 One-Dimensional Array 19
8.2 Multidimensional Arrays 19
8.3 Variable-Length Arrays 19

9 Functions **20**
9.1 Defining and Calling Functions 20
9.1.1 Function Definitions 20
9.2 Function Declarations 20
9.3 Arguments 20
9.4 The `return` Statement 21
9.5 Program Termination 21
9.6 Recursion 21

10 Program Organization **22**
10.1 Local Variables 22
10.2 External Variables 22
10.3 Blocks 22
10.4 Scope 22
10.5 Organizing a C Program 22

11 Pointers **24**
11.1 Pointer Variables 24
11.2 The Address and Indirection Operators 24
11.3 Pointer Assignment 24
11.4 Pointers as Arguments 24
11.5 Pointers as Return Values 24

12 Pointers and Arrays **25**
12.1 Pointer Arithmetic 25
12.2 Using Pointers for Array Processing 25
12.3 Using an Array Name as a Pointer 26
12.4 Pointers and Multidimensional Arrays 26
12.5 Pointers and Variable-Lenght Arrays (C99) 26

13 Strings **27**
13.1 String Literals 27
13.2 String Variables 27
13.3 Reading and Writing Strings 27
13.4 Accessing the Characters in a String 28
13.5 Using the C String Library 28
13.6 String Idioms 28
13.7 Arrays of Strings 29

14 The Preprocessor **30**
14.1 How the Preprocessor Works 30
14.2 Preprocessing Directives 30
14.3 Macro Definitions 30
14.4 Conditional Compilation 31
14.5 Miscellaneous Directives 33

15 Writing Large Programs **34**
15.1 Source FIles 34
15.2 Header Files 34
15.3 Building a Multiple-File Program 35

16 Structures, Unions, and Enumerations **36**
16.1 Structure Variables 36
16.2 Structure Types 37
16.3 Nested Arrays and Structures 37
16.4 Unions 38
16.5 Enumerations 38

17 Advanced Uses of Pointers **39**
17.1 Dynamic Storage Allocation 39
17.2 Dynamically Allocated Strings 39
17.3 Dynamically Allocated Arrays 39
17.4 Deallocating Storage 40
17.5 Linked Lists 40
17.6 Pointers to Functions 41
17.7 Restricted Pointers (C99) 41
17.8 Flexible Array Members (C99) 41

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 General 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 the 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 the future use. Variable's characteristics:

- **Types** - For now, there are two types of variables:
  - `int` - Integer types, can store quite big whole numbers, but that depends on your computer's architecture.
  - `float` - Can store bigger numbers, as well as digits after the 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, unlike in C89.
- **Assignment** - Assigns 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. 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 the `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, 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. Most of the time, we could 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 how much of arguments could be there. 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 this number is less than specified, then the number is right justified with spaces added. If the number of characters is greater than specified, then it will automatically expand to display all of the 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 the 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. 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 the standard input 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 occurs, 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 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 tries to match it with an inputted character. If it fails, it returns this variable without assigning to it anything.

Chapter 4

Expressions

4.1 Arithmetic Operators

<table>
<tr><th rowspan="2">Unary</th><th colspan="2">Binary</th></tr>
<tr><th>Additive</th><th>Multiplicative</th></tr>
<tr><td>+ unary plus
- unary minus</td><td>+ addition
- subtraction</td><td>* multiplication
/ division
% reminder</td></tr>
</table>

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 on its left side 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 a side effects - after adding or subtracting 1, values of their operands are 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 condition's borders, e.g. `if` and `switch` statements.
- **Iteration statements** - Iterate over and over again, until the 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 is not true and 1 if it is.

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 an `#include <stdbool.h>` directive.

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** - Come 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 would 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. 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 the first you could initialize or assign values to variables. In the second you place a condition, which is checked every execution and while it is true the 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 commas), 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 stays in the 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 maximally can only hold two times less than the `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 int` 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. Could 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 letters E or e. We could put the letters 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 the `char` type is a really short int, we could take characters from the ASCII table and print them according to their numbers 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

The C standard does not state that the `char` type is explicitly a `signed` or an `unsigned` type. Most of the time 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 escapes 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 a 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 `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 there is a `scanf` it will leave peaked (not assigned) variables and than calling the `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 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 desired 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

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 **subscripting** 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 the 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 permit any modification of them.

8.3 Variable-Length Arrays

C99 feature, which could be used to supply a non-constant number to the declaration. For example we could first ask to specify a length and then declare array with that length, by writing in the place of the array's lenght, that variable.

Chapter 9

Functions

9.1 Defining and Calling Functions

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

`double` is the **return type** of the function, `a` and `b` are **parameters** supplied from the call of the function to be used in this function. The `return` statement is in the 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 return a value. If there is not a return value, type is `int` in C89, 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 the parentheses (if there are not any parameters, we need to write parentheses without anything inside).

9.2 Function Declarations

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 the guilty function.
- Functions using it are hard to reuse in other programs.

10.3 Blocks

We could declare a variable in 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 has 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`.

11.3 Pointer Assignment

`int p = &i;` - copies the address of i into p. `q = p` copies the contents of p (the address of i) into q, so now q points to i. We could change i by chaging q and p.

11.4 Pointers as Arguments

Pointers can be used as function's arguments (as aliases to variables in calling function) and be used to modify values inside the function and store them outside. We could declare p to be constant `const p;` which means that it cannot be changed.

11.5 Pointers as Return Values

Pointers (as aliases to other variables or external variables or static variables) could also return from functions.

Chapter 12

Pointers and Arrays

12.1 Pointer Arithmetic

```
int a[10], *p;
p = &a[10];
p = 5;
```

p points to first element in array and assigns to it 5.

Adding an Integer to a Pointer

```
int a[10], p, q, i;
p = &a[2];      /* p points to the 2nd place */
q = p + 3;      /* q points to 5th (p + 3) */
p += 6;         /* p points to 8th (2 + 6) */
```

Is equivalent to `a[i + j];`.

Subtracting an Integer to a Pointer

```
int a[10], p, *q, i;
p = &a[8];      /* p points to 8th place */
q = p - 3;      /* q points to 5th (p - 3) */
p -= 6;         /* p points to the 2nd (8 - 6) */
```

Is equivalent to `a[i - j]`.

Subtracting One Pointer from Another

The result is distance between the pointers.

```
p = &a[5];
q = &a[1];

i = p - q;    /* i is 4 */
i = q - p;    /* i is -4 */
```

12.2 Using Pointers for Array Processing

We could iterate through the whole array by incrementing the pointer itself.

12.3 Using an Array Name as a Pointer

Array subscripting can be viewed as a form of pointer arithmetic. We cannot directly change where one element points to, but we could create a new pointer to this array and make this element point elsewhere.

```
int a[10];
a = 7;         /* stores 7 in a[0] is equivalent to &a[0] */
(a+1) = 12;    /* stores 12 in a[1] is equivalent to a[1] */

/* easier way to calculate sum */
for (p = a; p < a + N; p++)
    sum +=p;
```

Arrays passed to functions always are treated as pointers and are not copied in contrast to that array is not protected agains change. Ordinary values are copied, but arrays are not. We could pass a slice of the whole array to a function. We could also create new pointer which points to an array and perform pointer arithmetic on it which will be as we operated on this array itself.

12.4 Pointers and Multidimensional Arrays

Arrays are stored in row-major order, which means that no matter how many dimensions they have, they all are stored in one big line, which could be operated through pointer arithmetic.

12.5 Pointers and Variable-Lenght Arrays (C99)

Pointers can point to VLAs (any dimensions). With multi-dimensional arrays pointer need to have the type of the last dimension `int a[m][n], (*p)[n];`.

Chapter 13

Strings

13.1 String Literals

A series of characters enclosed within double quotes e.g. `"Hello, World!"`.

How Strings Literals Are Stored

C treats string literals as character arrays. C compiler sets n + 1 characters for literal and adds a **null character** (`\0` escape sequence) at the end which is used to indicate end of this literal. So compiler treats it like a `char` type.

13.2 String Variables

We need to declare a string one character longer to leave space for the null character at the end. Initialize a string variable:
`char date[8] = "June 14";`
Where `"June 14"` is not a string literal, but an initialization of string `date`. If initializer is shorter than number of elements in the array, the leftover elements will be null.

If there is no room for string, it will be cut and no null character will be assigned (meaning that it will not be usable as a string). We could also omit setting the initializer, then the compiler will calculate length of the array for us (adding the null character at the end), but this length will be fixed and cannot be changed.

Character Arrays versus Character Pointers

```c
char date1[] = "June 14";
char date2 = "June 14";
```

`date1` is an array, while `date2` is a pointer. Array elements can be modified, but string literal pointed to by `date2` cannot. `date1` is an array name, while `date2` is a pointer which could point to other strings during program execution.

13.3 Reading and Writing Strings

Writing Strings Using `printf` and `puts`

```c
printf("%s\n", str);
puts(str);
```

`printf` using the `%s` conversion specification reads characters one by one until it finds the null character (if not found will continue to read out of bounds memory locations). `puts` after reading a string always advances to the next line (by printing a new-line character).

Reading Strings Using **scanf** and **gets**

```
scanf("%s", str);
gets(str);
```

`str` in `scanf` is treated like a pointer so there is no need for the address operator (`&`). When it is called it discards white-space characters and writes into `str` every character until it encounters a white-space, will always store a null character at the end. A new-line, space or a tab character in the middle of input will cause `scanf` to stop reading. `gets` reads the whole line of input and stores a null character at the end, it does not skip white-spaces and reads until it finds a null character. There is a better and safer alternative of `gets` - `fgets` which has a length parameter and cannot go over this length.

13.4 Accessing the Characters in a String

We could use the array subscripting or pointer arithmetic to process strings. Pointers simplify this. There is no difference between string parameter being declared as an array or as a pointer.

13.5 Using the C String Library

Header `string.h` includes many functions that are helpful for processing strings.

Short Explanation of Some Functions

- `char *strcpy(char s1, char s2)` - copies the string s2 (until first null character) into string s1 and returns s1. Function `strncpy` has a third argument which is the size of the string, is safer but slower.
- `size_t strlen(const char *s)` - returns the length of the string (number of characters is up to, but not including, the first null character).
- `char *strcat(char *s1, const char *s2)` - appends the contents of the string s2 to the end of the string s1; it returns s1 (a pointer to the resulting string). There is also a `strncat` function.
- `int strcmp(const char *s1, const char *s2)` - compares the strings s1 and s2, returning a value less than, equal to, or greater than 0, depending on whether s1 is less than, equal to, or greater than s2.

13.6 String Idioms

Searching for the End of a String:

```
while (*s++)
    ;
```

Copying a String:

```
while (*p++ = *s2++)
    ;
```

13.7 Arrays of Strings

To fight the inefficiency in storing arrays of strings we need to use the `ragged array` - a two-dimensional array whose rows can have different lengths. This can be achived by creating the array of pointers to strings, where every element is a pointer to one string which could be of any size. Then these string can be accessed simply by the array's subscript.

Command-Line Arguments

To obtain access to command-line arguments (called program parameters), we must define `main` as a function with two parameters:

```
int main(int argc, char *argv[])
{
    ...
}
```

`argc` ("argument count") is the number of command-line arguments (including the name of the program itself). `argv` ("argument vector") is an array of pointers to the command-line arguments, which are stored in string form. `argv` has one additional element `argv[argc]` which is always a null pointer.

Chapter 14

The Preprocessor

`#define` and `#include` (and any that begin with a `#` character) directives are handled by the preprocessor, a piece of software that edits C programs just prior to compilation.

14.1 How the Preprocessor Works

The `#define` directive defines a macro - a name that represents something else, e.g. `#define WIDTH 20`. The `#include` directive tells the preprocessor to open a particular file and "include" its contents as part of the file being compiled.

14.2 Preprocessing Directives

Most preprocessing directives fall into one of three categories:

- **Macro definition** - The `#define` directive defines a macro; the `#undef` directive removes a macro definition.
- **File inclusion** - The `#include` directive causes the contents of a specified file to be included in a program.
- **Conditional compilation** - The `#if, #ifdef, #ifndef, #elif, #else` and `#endif` directives allow blocks of text to be either included in or exclude from a program, depending on conditions that can be tested by the preprocessor.

Rules that apply to all directives:

- Directives always begin with the `#` symbol.
- Any number of spaces and horizotal tab characters may separate the tokens in a directive.
- Directive always end at the first new-line character, unless explicitly continued (continue by adding a \ character at the end of the line).
- Directives can appear anywhere in a program.
- Comments may appear on the same line as a directive.

14.3 Macro Definitions

Simple Macros

Have the form: `#define identifier replacement-list`. `replacement-list` is any sequence of preprocessing tokens.

Parameterized Macros

Have the form: `#define identifier( x1 , x2 , ... , xn ) replacement-list`. Where `x1, x2, ..., xn` are identifiers (the macro's parameters). The parameters may appear as many times as desired in the replacement-list.

For example. we've defined the following macro:
`#define MAX(x,y) ((x)>(y)?(x):(y))`
And now we invoke it:
`i = MAX(a, b);`
What we get is:
`i = ((a)>(b)?(a):(b))`
Which works as a simple function.

The # Operator

The # operator converts a macro argument into a string literal ("stringization"). We basically use it to print the inputted string into the outputted one.

`#define PRINT_INT(n) printf(#n " = %d\n", n)`
Invoking it:
`PRINT_INT(i);`
Will create:
`printf("i = %d\n", i);`

The ## Operator

Can "paste" two tokens together to form a single token.

`#define MK_ID(n) i##n`
Invoking it:
`MK_ID(1)`
The preprocessor will join i and 1 to make a single token (i1):
`int MK_ID(1);`
After preprocessing, this declaration will become:
`int i1;`

General Properties of Macros

Are the following:

- A macro's replacement list may contain invocations of other macros.
- The preprocessor replaces only entire tokens, not portions of tokens.
- A macro definition normally remains in effect until the end of the file in which it appears.
- A macro may not be defined twice unless the new definition is identical to the old one.
- Macros may be "undefined" by the #undef directive.

14.4 Conditional Compilation

The inclusion or exclusion of a section of program text depending on the outcome of a test performed by the preprocessor.

The **#if** and **#endif** Directives

For example:

```
#define DEBUG 1
...
#if DEBUG
printf("Value of i: %d\n", i);
printf("Value of i: %d\n", j);
#endif
```

Will only leave out the `printf` calls when the DEBUG is 1.

The **defined** Operator

Produces the value 1 if the identifier is a currently defined macro; it produces 0 otherwise. E.g.

```
#define DEBUG 1
...
#if defined(DEBUG)       // or #if defined DEBUG
...
#endif
```

The **#ifdef** and **#ifndef** Directives

The `#ifdef` directive tests whether an identifier is currently defined as a macro. E.g.

```
#define DEBUG 1
...
#ifdef DEBUG
...
#endif
```

The `#ifndef` directive tests wheter an identifier is not defined as a macro.

The **#elif** and **#else** Directives

For example:

```
#if expr1
...
#elif expr2
...
#else
...
#endif
```

The `#ifdef` and `#ifndef` directives could also be used.

Uses of Conditional Compilation

- Writing programs that are portable to several machines or operating systems.
- Writing programs that can be compiled with different compilers.
- Providing a default definition for a macro.
- Temporarily disabling code that contains comments.

14.5 Miscellaneous Directives

The #error Directive

If the preprocessor encounters an #error directive, it prints an error message. Often used with conditional compilation. The message is any sequence of tokens, not a string literal. E.g.

```
#if INT_MAX < 100000
#error int type is too small
#endif
```

Will produce: `Error directive: int type too small`

The #line Directive

Is used to alter the way program lines are numbered. Has two forms: `#line n` and `#line n "file"`. Used rarely.

The #pragma Directive

Provides a way to request special behavior from the compiler. Has the form: `#pragma tokens`.

The _Pragma Operator

Has the form: `_Pragma ( string literal )`

Chapter 15

Writing Large Programs

15.1 Source FIles

A C program can consists of many source files (parts of it with a .c extension). One of which must contain a `main` function.

15.2 Header Files

The `#include` directive tells the preprocessor to open a specified file and insert its contents into the current file. The have the extension .h. In these header files we can include type definitions, function prototypes, and variable declarations so that other source files can see and use these.

The `#include` Directive

Has two forms, first is used for header files that belong to C's own library:

```
#include <filename>
```

The second form is used for all other header files, including any that we write:

```
#include "filename"
```

Sharing Variable Declarations

To declare i without defining it, we must put the keyword `extern` at the beginning of its declaration:

```
extern int a[];
```

We first put a definition of i in one file:

```
int i;
```

If i needs to be initialized, the initializer would go here. When this file is compiled, the compiler will allocate storage for i. The other files will contain declarations of i:

```
extern int i;
```

Because of the word `extern`, however the compiler does not allocate additional storage for i each time one of the files is compiled.

Protecting Header Files

For example, we define a `boolean.h`:

```
#ifndef BOOLEAN_H
#define BOOLEAN_H

#define TRUE 1
#define FALSE 0
typedef int Bool;

#endif
```

15.3 Building a Multiple-File Program

Makefiles

A file containing the information necessary to build a program. Also describes dependencies among the files. Are handy for compiling and linking multiple-file programs and partial compilation. For description of the `Makefile`, confront pages 367 - 368.

Defining Macros Outside a Program

Most compilers (including GCC) support the -D option, which allows the value of a macro to be specified on the command line:
`gcc -D DEBUG=1 foo.c`.

Chapter 16

Structures, Unions, and Enumerations

16.1 Structure Variables

The members of the structure aren't required to have the same type. To select a member we specify its name, not its position.

Declaring Structure Variables

```
struct {
    int number;
    char name[NAME_LEN+1];
    int on_hand;
} part1, part2;
```

There are two parts - two blocks of memory with all these variables. The members of a structure are stored in memory in order in which they are declared. Each structure represents a new scope; any names declared in that scope won't conflict with other names in a program.

Initializing Structure Variables

```
struct {
    int number;
    char name[NAME_LEN+1];
    int on_hand;
} part1 = {528, "Disk Drive", 10},
  part2 = {914, "Printer Cable", 5};
```

The values in the initializer must appear in the same order as the members of the structure.

Operations on Structures

```
printf("Part number:  %d\n", part1.number);
part1.number = 258;
scanf("%d", &part1.on_hand);
par2 = part1; \\copies every element from part1 to part2
```

16.2 Structure Types

Declaring a Structure Tag

Is a name used to identify a particular kind of structure.

```
struct part {
    int number;
    char name[NAME_LEN+1];
    int on_hand;
};
```

Then we can use it to declare variables:
`struct part part1, part2;`

But we could also combine them in declaration:

```
struct part {
    int number;
    char name[NAME_LEN+1];
    int on_hand;
} part1, part2;
```

Then they are compatible:
`struct part part1 = {528, "Disk Drive", 10};`
`par2 = part1;`

Defining a Structure Type

```
typedef struct {
    int number;
    char name[NAME_LEN+1];
    int on_hand;
} Part;

Part part1, part2; \\not struct Part, because of typedef
```

16.3 Nested Arrays and Structures

Nested Structures

In one structure we can declare an array, and then include this structure as a part of a larger structure. Then we would go in the reverse order (from the largest) to access an array member.

Arrays of Structures

`struct part inventory[100];`
Is capable of storing information about 100 parts. To access it:
`print_part(inventory[i]);`
To assign 883 to the number member:
`inventory[i].number = 883;`
To access a singe character in a part name:
`inventory[i].name[0] = '\0';`

16.4 Unions

The compiler allocates only enough space for the largest of the members, which overlay each other within this space. As a result, assigning a new value to one member alters the values of the other members as well.

```
union {
    int i;
    double d;
} u;

\\ occupy most of the union (because int is less than double)
u.i = 82;
\\ alter i and now union stores only d
u.d = 74.8;
```

16.5 Enumerations

An enumerated type is a type whose values are listed ("enumerated") by the programmer, who must create a name (an enumeration constant) for each of values.

Enumeration Tags and Type Names

```
enum suit {CLUBS, DIAMONDS, HEARTS, SPADES};
enum suit s1, s2;
```

Enumerations as Integers

By the default enumeration variables are assigned to numbers 0, 1, 2, ... We could declare them in the other order:

```
enum suit {CLUBS = 1, DIAMONDS = 2, HEARTS = 3, SPADES = };
```

Chapter 17

Advanced Uses of Pointers

17.1 Dynamic Storage Allocation

Dynamic storage allocation - the ability to allocate storage during program execution. We use it to design various data structures.

Memory Allocation Functions

From `<stdlib.h>` header:

- `malloc` - Allocates block of memory but doesn't initialize it.
- `calloc` - Allocates block of memory and clears it.
- `realloc` - Resizes a previously allocated block of memory.

Null Pointers

Function returns a null pointer when cannot satify a request for memory allocation. E.g.

```
if ((p = malloc(10000)) == NULL) {
    /* allocation failed; take appropriate action */
}
```

17.2 Dynamically Allocated Strings

Using **malloc** to Allocate Memory for a String

Which has the following prototype:
`void *malloc(size_t size);`
Allocates a block of `size` bytes and returns a pointer to it.

```
p = malloc(n + 1);     // p is an empty string (one for the null char)
strcpy(p, "abc");      // initialize it
```

17.3 Dynamically Allocated Arrays

Using **malloc** to Allocate Storage for an Array

```
int *a;
a = malloc(n * sizeof(int));
```

The `calloc` Function

`calloc` allocates space for an array with `nmemb` elements, each of which is `size` bytes long. Initializes all elements to 0.

```
void *calloc(size_t nmemb, size_t size);

a = calloc(n, sizeof(int));
```

The `realloc` Function

`realloc` resizes to a new `size` a block pointed to by the `ptr` pointer.

```
void *realloc(void *ptr, size_t size);

a = calloc(n, sizeof(int));
```

17.4 Deallocating Storage

In order to collect garbage and prevent memory leaks, we need to use the `free` function.

The `free` Function

```
void free(void *ptr);

p = malloc(...);
q = malloc(...);
free(p);           // p won't be used anymore
p = q;
```

Calling `free` releases the block of memory that p points to.

17.5 Linked Lists

Are the collection of memory allocated nodes, that point one to another, and end with the one containing the null pointer. Can grow and shrink if needed, but cannot be random accessed, meaning that we would need to go over every one and access them that way. For creating, editing, processing etc. on linked lists, read the **Linked Lists** section on pages 424 - 438.

The `->` Operator

Right arrow selection is useful for accessing a member of a structure using a pointer. We can write:
texttnew_node->value = 10;
instead of:
`(*new_node).value = 10;`

17.6 Pointers to Functions

Functions occupy memory locations.

```
double integrate(double (*f)(double), double a, double b);
\* we could also declare f as though it were a function *\
double integrate(double f(double), double a, double b);

result = integrate(sin, 0.0, PI / 2);
```

`sin` is without parentheses, because then compiler produces a pointer to the function instead of generating code for a function call.

17.7 Restricted Pointers (C99)

`int restrict p;` The keyword `restrict` means that p cannot be accessed in any way other than through p.

17.8 Flexible Array Members (C99)

When the last member of a structure is an array, its length may be ommited:

```
struct vstring {
    int len;
    char chars[];   /* flexible array member - C99 only */
}

struct vstring *str = malloc(sizeof(struct vstring) + n);
str->len = n;
```

The length of the `chars` array isn't determined until memory is allocated for a `vstring` structure. The `sizeof` operator ignores the `chars` member when computing the size of the structure. It takes up no space within a structure.