C Programming For Beginners
Master the C Language
Notes

David Corzo

2020 May 18

Contents

Chapter 1

Introduction

1.1 Fundamentals of a program

- Computers are told instructions in the form of algorithms, when a computer program is executed the CPU computes the instructions of the algorithm.

1.1.1 Terminology

- CPU: central processing unit, instructions are executed here.
- RAM: random access memory, stores data while it is running; RAM != hard drive.
- Hard drive: permanent storage, stores files, program source code, even while the computer is turned off.
- Operating system: program that controls all the operations of the computer, it knows how to interact with the hardware, manages computer resources, executes programs. Windows, Unix, android are all OS.
- fetch / execute cycle (life of a CPU): fetches an instruction from memory (using registers) and executes, this is a loop. Gigahertz are the unit of measurement of which this process referenced, means CPU will fetch and execute one billion times a second.

1.1.2 Higher level programming language

- A high level language is the opposite of an assembly language.
- C is a higher level language.
- A compiler translates higher level code in to machine code. Compilers also checks for syntax rules.

1.1.3 Writing a program

↓ 1. Define a program objectives: understand the requirements.

↓ 2. Design the program: how are you going to do it.

↓ 3. Write the code: using the programming language write it.

↓ 4. Compile

↓ 5. Run the program: executed

↓ 6. Test and debug the program: just because it's running it doesn't mean it works.

↓ 7. Maintain and modify the program: sometimes the most expensive step.

- The larger the program the larger the planning.
- Divide and conquer: small steps and constantly test you code.

1.2 Overview

- C is a general purpose imperative computer programming languages that supports structures programming.
- Uses statements that change a program's state.
- C is the preferred language for producing word processing, spreadsheet programs and compilers; microprocessors, DVD's, cameras all use C.
- C++ is a subset of C, if you learn C you also learn C++.
- Provides low-level access to memory, has low level capacities.
- Was invented by Dennis Ritchie of Bell Laboratories, he was working on the design of the UNIX operating system.
- Main goal is to be a useful language, not having to write 10 lines to add two numbers.
- C evolved from a previous programming language named B, in B all data items occupied one "word" in memory, it was a "typeless" language.
- C is available for most computers. C is also hardware independent.
- 1970 evolution of C to "traditional C", versions of C where: C90,C89,C99 standardization is important. The current standard is C11.
  - C89: most C code is based on C89.
  - C99: refines and expands the capabilities of C.

1.3 Language features

- Efficiency and portability: C is efficient because of it's compact and fast customizable.
- C is the leader on portability.
- C was used to create FORTRAN, Perl, Python, Pascal, LISP, Logo and Basic.
- C programs have been used for solving physics and engineering problems and even for animating effects for movies.
- C is flexible: flexibility can cause more bugs and others.
- C fulfills the needs of programmers:
  - Gives you access to hardware.
  - Enables you to manipulate individual bits in memory.
- C contains a large selection of operators.

- C gives you more freedom, but it also puts more responsibility on you.
- C implementations have a large library of useful C functions.
- Programs can be manipulated at a bit level.

Disadvantages:

- Flexibility and freedom also added responsibility.

1.4 Creating a C program

- Editing:
  - writing the code.
- Compiling:
  - converts your source code into machine language.
  - Two phases: preprocessing phase and the compilation stage.
  - Compiler examines each program statement and checks for syntax and semantic errors.
  - The compiler will take each statement of the program and translate it into assembly language, then translates the assembly language statements in to machine code.
  - The output from the compiler is known as object code and it is stored with file extensions .o or .obj called object files.
  - Compiling: `gcc -c myprog.c` if you omit the `-c` flag the program will automatically link as well.
- Linking: is just getting all the dependencies in place so that one program is turned into one executable.
  - A linker combines the object modules generated by the compiler with the additional libraries needed by the program to create the whole executable.
  - Linking errors occur when in this linking phase a .o dependency is not found.
- Executing:
  - Results of the program are displayed on a window called console.

Chapter 2

Starting to write code

2.1 Create a new project in Code::Blocks

↓ File

↓ New

↓ Project

2.2 Set up a project in vscode

↓ Install C/C++ extension from Microsoft.

↓ Search in command palette: `C/C++: Edit configurations (UI)`

- Search for the compiler path and set it for: `C:/cygwin64/bin/gcc.exe`
- Set intelligence mode to: gcc-x64
- This will create a .vscode folder containing a file called `c_cpp_properties.json`
- Open the `c_cpp_properties.json`:

```json
{
    "configurations": [
        {
            "name": "Win32",
            "includePath": [
                "${workspaceFolder}/**"
            ],
            "defines": [
                "_DEBUG",
                "UNICODE",
                "_UNICODE"
            ],
            "compilerPath": "C:\\cygwin64\\bin\\gcc.exe",
            "cStandard": "c11",
            "cppStandard": "gnu++14",
            "intelliSenseMode": "gcc-x64"
        }
    ],
    "version": 4
}
```

↓ Search in command palette: `Tasks: Configure Default Build Task`

- Create a new `tasks.json` file from template.
- Then select the `others` template.
- Enter the following arguments:
  - “label”: “build ⟨name of project⟩ ”
  - “type”: “shell”
  - “command”: “gcc”
  - “args”:
    - -g means global
    - -o means to modify the name of the file to.
    - write the source file with and without extension: “ ⟨name⟩ ”, “ ⟨name.c⟩ ”

```
{
    // See https://go.microsoft.com/fwlink/?LinkId=733558
    // for the documentation about the tasks.json format
    "version": "2.0.0",
    "tasks": [
        {
            "label": "build",
            "type": "shell",
            "command": [
                "C:\\cygwin64\\bin\\gcc.exe"
                // "&& helloworld"
            ],
            "args": [
                "-g", // global
                "helloworld.c",
                "-o",
                "helloworld",
                // "&&",
                // "helloworld_compiled.exe"
            ],
            "group": {
                "kind": "build",
                "isDefault": true
            }
        }
    ]
}
```

↓ Search in command palette: `Debug: Open launch.json` then select the `C++(GDB/LLDB)` environment.

- Change the workspace folder in “program”
- set stopAtEntry to true
- set miDebuggerPath to `C:/cygwin64/bin/gdb.exe`

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "name": "(gdb) Launch",
            "type": "cppdbg",
            "request": "launch",
            "program": "${workspaceFolder}/helloworld.exe",
            "args": [],
            "stopAtEntry": true,
            "cwd": "${workspaceFolder}",
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb",
            "miDebuggerPath": "C:/cygwin64/bin/gdb.exe",
            "setupCommands": [
                {
                    "description": "Habilitar la impresin con sangra para gdb",
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                }
            ]
        }
    ]
}
```

↓ Search in command palette: `Preferences: Open Settings (JSON)`

  – set file.associations to "*.c":"c"
  – set "terminal.integrated.shell.windows":"`C:\\cygwin64\\bin\\bash.exe`" to specify a default terminal.

```
{
    "terminal.integrated.shell.windows": "C:\\WINDOWS\\system32\\cmd.exe",
    "files.associations": {
        "*.c": "c"
    },
    "window.zoomLevel": 0,
    "editor.fontSize": 20
}
```

↓ Exit the IDE and reenter so that the changes specified go in to effect.

2.2.1 Use the settings and shortcuts

- F5 to display the debugger.

2.3 Structure of a c programs

- main() is a function, it's the entry point of a program, there should be at least one main in all c programs and only one main function.
- C is a case-sensitive language.
- C isn't tab sensitive like Python, you can actually not indent code and it will compile, however it's recommended to do so.
- Readability is very important.

Chapter 3

Basic concepts

3.1 Comments

- Useful for documenting a program.
- Useful for reminging.
- Comments are ignored by the compiler.

3.1.1 Syntax

- Multiline comments: /*Comment*/
- Single line comments: // Comment

3.1.2 Use of comments

- Use comments in moderation.
- Self documenting comments by using meaningful names for variables and other elements of the code.

3.2 The proprocessor

- It's part of the compiling phase.
- This is done before compiling any sourcode.

3.2.1 Syntax

- Preprocessor statements are preceded by a # sign. It must not include a space, it's the first non-space character on the line.
- The #include is a preprocessor directive.

3.2.2 Defining your own preprocessors

- For creating constants: #define
- For building your own library files: #include
- Make more powerful programs with the conditional #ifdef, #endif, #else and #ifndef

3.3 The #include statement

- It is a preprocessor directive.
- The program won't work without it.
- Tells the compiler to include the contents of the header file stdio.h.

3.3.1 Header files

- Header files define information about some functions that are provided by that file.
- stdio.h is standard library header and provides functionality for displaying output.
- It for example contains the information for the compiler to know what printf means.
- stdio stands for standard input / output.
- Header files are case-sensitive.
- Two ways to include header files:
  - The `<stdio.h>` syntax is usually used for built-in functions.
  - The `"something.h"` syntax is usually for user defined header files.
- The `#define` and `#ifndef` protect against multiple inclusions of the header file for efficiency.
- The header files can have: `#define` directives, structure declarations, typedef statements, function prototypes, constants, etc.
- Executable code goes in a .c file not in a header file.
- Header files are needed to create linking during compilation.

3.4 Displaying output

- `printf()` is a standard library function.
- Used to display program results, print values of variables, results of computations, used for debugging.

3.5 Reading input from the terminal

- Used to ask a user for data.
- `scanf()` is used for console reading, also to read data from files.
- stdin standard input stream.
- You must have a format specifier.

3.5.1 scanf()

- String followed by list of arguments.
- `scanf()` function uses pointers.

Three rules for scanf:

1. returns the number of items that it successfully reads.
2. If you use `scanf()` to read a value for one of the basic variable types we've discussed, precede the variable name with a & this is a pointer.
3. If you use `scanf()` to read a string into a character array, don't use an & this is a pointer.

scanf takes two arguments:

1. The string with format specifiers.
2. The & to the variable.

3.5.2 Peculiarities of the scanf

- The scanf() function uses white space (newlines, tabs and spaces) to decide how to divide the input into separate fields.
- scanf() is the inverse of printf(), converts integers, floating-points, characters and C strings to text that is to be displayed onscreen
- When scanf() is executed the program is paused and you will be required to enter data.
- scanf() expects input in the same format as you provided `%s` and `%d`, you have to proide valid input like string then integer for `%s` `%d`.
- scanf() is problematic when you call it again and again do to the fact that it only reads up until a space and not a return, thus it gets messed up.

3.5.3 Example

```c
#include <stdio.h>
int main()
{
    char str[100];
    int i; double x;
    printf("Enter a value: ");
    // read a integer first (space) a string
    scanf("%d %s %lf", &i, str, &x);
    printf("\nYou entered: %d %s %lf\n", i, str, x);
    return 0;
}
/* Output:
Enter a value: 25 string 45.268

You entered: 25 string 45.268000

*/
```

Chapter 4

Variables and data types

4.1 Overview

- The concept of memory is directly related to variables and data types.
- Each byte of a computer has been labeled with a number called an address.
- The address of a byte is unique.
- This all happens in RAM, which can be thought of as an ordered sequence of boxes with binary placeholders.
- Memory thus consists of a large number of bits that are arranged in to groups of eight called bytes and each byte has a unique address.

4.2 Variables

- Programs are only useful if they can manipulate data.
- Constants are the opposite of a variable.
- Variables are types of data may change or assigned values as the program runs.
- Variables are the names you give to computer memory which are used to store in a computer program, more practical than memorizing all the hexadecimal addresses.
-

4.2.1 Naming rules

Cannot be:

- All variable names must begin with a letter or an underscore.
- It can be afterward followed by any combination of letters (upper and lower), underscores or digits.

Examples of invalid variable names:

- `temp$value`: dollar sign isn't valid.
- `my flag`: embedded spaces aren't permitted.
- `3Jason`: variables can't start with a digit.
- `int`: is a reserved keyword.

Remember:

- Have self documenting variable names.

4.3 Data types

- A way for the computer to identify and use data types such as integers, strings, chars, etc.
- Different data types occupy different memory sizes.
- In C everything is a primitive data type, there are no objects.
- Primitive data types are types that are not objects and store all sorts of data.

4.3.1 Declaring variables

- Syntax: ⟨data type⟩ ⟨variable name⟩
- C requires all variables to be declared before use.

4.3.2 Initializing variables

- To initialize a variable means to assign a starting, or initial value.
- This can be done during the declaration such as `int x = 10;` or after the declaration `int x; x = 10;`.
- Use the `=` operator to assign values.

4.4 Examples

```c
#include <stdio.h>

int main()
{
    int num; // declared
    num = 10; // initialize
    int num_ = 11; // declared and initialized
    return 0;
}
```

4.5 Basic data types

Data types suported:

- `int`
- `float`
- `double`
- `char`
- `_Bool`

What the type means:

- How many memory needs to be allocated for each variable.
- This is system dependent or machine dependent, meaning depending on your machine, the memory allocated is determined.
- An int can take 32 bits or 64 bits depending on you machine.

4.5.1 `int`

- Can contain integer values.
- These values can be negative.
- If the int integer is preceded by a 0x (lower or upper), the value is a hexadecimal one. Ex: `int rgbcolor = 0xFFEF0D;`
- No embedded spaces are permitted between the digits.
- Values greater than 3 digits can't be separated using commas.

4.5.2 `float`

- Contains: numbers with decimal places.
- Floating points can be also expressed in scientific notation. Ex: 1.7e4 is $1.7e^4$.

4.5.3 `double`

- The same as a float, the difference is that the memory allocated is double a float, this means you can store higher numbers.
- All floating-point constants are taken as double values by the C compiler by default.
- To explicitly express a float, append either an f of F to the end of the number. Ex: 12.5f

4.5.4 `_Bool`

- Data type for binary values, true or false.
- Yes or no.
- 0 is a false value, 1 is a true value.

4.5.5 Other data types

C can represent other data types:

- You can have more specified adjectives to describe your data type more specifically.
- Three adjective keywords (called data specifiers), you can also just put short, long of unsigned: syntax is: ⟨adjective⟩ ⟨datatype⟩
  1. `short`: less memory as an int.
  2. `long`: more memory as an int. Append an l of L at the end of the number.
  3. `unsigned`: is used for variables that have only non-negative values, this is just like having an absolute value for your data type, the accuracy of the integer value is extended.
  4. *extra: signed: means they can be negative and positive.

4.5.6 Examples

```c
#include <stdio.h>
#include <stdbool.h>
int main()
{
    float num = 23.333;
    double num2 = 55.555555555555;
    bool boolVar = true;
    printf("%lf,%lf,%d",num,num2,boolVar);
    long int numberOfPoints = 131071100L;
    return 0;
}
/* Output:
23.333000,55.555556,1
*/
```

4.6 Enums and Chars

4.6.1 `enum`

- Data type that allows a programmer to define a variable and specify the valid values that could be stored into that variable.
- You create an enum so that only includes values you predefine.
- Syntax: enum ⟨name⟩ { ⟨valid values⟩ }
- Example: `enum primaryColor { red,yellow,blue }`
- No other values other than the predefined values.
- The compiler treats enum values as integer values.
- Enums can still be assigned index numbers using the `=` operator.

4.6.2 `char`

- chars represent **single** characters.
- chars literals use single quotes such as 'a'.
- You can declare an unsigned char because of the ASCII table.
- You can used numerical codes and it will take it as an ASCII table value and convert it.

4.6.3 Escape character

- Characters that represents an action.
- Also called escape sequences.
- Ex: `char x = '\n'`

Escape sequences table:

Sequence	Meaning
`"\a"`	Alert (ANSI C)
`"\b"`	Backspace
`"\f"`	Form feed
`"\n"`	Newline
`"\r"`	Carriage return
`"\t"`	Horizontal tab
`"\v"`	Vertical tap
`"\\"`	Backslash (\)
`"\'"`	Single quote
`"\""`	Double quote
`"\?"`	Question mark (?)
`"\0oo"`	Octal value
`"\xhh"`	Hexadecimal value

Taken from "C Primer Plus", Prata.

4.6.4 Code

```c
#include <stdio.h>
int main()
{
    enum month {
        January,February,March,April,May,June,July,August,Spetember,October
    };
    enum month thisMonth;
    thisMonth = February;
    printf("%d ",thisMonth);
    enum direction {up,down,left=10,right};
    enum direction dir = left;
    printf("%d ",dir);

    char myChar = '\n';
    printf("%cnewline?",myChar);

    return 0;
}
/* Output:
1 10
newline?
*/
```

4.7 Format specifiers

- These are used to display the value of variables using printf().
- Also used to convert data.
- You can format data precision using a format specifier. Syntax: `%.<precision>f`

4.7.1 Table

Type	printf chars
`char`	`"%c"`
`_Bool`	`"%i"`, `"%u"`
`short int`	`"%hi"`, `"%hx"`, `"%ho"`
`unsigned short int`	`"%hu"`, `"%hx"`, `"%ho"`
`int`	`"%i"`, `"%x"`, `"%o"`
`unsigned int`	`"%u"`, `"%x"`, `"%o"`
`long int`	`"%li"`, `"%lx"`, `"%lo"`
`unsigned long int`	`"%lu"`, `"%lx"`, `"%lo"`
`long long int`	`"%lli","%llx","%llo"`
`unsigned long long int`	`"%llu","%llx","%llo"`
`float`	`"%f"`, `"%e"`, `"%g"`, `"%a"`
`double`	`"%f"`, `"%e"`, `"%g"`, `"%a"`
`long double`	`"%Lf"`, `"$Le"`, `"%Lg"`

Taken from “Programming in C”, Kochan.

4.7.2 Examples

```c
#include <stdio.h>
#include <stdbool.h>
int main()
{
    int integervar = 100;
    float floatingpoint = 331.994523934;
    double doublevar = 8.44e+11;
    char charvar = 'W';
    bool boolvar = 0;

    printf("integervar = %i\n",integervar);
    printf("floatingpoint = %f\n",floatingpoint);
    /*For precision use: %.2f*/
    printf("doublevar = %.2f\n",floatingpoint);
    printf("doublevar = %e\n",doublevar);
    printf("doublevar = %g\n",doublevar);
    printf("charvar = %c\n",charvar);
    printf("boolvar = %i\n",boolvar);
    return 0;
}
/* Output:
integervar = 100
floatingpoint = 331.994537
doublevar = 331.99
doublevar = 8.440000e+11
doublevar = 8.44e+11
charvar = W
boolvar = 0

*/
```

4.8 Command line arguments

- Sometimes data is required via the command line.
- In order to send command line arguments add the main() function parameters, and compile. After compiling type in the command line: ⟨executable name⟩ ⟨command line args⟩

```c
#include <stdio.h>
#include <stdbool.h>
int main(int argc, char *argv[]) // main parameters
{
    int numberOfArguments = argc;
    char *argument1 = argv[0];
    char *argument2 = argv[1];
    printf("Number of arguments: %d\n",numberOfArguments);
    printf("Argument 1 is the program name: %s\n", argument1);
    printf("Argument 2 is the program name: %s\n", argument2);

    return 0;
}
/* Output:
DirectoryToExecutable> command_line_args.exe David
Number of arguments: 2
Argument 1 is the program name: command_line_args
Argument 2 is the program name: David

*/
```

4.9 Challenge 1

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) // main parameters
{
    double height = atoi(argv[1]),width = atoi(argv[2]);
    printf("Enter the height and width: ");
    double perimeter = 2 * (height + width);
    printf("%.3f",perimeter);
    return 0;
}
/* Output:
C:/Directory>helloworld.exe 10 10
Enter the height and width: 40.000
*/
```

```c
#include <stdio.h>
int main(int argc, char const *argv[])
{
    enum company { GOOGLE,FACEBOOK,XEROX,YAHOO,EBAY,MICROSOFT };
    enum company x = XEROX;
    enum company g = GOOGLE;
    enum company e = EBAY;
    printf("%d %d %d",x,g,e);
    return 0;
}
/* Output:
2 0 4
*/
```

Chapter 5

Operators

5.1 Overview

- Operators are functions that use symbolic name, used for mathematical and logical functions.
- Operators are predefined in C,

5.1.1 Operators

- Logical operators.
- Arithmetic operators.
- Relational.
- Bitwise.

5.1.2 Expressions and statements

- They are different.
- Statements: form the basic program in C, most statements are constructed from expressions.
  - They are the building blocks of a programs. Simple statements are short oneliners that end in a ;
  - A complete instruction to the computer.
  - Examples of simple statements:
    - declaration statement: `int a;`
    - Assignment statement: `a = 5;`
    - Function call statement: `funct();`
    - Structure statement: `while (condition) var = num;`
    - return statement: `return 0;`
  - C considers any expression to be a statement if you append a semicolon (expression statements).
- Expressions: consists of a combination of operators and operands: example: -6, 4+21
  - Operands are what an operator operates on.
  - Operands can be constants, variables, or combinations of the two.
  - Every expression has value.
  - Expressions generally return values.

5.1.3 Compond statements

- Two or more statements grouped together by enclosing them in braces (block).
- Example: `int i = 0; while (i < 10) {printf("hello");i++;}` the while statement is enclosing the printf() function statement, they are compound statements.

5.2 Basic operators

- Arithmetic: performs mathematical operations.
- Logical operator: boolean operator.
- Assignment operators: set variables equal to values.
- Relational operator: compares variables against each other.

Operator	Type
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`%`	Modulus
`++`	Increment by one.
`--`	Decrement by one.

Taken from tutorial point website.

Example

```c
#include <stdio.h>
int main(int argc, char const *argv[])
{
    int a = 33, b = 15, result = 0;
    result = a + b;
    printf("Result +: %d\n",result);
    result = a - b;
    printf("Result -: %d\n",result);
    result = a * b;
    printf("Result *: %d\n",result);
    result = a / b;
    printf("Result /: %d\n",result);
    result = a % b;
    printf("Result %%: %d\n",result);
    result = a++ + b;
    printf("Result ++: %d\n",result);
    result = a + b--;
    printf("Result --: %d\n",result);
    return 0;
}
/* Output:
Result +: 48
Result -: 18
Result *: 495
Result /: 2
Result %: 3
Result ++: 48
Result --: 49

*/
```

5.2.1 Logical operators

Operator	Type
&&	and
|	or operator
!	not operator

Example

```c
#include <stdio.h>
int main()
{
    _Bool a = 1, b = 1, result;
    result = a && b;
    printf("a && b: %d\n",result);
    result = a || b;
    printf("a || b: %d\n",result);
    result = !a;
    printf("!a: %d\n",result);
    return 0;
}
/* Output:
a && b: 1
a || b: 1
!a: 0

*/
```

5.2.2 Assignment operators

Operator	Type
=	assignment operator
+=	add and assignment operator
-=	subtract and assign operator
*=	multiply and assign operator
/=	divide and assign operator
%=	modulus and assign operator

5.2.3 Relational operators

Operator	Type
==	equality operator
!=	inequality operator
>	greater than
<	less than
>=	greater or equal than
<=	less than or equal than

5.3 Bitwise operators

- C offers bitwise logical operators and shift operators.
- They operate of bits, inside the integers.
- Useful for: turning on and off bits inside data types.
- Example: using just one int variable store binary data such as (using 0 bit, or 8 bits): makes you more memory efficient.
  - First bit store weather a person is female or male.
  - Second bit if they can speak English.
  - Third bit to represent if they can speak German.
  - Fourth bit ... eight bit.

5.3.1 Binary numbers

- A binary number is a number that includes only ones or zeros.
- Each position of a binary number has value.

Bitwise:

`&`	Binary AND operator
`|`	Binary OR operator
`^`	Binary XOR operator
`~`	Binary ones component operator
`<<`	Binary Left Shift operator
`>>`	Binary Right Shift operator

Truth table:

p	q	$p\&q$	$p\|q$	$p\hat{}q$
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

5.3.2 Examples

```c
#include <stdio.h>
int main()
{
    unsigned int a = 60; // 00111100
    unsigned int b = 13; // 00001101
    printf("a: %d, b: %d\n",a,b);
    int c = 0;
    c = a & b;
    printf("c and: %d\n",c);
    c = a | b;
    printf("c or: %d\n",c);
    c = a ^ b;
    printf("c xor: %d\n",c);
    c = ~a;
    printf("~a: %d\n",c);
    c = a << 2;
    printf("a << 2: %d\n",c);
    c = a >> 2;
    printf("a >> 2: %d\n",c);
    return 0;
}
```

5.4 The cast and sizeof operators

- These look like functions, but they are not, they are actually operators.

5.4.1 Type conversions

- You can convert data automatically, known as implicit conversion.
- Converted data can be truncated or promoted, truncated is become less precise, promoted is become more precise.
- Example: a float is converted to an int, decimal portion will be truncated. `int x = 0; float f = 12.125; x = f` x will only keep the integer part. This is an example of implicit conversion.
- By contrast if you go from int to float, nothing will happen, this will be promoted.
- Two ints are doing division, the result is going to be truncated to an int if it is a decimal.
- Explicit conversions are done with the casting operator, you want to do this to accurately transcribe the data to the desired type.
- Type cast operator has higher precedence than all the arithmetic operators except unary minus and unary plus.
- Example: `(int)21.51 + (int)26.99` is going to be `21 + 26`

5.4.2 sizeof

- Tells you how many bytes are occupied in memory.
- sizeof() is an operator not a function.
- sizeof() is evaluated in compile time, not in runtime, unless a variable-length array is used in its argument.
- Arguments of sizeof can be: basic data types, array names, variables, name derived from a data type, or an expression (such as `x+y`).
- Use the sizeof as much as you can, sizeof(int) can be 4 in one system or 8 in another, you want programs that aren't system dependent.

Other operators:

- `*` is used for pointers.
- `?:` is an operator used for comparisons:
  - if condition is true ? then value x : otherwise value y.
  - name is the ternary operator.

5.5 Operator precedence

- Determines how an expression is evaluated.
- Determines the order of evaluation in the expression.
- If parentheses are provided to prioritize an expression to evaluate.
- Associativity: rules that determine what to do in a situation such as: two expressions have the same precedence, associativity is used to apply additional rules in order to determine which goes first.
- Associativity usually favors what comes first. `1 == 2 != 3`: what do you evaluate first?, the prior expression is equal to `((1 == 2) != 3)`, this is implicit, provide the parentheses in order to prioritize an expression.

Operator precedence (highest to lowest):

Category	Operator	Associativity
Postfix	`()[]->.++--`	left to right
Unary	`+-!~++--(type)*&sizeof`	right to left
Multiplicative	`*/%`	left to right
Additive	`+-`	left to right
Shift	`<<>>`	left to right
Relational	`<<=>>=`	left to right
Equality	`== !=`	left to right

Bitwise precedence (highest to lowest):

Category	Operator	Associativity
Bitwise AND	`&`	left to right
Bitwise XOR	`^`	left to right
Bitwise OR	`|`	left to right
Logical AND	`&&`	left to right
Logical OR	`||`	left to right
Conditional	`?:`	right to left
Assignment	`=+=-=*=/=%/=>>=<<=&=^=|=`	right to left
Comma	`,`	left to right

5.6 Challenge years to days provided minutes

```c
#include <stdio.h>
int main()
{
    long int minutes; double years, minutes_in_a_year = 525600, days;
    printf("Enter the value to compute: ");
    scanf("%d",&minutes);
    years = (double)minutes / minutes_in_a_year ;
    days = ((double)minutes / 60) / 24;
    printf("minutes: %d, minutes in a year: %lf, minutes in days: %lf",minutes,years,days);
    return 0;
}
/* Output:
Enter the value to compute: 500000000
minutes: 500000000, minutes in a year: 951.293760, minutes in days: 347222.222222
*/
```

5.7 Challenge on printing the bytesize of data

```c
#include <stdio.h>
int main()
{
    printf("int: %zd\n",sizeof(int));
    printf("char: %zd\n",sizeof(char));
    printf("long: %zd\n",sizeof(long));
    printf("short: %zd\n",sizeof(short));
    printf("long long: %zd\n",sizeof(long long));
    printf("double: %zd\n",sizeof(double));
    printf("long double: %zd\n",sizeof(long double));
    return 0;
}
/* Output:
int: 4
char: 1
long: 8
short: 2
long long: 8
double: 8
long double: 16

*/
```

Chapter 6

Control flow

6.1 Overview

- Statements are executed in the order they appear.
- Control flow statements will break this order based on decision-making.
- Code can be repeated; branching statements can execute only certain code, these are used inside loops; decision-making.

6.1.1 Decision-making

- Asks a question inside, the answer will respond.
- If the condition is true a statement is executed, if not then the code isn't executed.

If statements:

Statement	Description
if statement	If statement consists of a boolean expression, followed by on or more statements.
if...else statement	if statements can be followed by else statements which executes when the boolean expression of the if statement is false
nested if statements	multiple conditions are evaluated, if statement inside a statement.

6.1.2 Repeating code

- Loop statements allows us to repeat code.
- When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Loops

Loop type	Description
while	repeats until condition
for	repeats a number of times
do...while	while loop but with the exception that it tests the condition at the end of the loop body
nested loops	using one inside another

6.2 If statements

6.2.1 if else

- Syntax is: `if (expression) { /*program statement */};`
- For compound statements you need to surround the program statement with curly braces, if only one statement is used it's okay to use it without the curly braces.

If with an else:

- An addition to the if block, you can add an else clause.
- Syntax is: `if (expression) statement; else statement;`

Example

```c
#include <stdio.h>
int main()
{
    int number_to_test, remainder;
    printf("Enter your number to be tested: "); scanf("%i",&number_to_test);
    remainder = number_to_test % 2;
    if (remainder == 0) printf("Even");
    else printf("odd");
    return 0;
}
/* Output:
Enter your number to be tested: 45
odd
*/
```

6.2.2 else if

- To ask multiple questions.
- Syntax is:

```c
if (expression) /* program statement */;
else if (expression) /* program statement */;
else /* program statement */;
```

Example

```c
#include <stdio.h>
int main()
{
    int number, sign;
    printf("Type in a number: ");scanf("%i",&number);
    if (number < 0) sign = -1;
    else if (number == 0) sign = 0;
    else sign = 1;
    printf("Sign %i\n",sign);
    return 0;
}
```

```
/* Output:
Type in a number: 45
Sign 1

*/
```

6.2.3 nested if-else statement

- If or if else statement inside another.
- Syntax is:

```
if (expression) {
    if (expression) {
        statements;
    }
}
```

6.2.4 The conditional operator (ternary operator)

- This is equivalent to an if else statement.
- Syntax is: ⟨`condition`⟩ ? `expression1: expression2;`—
- First operand is placed before the ?, the second after the ? and before the : and the third one after the :
- This is a short hand notation for if else statements, there are no compound statements here, only simple.
- Example:

```
x = y > 7 ? 25 : 50;
```

result in x being set to 25 if y is greater than 7 or to 50 otherwise.

Example

```
#include <stdio.h>
int main()
{
    // One liner
    int y = 1;
    int x_with_ternary = y > 7 ? 25 : 50;
    // Traditional notation
    int x_without_ternary;
    if (y > 7) x_without_ternary = 25;
    else x_without_ternary = 50;
    printf("%d == %d",x_with_ternary,x_without_ternary);
    return 0;
}
/* Output:
50 == 50
*/
```

6.3 Switch statement

- Serve the same purpose as an if, the difference is that is more organized and made for multiple alternatives rather than using else if multiple times.
- else ifs can be prone to errors.
- Switch are more efficient than else if.
- Syntax is:

```
switch (expression) {
    case val_1:
        program statement; break;
    case val_2:
        program statement; break;
    case val_n:
        program statement; break;
    default:
        program statement; break;
}
```

- The expression in the switch argument is the thing to check for. Cases are like if (val). Default means else.
- Cases must be constants or constant expression.
- When more than one statement is included they don't have to have curly braces surrounding the statement.
- The break statement indicates the termination of a particular case and terminates the switch statement. It jumps out. If the expression is not mutually exclusive (more than one case will be true), don't put the break statements, otherwise put them, keep in mind that there must be one case that terminates the switch statement. This can cause bugs.
- If none of the cases are true the default block is executed.

6.3.1 Example

```
#include <stdio.h>
int main()
{
    enum Weekday {Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday};
    enum Weekday today = Monday;
    switch (today)
    {
    case Sunday:
        printf("Sunday");break;
    case Monday:
        printf("Monday");break;
    case Tuesday:
        printf("Tuesday");break;

    default:
        printf("Whatever");break;
    }
    return 0;
}
/* Output:
Monday
*/
```

```
#include <stdio.h>
int main()
{
    float value1, value2;
    char operator;
    printf("Type in your expression.\n");
    scanf("%f %c %f",&value1,&operator,&value2);
    switch (operator) {
        case '+': printf("%.2f\n",value1 + value2); break;
        case '-': printf("%.2f\n",value1 - value2); break;
        case '*': printf("%.2f\n",value1 * value2); break;
        case '/': value2 == 0 ? printf("division by zero error"):
        ↪  printf("%.2f\n",value1 / value2); break;
        default: printf("unknown operator\n"); break;
    }
    return 0;
}
/* Output:
Type in your expression.
45+5
50.00

*/
```

6.3.2 goto statement

- Used for jumping to a specific line of code.
- Has two parts, the goto and the label name.
- Label is named following the same convention used in naming a variable.
- Example: goto part2; part2 is a label and it has to be labeled for the compiler to know what line you mean.
- You can jump all around in your code.

Example

```
#include <stdio.h>
int main()
{
    int sum = 0;
    for (int i = 0; i <= 10; i++)
    {
        sum += i;
        if (i == 5) {
            goto addition;
        }
```

```c
    }
    addition: // label
    printf("%d",sum);
    return 0;
}
```

6.4 Challenge calculate week pay

```c
#define PAYRATE 12.00
#define TAXRATE_300 .15
#define TAXRATE_150 .20
#define TAXRATE_REST .25
#define OVERTIME 40
/*Overtime: exces of 40 hours = time and a half. */
#include <stdio.h>
int main()
{
    double hours = 0, grossPay = 0, taxes = 0, netPay = 0, overTimePay = 0;
    printf("Please enter the number of hours worked this week: ");
    scanf("%lf",&hours);
    if (hours <= 40) {
        grossPay = hours * PAYRATE;
    } else {
        grossPay = 40 * PAYRATE;
        overTimePay = (hours - 40) * (PAYRATE * 1.5);
        grossPay += overTimePay;
    }

    if (grossPay <= 300) {
        taxes = grossPay * TAXRATE_300;
    } else if (grossPay > 300 && grossPay <= 450) {
        taxes = 300 * TAXRATE_300;
        taxes += (grossPay - 300) * TAXRATE_150;
    } else if (grossPay > 450) {
        taxes = 300 * TAXRATE_300;
        taxes += 150 * TAXRATE_150;
        taxes += (grossPay - 300 - 150) * TAXRATE_REST;
    }

    netPay = grossPay - taxes;
    printf("Gross pay: %.2f\n",grossPay);
    printf("Taxes: %.2f\n",taxes);
    printf("Net pay: %.2f\n",netPay);
    printf("Overtime: %.2f\n",overTimePay);
    return 0;
}
/* Output:
Please enter the number of hours worked this week: 30
Gross pay: 360.00
Taxes: 57.00
Net pay: 303.00
Overtime: 0.00
```

```

*/
```

6.5 For loops

- Allows us to repeat code, this is a counter controlled loop because the number of iterations are predefined.
- Sentinel loops execute an undefined number of times until a certain condition is met.
- For simple statements you can omit the braces.
- Below C99 you must declare the counter variable outside the for loop.
- Syntax is: `for (starting condition; continuation condition; action per iteration) {statements;}`
- Example:

```c
for (int i, j = 2; i <= 5; ++i, j += 2)
    printf("%5d",i*j);
```

- If the action per iteration is not put, the for loop becomes an infinite loop.
- Something like: `for (;;){statements;}`

6.5.1 Example

```c
#include <stdio.h>
int main()
{
    for (;;printf("hello "));
    return 0;
}
```

6.6 While loop, do while

6.6.1 While loops

- Mechanism to execute a set of statements as long as a condition is met.
- Syntax is: `while (expression) {statements;}`
- You don't need to put curly braces when you only have one statement.
- If the loop is true the loop has to have a breaking mechanism so that it doesn't become an infinite loop.
- While loops are called pretest loop, it executes if a condition pre-evaluated results in true.

6.6.2 do-while loops

- In the do while loop, you execute the code at least once, while loops execute code if the condition is true, do-while loops are going to execute the code at least once, never zero, regardless of the condition being true or false.
- After the first iteration it will become a while loop.
- The condition is at the bottom. This is called a post-test loop.
- Syntax is: `do{statement;}while(condition)`
- Example:

```
do {
    // prompt for password;
    // read user input;
} while (/*input not equal to password*/);
```

Example

```c
#include <stdio.h>
int main()
{
    int number = 0;
    do
    {
        printf("\nNumber = %d",number);
        number++;
    } while (number < 4);

    return 0;
}
/* Output:

Number = 0
Number = 1
Number = 2
Number = 3
*/
```

6.6.3 Which loop to use

- If you want to execute it at least once unconditionally, use the do-while.
- if you want to execute it while some condition is true then use a regular while loop.
- Its a matter of taste, what you can do in a for loop you can do in a while loop.
- `for(;test;)` is the same as `while (test)`.

6.6.4 For loop and while loop equivalents

For equivalent in a while loop.

```c
initalize;
while (test){
    body;
    update;
}
```

Same as:

```c
for (initialize; test; update){
    update;
}
```

- For loops are appropriate when the loop involves initializing and updating a variable.
- A while loop is better when the conditions are otherwise.
- Use the while loop for logic controlled loops and the for loop for counter controlled loops.

6.7 Nested loops and loop control - break and continue

Nested loops

- Nested loops are loops inside loops.
- You can have a while loop inside a for loop and vice versa, etc.

Continue statements

- It will skip that iteration, if a condition is met you can skip the iteration.

Example

```c
#include <stdio.h>
int main()
{
    enum Day {Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday};
    for (enum Day day = Monday; day <= Sunday; ++day){
        if (day == Wednesday){
            continue;
        }
        printf("It's not wednesday %d\n",day);
    }
    return 0;
}
/* Output:
It's not wednesday 0
It's not wednesday 1
It's not wednesday 3
It's not wednesday 4
It's not wednesday 5
It's not wednesday 6

*/
```

PS you can iterate through enums.

6.7.1 Break statement

- Breaks are used to jump out of loops, they are a way to stop execution of the current loop.
- If you are breaking out of a nested loop, the break will only affect the innermost loop containing the break.
- Used to break out of the loop if a condition is met.
- Switch statements also use the break keyword, they do the same things.

Example

```
#include <stdio.h>
int main()
{
    int p = 0, q = 0;
    while (p > 0) {
        printf("%d\n",p);
        scanf("%d",&q);
        while (q > 0) {
            printf("%d\n",p*q);
            if (q > 100) break;
            scanf("%d",&q);
        }
        if (q > 100) break;
        scanf("%d",&p);
    }
    return 0;
}
```

6.8 Challenge guess the number

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#define NUM 20
int main()
{
    int guess,num;
    time_t t; srand((unsigned) time(&t));
    num = rand() % NUM; printf("NUM:%d\n",num);
    printf("This is a guessing game.\n");
    printf("Choose a number between 0 and 20 which you must guess.\n");
    for (int i = 0, j = 5; i < 5; i++) {
        do {
            printf("\nEnter a guess: ");
            scanf("%d",&guess);
            if (guess < 0 || guess > 20) printf("\nNumbers must be between 0 and 20.");
        } while (guess < 0 || guess > 20);

        if (guess == num) {
            printf("\nCongrats, you guessed it.");
            break;
```

```c
        } else {
            --j == 0 ? printf("No tries left. \n") : printf("Not correct. %d tries
             ↪  left.\n",j);
        }
    }

    return 0;
}
/* Output:
NUM:5
This is a guessing game.
Choose a number between 0 and 20 which you must guess.

Enter a guess: 12
Not correct. 4 tries left.

Enter a guess: 13
Not correct. 3 tries left.

Enter a guess: 9
Not correct. 2 tries left.

Enter a guess: 6
Not correct. 1 tries left.

Enter a guess: -1000000

Numbers must be between 0 and 20.
Enter a guess: 45

Numbers must be between 0 and 20.
Enter a guess: 4
No tries left.

*/
```

Chapter 7

Arrays

7.1 Creating and using arrays

- Arrays permit storing many values associated to a single variable.
- Allows us to group values together, under the same name. No longer do you need a separate variable for a new value.
- A limitation is that an array has to be declared with the size from the beginning, and all the elements need to be of the same type.
- Declaring an array, data inside an array are called elements.
- Syntax is: `long numbers[10];` an array named numbers with 10 elements.
- To access an array you must write data to one or more of the indexes. Indexes start from 0.
- You can access the array by typing the index number or an equivalent expression that results in a number that corresponds to an index in the array.
- It's common to use a loop to access an array.
- Trying to access an index outside a range of the array you'll get a `array out of bounds error`. An `array out of bounds error` can crash your program, can also not crash your program but make it run with bugs.
- The compiler cannot check for out of bounds errors so your program will still compile. This type of error is a run time error.
- Assigning values to an array, syntax is: `grades[100] = 95;`

Example

```c
#include <stdio.h>
int main()
{
    int grades[10], count = 10; long sum = 0; float average;
    printf("\nEnter the 10 grades: \n");
    for (int i = 0; i <= count; i++)
    {
        printf("%2u> ",i + 1);
        scanf("%d",&grades[i]);
        sum += grades[i];
    }
    printf("Average: %f",(float)(sum/count));
    return 0;
}
/* Output:
// index out of bounds visualized

Enter the 10 grades:
 1> 46
 2> 48
 3> 49
 4> 41
 5> 50
 6> 46
 7> 48
 8> 46
 9> 19
10> 79
11> 46
Average: 51.000000
*/
```

7.2 Creating and using arrays

- Initialization, you can intialize the array in the same line, using curly braces.
- Example: `int counters[5] = {0,0,0,0,0};`
- It's not necessary to initialize the entire array.
- Example: `float sample_data[500] = {100.0,300.0,500.5};` this will initialize only the first three elements, the remaining elements are set to 0.
- You can also initial only certain indices, such as: `float sample_data[500] = {[2]=500.5,[1]=300.0,[0] = 100.0};` not all compilers allow this but some do.
- The array size needs to be a positive number.

7.2.1 Example

```
#include <stdio.h>
#define MONTHS 12
int main()
{
    int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
    for (int index = 0; index < MONTHS; index++) {
        printf("Month %d has %2d days. \n", index + 1, days[index]);
    }
    return 0;
}
/* Output:
Month 1 has 31 days.
Month 2 has 28 days.
Month 3 has 31 days.
Month 4 has 30 days.
Month 5 has 31 days.
Month 6 has 30 days.
Month 7 has 31 days.
Month 8 has 31 days.
Month 9 has 30 days.
Month 10 has 31 days.
Month 11 has 30 days.
Month 12 has 31 days.

*/
```

7.3 Multidimensional arrays

- C allows arrays of any dimension to be defined.
- Two-dimensional array can have a use case for spreadsheet and matrices.
- Syntax is:

```c
int numbers[3][4] = {
    {10,20,30,40},
    {15,25,35,45},
    {47,48,49,50}
}
```

- The inner brackets are optional, do it for readability.
- You can also initialize only a limited number of elements in you array. In the example we initialize the first three elements out of 5 in the nested array. In this case the inner pairs of braces are required.

```c
int matrix[4][5] = {
    {10,5,-3},
    {9,0,0},
    {32,20,1},
    {0,0,8}
};
```

- You can also use designated initializers.

```c
int matrix[4][3] = {[0][0] = 1, [1][1] = 5, [2][2] = 9};
```

- You can declare a three-dimensional array like this: `int box[10][20][30];`
- You can visualize dimensions like this:
  - One-dimension: row of data.
  - Two-dimensional: table of data, matrix, spreadsheet.
  - Three-dimensional: stack of data tables.
- The more dimensions you have, the more nested for loops you need to iterate through them.
- Example of initialization of a three-dimensional array:

```c
int numbers[2][3][4] = {
    {
        {10,20,30,40},
        {15,25,35,45},
        {47,48,49,50}
    },
    {
        {10,20,30,40},
        {15,25,35,45},
        {47,48,49,50}
    }
};
```

- Iteration in an n dimensional array: The number of nested loops corresponds directly with the dimensions of arrays.

```c
int sum = 0;
for (int i = 0, i = 0; i < 2; ++i) {
    for (int j = 0; j < 3; ++j) {
        for (int k = 0; k < 4, ++k) {
            sum += numbers[i][j][k];
        }
    }
}
```

- In C there is no limit to the number of nested loops you can have.

7.4 Variable length arrays

- Arrays are of a fixed length after they are declared, this is a problem.
- Variable length is introduced in C99, variable length just means that instead of hard-coding the array size, you can have a variable that does that for you, the array is still going to have a fixed size but it won't be hard coded, this was useful when they were transcribing FORTRAN algorithms to C.
- Examples:

```c
int n = 5, m = 8;
float a9[m][n]; // this wasn't allowed before C99.
```

- It was introduced for compatibility with FORTRAN libraries.

7.5 Challenge: generate prime numbers from 0 to 100

```c
#include <stdio.h>
#include <stdbool.h>

int main()
{
    int p;
    int i;

    int primes[50] = {0};

    int primeIndex = 2;

    bool isPrime;

    // hardcode prime numbers
    primes[0] = 2;
    primes[1] = 3;

    for(p = 5; p <= 100; p = p + 2) {
        isPrime = true;
        for (i = 1; isPrime && p / primes[i] >= primes[i]; ++i)
            if (p % primes[i] == 0)
                isPrime = false;


        if (isPrime == true)
        {
            primes[primeIndex] = p;
            ++primeIndex;
        }
    }

    for ( i = 0;  i < primeIndex;  ++i )
         printf ("%i  ", primes[i]);

    printf("\n");
    return 0;
}
/* Output:
2  3  5  7  11  13  17  19  23  29  31  37  41  43  47  53  59  61  67  71  73  79  83
↪  89  97
*/
```

```c
#include <stdio.h>
#define MONTHS 12
#define YEARS 5
int main()
{
    float rainfall[YEARS][MONTHS] = {
        {4.3,4.3,4.3,3.0,2.0,1.2,0.2,0.2,0.4,2.4,3.5,6.6},
        {8.5,8.2,1.2,1.6,2.4,0.0,5.2,0.9,0.3,0.9,1.4,7.3},
        {9.1,8.5,6.7,4.3,2.1,0.8,0.2,0.2,1.1,2.3,6.1,8.4},
        {7.2,9.9,8.4,3.3,1.2,0.8,0.4,0.0,0.6,1.7,4.3,6.2},
        {7.6,5.6,3.8,2.8,3.8,0.2,0.0,0.0,0.0,1.3,2.6,5.2}
    };
    float averages[YEARS], current = 0;
    for (int i = 0; i < YEARS; i++){
        current = 0;
        for (int j = 0; j < MONTHS; j++){
            current += rainfall[i][j];
        }
        averages[i] = current / MONTHS;
    }
    for (int i = 0; i < 5; i++){
        printf("Year: %d> %.2f",i,averages[i]);
        printf("\n");
    }
    return 0;
}
/* Output:
Year: 0> 2.70
Year: 1> 3.16
Year: 2> 4.15
Year: 3> 3.67
Year: 4> 2.74

*/
```

Chapter 8

Functions

8.1 Overview

- A function is a self-contained unit of a program code designed to accomplish a particular task.
- Something you can invoke to do something.
- Syntax rules define the structure of a function and how it can be used.
- A function in C is the same as subroutines or procedures in other programming languages.
- Some functions cause an action.
- Functions allow divide and conquer strategy.
- It's very hard to do everything in the main function.
- Functions make it easier to debug and test, also to maintain.
- Functions can separate sub-tasks, you can further subdivide.
- Reduces overall complexity.
- Functions reduce code repetition, you can write a function to a library, and they can be invoked, they can be reused.
- Advantages:
  - Helps with readability.
  - Program is better organized.
  - Easier o read and easier to maintain (maintaining is often the most expensive part in a program).
- Functions make it easier to work in teams, to develop large projects and can be debugged independently. This reduces overall development time.
- Functions, since they can be reused, this means you can use it again and again instead of writing the code again and again.
- View functions like a black box: input and output. Functions are all about input and output.
- It's important to think before writing a program, functions can obligate you to do so.
- You can pass data inside the parentheses, called arguments. Data can be also returned, this is useful when we need a result and assign it to a variable.

8.1.1 Implementing functions

- You can create your own functions.
- Remember to also use self documenting names for functions.
- Functions are implemented → invoked → pass and return data from them.

8.2 Defining functions

- Defining a function is saying that a function exists and does something if the right data is passed to it.
- Functions always use curly braces, you can't skip the curly braces even if the function just uses one statement.
- In between the braces is called the function body, the name, type and parameters are called the function header.
- The data passed to a function can be manipulated in any normal way and that manipulated data can be returned.
- Syntax is: ⟨return type⟩ ⟨name of function⟩ `{ statements; }`
- The header tells the compiler three things, the type it will return, the function name and the arguments it takes in.
- `void` is a function that doesn't return anything.
- The braces of the function body can be empty, however the function must always have curly braces.
- Naming functions follow the same rules as the variables, you can't have two functions with the same name, each must be unique, you can't name a function keyword names such as `sizeof`. Don't name your functions the same as standard library functions.
- Naming functions: `camelCase`, `snake_case`, `Capitalize`
- Function prototypes: a statement that defines the functions, also called function signature, you want to write a function prototype the same way you write the function header. Function prototypes are usually defined in a header file. If they are used by various programs you should place it in a header file.
- You don't have to have the same parameter names when you pass in data in variables.
- Its good practice to always include declarations for all the functions in a program source file regardless of where they are called, this will help your programs be more consistent in design, it will also prevent errors from occurring if, at any stage, you choose to call a function from another part of you program.
- Example of function prototypes:

```c
#include directives
// func prototypes
double average(double data_val[], size_t count);
double sum(double*x, size_t n);
size_t get_data(double*, size_t);

int main(void){
    statements;
}
// define funcs
```

```c
average(){
    statements;
}
```

8.2.1 Example

```c
#include <stdio.h>
// function prototypes
void add();
int main()
{
    add();
    return 0;
}

void add(){}
```

8.3 Arguments and parameters

- A parameter is a variable in a function declaration and function definition / implementation.
- When a function is called, the arguments are the data that you pass into the function parameters. The actual value of a variable that gets past to a function.
- Function parameters are defined within the function header.
- The parameters for a function are a list of parameter names with their types.
- If the function doesn't require any parameters its good practice to put the keyword void inside the parentheses.
- The names of the parameters are local to the function.
- The body of the function should use these parameters in its implementation.
- You may have inside the body other declared and initialized variable.
- When passing an array as an argument to a function, you must:
  - Pass an additional argument specifying the size of the array.
  - The function has no means of knowing how many elements are in the array.
- Parameters greatly increase the usefulness and flexibility of a function. If the printf() wouldn't have parameters it wouldn't be useful.
- It's a good idea to add comments before each of your own function definitions.

8.3.1 Example

```c
#include <stdio.h>
void multiply_two_nums(int x, int y);
int main()
{
    multiply_two_nums(3,4);
    multiply_two_nums(5,6);
    multiply_two_nums(5,5);
    return 0;
}

void multiply_two_nums(int x, int y){
    int result = x * y;
    printf("x: %d, y: %d, result: %d\n",x,y,result);
}
/* Output:
x: 3, y: 4, result: 12
x: 5, y: 6, result: 30
x: 5, y: 5, result: 25

*/
```

8.4 Returning data

- Functions can return data using specific syntax.
- Example:

```c
/*type*/ func(params){
    /* statements */;
    return something;
}
```

- You can specify the type of value to be returned by a function as any of the legal types in C, includes enumeration types and pointers.
- void means that no data will be returned.
- Return statement: if you provide the return function like this `return;` this means that you are just simply exiting the function, you use it in a void function. In the function header you declared the data that will be returned, the function has to return type in accordance with the function header. Use `return something;` when this is the case.
- If you return something that can be converted, the compiler will implicitly convert it, this can cause bugs. If it can't implicitly convert you will get a compiler error.
- You can have multiple returns in one functions.
- Invoking a function: you invoke a function you call it by name, you then pass in the parameters, order matters, you must pass the data in the same way in which it's declared in the function header.
- Parameters can also be expressions that result in a given and accepted type.
- You can assign function returns to variables, like so: `int x = my_func();` use the equals operator.
- If the function returns data and you don't care about that data you don't need to assign it to a variable, you can just use it as a void. The compiler will not complain if you don't use the data returned from function calls.

8.4.1 Example

```c
#include <stdio.h>
int multiply_two_nums(int x, int y);
int main()
{
    int result = multiply_two_nums(3,4);
    printf("%d",result);
    return 0;
}

int multiply_two_nums(int x, int y){
    return x * y;
}
/* Output:
12
*/
```

8.5 Local and global variables

- Variables defined inside a function are known as automatic local variable, you can't access that variable from another one, it's only accessible from where it's defined.
- If an initial value is given to a variable inside a function, that initial value is assigned to the variable each time the function is called.
- You can use a static variable to counter this effect, for example if you were to want to calculate the number of times a function was called you can declare a static variable and access it locally and globally. Just regular local variables are going to have different values each time.
- Use the auto keyword to be more precise, but the compiler usually adds it by default.
- Local variables are also applicable to any code where the variable is created in a block (loops, ifs, etc.), a block is where the brackets start and end.
- If you have an if statement declared inside a block, you are going to be able to access it only inside that block, not outside.
- Global variables: the opposite of local variables, a global variable can be used by all functions, this means putting it outside the main function.
- Global variables are declared outside any function, they don't belong to any particular function.
- This type of variable is alive for the entirety of the program, from when it starts to when it ends.
- Any function in the program can change the value of that global variable.
- If there is a global variable and a local variable with the same name, the local variable will take precedence. Global variable if you have variables with the same name locally.
- Avoid using global variables: global variables cause coupling between functions (dependencies), difficult to debug functions this way, if one function doesn't work and you fix it, you may cause other functions to malfunction.
  - Hard to find the location of a bug in a program.
  - Hard o fix a bug once it's found because other functions can malfunction (dependency issues).

- Use parameters instead of a global variable.
- If you do use global data, document that is global always; some modern programming languages actually abolished the idea of global variables because they are so inconvenient.

8.5.1 Example

```c
#include <stdio.h>
int my_global = 0;
int my_func(void);
int main()
{
    // my_global is accesible from here.
    printf("main: %d\n",my_global);
    printf("my_func: %d\n",my_func());
    return 0;
}
int my_func(void) {
    return my_global;
}
/* Output:
main: 0
my_func: 0

*/
```

8.6 Challenge Write some functions

```c
#include <stdio.h>
#include <stdlib.h>

int gcd(int u, int v);
float absoluteValue(float x);
float squareRoot(float x);

int main()
{
    int result = 0;

    float  f1 = -15.5, f2 = 20.0, f3 = -5.0;
    int    i1 = -716;
    float absoluteValueResult = 0.0;

    result = gcd(150, 35);
    printf("The gcd of 150 and 35 is %d\n", result);

    result = gcd(1026, 405);
    printf("The gcd of 1026 and 405 is %d\n", result);

    printf("The gcd of 83 and 240 is %d\n\n\n\n", gcd(83, 240));

    /* testing absolute Value Function */
    absoluteValueResult = absoluteValue (f1);
    printf ("result = %.2f\n", absoluteValueResult);
    printf ("f1 = %.2f\n", f1);

    absoluteValueResult = absoluteValue (f2) + absoluteValue (f3);
    printf ("result = %.2f\n", absoluteValueResult);

    absoluteValueResult = absoluteValue ( (float) i1 );
    printf ("result = %.2f\n", absoluteValueResult);

    absoluteValueResult = absoluteValue (i1);
    printf ("result = %.2f\n", absoluteValueResult);

    printf ("%.2f\n\n\n\n", absoluteValue (-6.0) / 4 );

    printf("%.2f\n", squareRoot(-3.0));
    printf("%.2f\n", squareRoot(16.0));
    printf("%.2f\n", squareRoot(25.0));
    printf("%.2f\n", squareRoot(9.0));
    printf("%.2f\n", squareRoot(225.0));


    /* testing square root */

    return 0;
}

int gcd(int u, int v)
{
    int temp;

    while( v != 0)
    {
        temp = u % v;
        u = v;
        v = temp;
    }

    return u;
}

float squareRoot(float x)
{
    const  float  epsilon = .00001;
    float  guess    = 1.0;
    float returnValue = 0.0;

    if ( x < 0 )
    {
        printf ("Negative argument to squareRoot.\n");
        returnValue = -1.0;
    }
    else
    {
        while  ( absoluteValue (guess * guess - x) >= epsilon )
```

```c
            guess = ( x / guess + guess ) / 2.0;

        returnValue = guess;
    }

    return returnValue;
}

float absoluteValue(float x)
{
    if (x < 0)
        x = -x;

    return x;
}
```

```c
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>

char square[3][3] = {
                    {'1','2','3'},
                    {'4','5','6'},
                    {'7','8','9'}
                };
int player_turn = 0;
char player_icon;
int size[2] = {sizeof(square) / (sizeof(char) * 3), 3};
char position;

/* Function Prototypes */
char check_for_win(char icon);
void mark_board();
bool tie();


int main()
{
    int i;
    printf("  Tic Tac Toe\n");
    mark_board();
    while (true){
        if (player_turn % 2 == 0) player_icon = 'O';
        else player_icon = 'X';
        printf("Player %c: ",player_icon);
        scanf("%c",&position);
        getchar();
        for (int i = 0; i < 3; i++){
            for (int ii = 0; ii < 3; ii++){
                if (square[i][ii] == position) {
                    square[i][ii] = player_icon;
                }
            }
        }
        printf("\n");
        system("clear");
        mark_board();
        player_turn += 1;
        if (check_for_win('O') != false){
            printf("\nCongratulations O wins!");
            break;
        }
        else if (check_for_win('X') != false){
            printf("\nCongratulations X wins!");
            break;
        }

        if (tie()) {
            printf("\nIt's a tie");
            break;
        }
    }

    return 0;
}

void mark_board(){
    for (int i = 0; i < 3; i++){
        for (int ii = 0; ii < 3; ii++){
            ii != 2 ? printf("  %c  |",square[i][ii]) : printf("  %c",square[i][ii]);
        }
        if (i != 2) printf("\n---------------\n");
    }
    printf("\n");
}

char check_for_win(char icon){
    if (square[0][0] == icon && square[1][0] == icon && square[2][0] == icon) return icon; // vertical
    if (square[0][1] == icon && square[1][1] == icon && square[2][1] == icon) return icon; // vertical
    if (square[0][2] == icon && square[1][2] == icon && square[2][2] == icon) return icon; // vertical
    if (square[0][0] == icon && square[0][1] == icon && square[0][2] == icon) return icon; // horizontal
    if (square[1][0] == icon && square[1][1] == icon && square[1][2] == icon) return icon; // horizontal
    if (square[2][0] == icon && square[2][1] == icon && square[2][2] == icon) return icon; // horizontal
    if (square[0][0] == icon && square[1][1] == icon && square[2][2] == icon) return icon; // diagonal
    if (square[2][0] == icon && square[1][1] == icon && square[0][2] == icon) return icon; // diagonal
    return false;
}

bool tie(){
    bool t = true;
    for (int i = 0; i < size[0]; i++){
        for (int ii = 0; ii < size[1]; ii++){
            if (square[i][ii] == '1' || square[i][ii] == '2' || square[i][ii] == '3' ||
                square[i][ii] == '4' || square[i][ii] == '5' || square[i][ii] == '6' ||
                square[i][ii] == '7' || square[i][ii] == '8' || square[i][ii] == '9' )
            t = false;
        }
    }
    return t;
}

/* Output:
  Tic Tac Toe
  1  |  2  |  3
---------------
  4  |  5  |  6
---------------
  7  |  8  |  9
Player O: 1
  O  |  2  |  3
---------------
  4  |  5  |  6
---------------
  7  |  8  |  9
Player X: 2
  O  |  X  |  3
---------------
  4  |  5  |  6
---------------
  7  |  8  |  9
Player O: 3
  O  |  X  |  O
---------------
  4  |  5  |  6
---------------
  7  |  8  |  9
Player X: 4
  O  |  X  |  O
---------------
  X  |  5  |  6
---------------
  7  |  8  |  9
Player O: 5
  O  |  X  |  O
---------------
  X  |  O  |  6
---------------
  7  |  8  |  9
Player X: 6
  O  |  X  |  O
---------------
  X  |  O  |  X
---------------
  7  |  8  |  9
```

```
Player O: 7
  O  |  X  |  O
---------------
  X  |  O  |  X
---------------
  O  |  8  |  9

Congratulations O wins!
*/
```

Chapter 9

Character strings

9.1 Overview

- You can't put a char in double quotes, the compiler will interpret anything between double quotes is a string constant.
- Single quotes = chars, Double quotes = strings.
- To display a double quote character use `\"`.
- The string in memory always one character more, this is a null character represented by `\0`.
- Null character with the code value 0 is added to the end of each string to mark where it ends in memory. `\0` character is always included.
- The length of a string is always one greater than the number of characters in the string.
- Don't confuse the null character with the null characters:
  - null character is a string terminator.
  - `NULL` is a symbol that represents a memory address that doesn't reference anything.
- You can add a `\0` character to a string explicitly, in this case it will create two strings. If you add `\0` to a string in the middle the string will only be counted or recognized up until that null character.

9.1.1 Example

```c
#include <stdio.h>
int main()
{
    printf("The character \0 marks the end");
    return 0;
}
/* Output:
The character
*/
```

9.2 Defining a string

- C doesn't have a special type for strings, this means you can't use any of the operators with C for strings, you'll have to use a library.

- Strings in C are stored in an array of type char.
- Character strings are stored in adjacent memory cells.
- Example: `char my_str[20];` 19 chars taking in to account the `\0` character.
- The compiler will add the null character automatically at the end of the string constant.

Initializing:

- To initialize a character array you put the characters in between the brackets.
- Example:

```c
char word[] = {'H','e','l','l','o','!'};
```

  The compiler will automatically add the null character, declaring a char array like a string doesn't require a number of elements in between the square braces, the compiler will add that automatically.

- Short hand notation:

```c
char word[] = {"Hello!"};
```

  Let the compiler figure out the size for you. Leave the braces empty.

- You can also initialize just part of the array such as: `char str[40] = "To be";` 5 will be used, the others are actually empty.

Assigning a value to a string after initializing:

- Since strings are arrays, you can't assign strings either.
- Things such as the following are not allowed:

```c
char s[100];
s = "hello"; // error
```

- You can't assign an array of characters to another array of characters. To do that use a special function from the string library called strncpy().

Displaying a string:

- To display a string with printf use the format specifier %s:

```c
        printf("\nThe message is: %s",message);
```

- %s expects a null terminated string.
- To print a character array you can use the format specifier to format the printf(), using just the name of the character array, this is unique, this is the only type of array that enables this type of printing.

Inputting a string:

- To input string using scanf():

```c
char input[10];
printf("Please input your name");
scanf("%s",input); // no ampersand because it's an array.
```

- %s is for inputting and outputing strings.

- Don't use the & in character arrays for scanf().

Testing if two strings are equal:

- You can't do this: `if (str1 == str2);`, strings are character arrays so operators can't be applied to them. Don't use == in strings. Remember "x" is two chars because of the null char, 'x' is just one.
- If you wanted to check if two strings are equal you would actually have to use a for loop and compare each character, so there is a standard library function called strcmp(), use this to compare.

9.2.1 Example

```c
#include <stdio.h>
int main()
{
    char str1[] = "To be or not to be";
    char str2[] = ",that is the question";
    unsigned int count = 0;
    while (str1[count] != '\0') ++count;
    printf("The lenght is: %s -> %d\n", str1, count);
    count = 0;
    while (str2[count] != '\0') ++count;
    printf("The lenght is: %s -> %d\n", str2, count);
    return 0;
}
/* Output:
The lenght is: To be or not to be -> 18
The lenght is: ,that is the question -> 21

*/
```

9.3 Constant strings

- the #define is used to define constants, there are constant strings.
- Constant strings remain hence the name, constant all throughout the program. You may want to use constant strings for readability. You can later change the constant string in one place instead of everywhere you used it. Avoid magic numbers, use constants.
- #define is a preprocessor statement to define constants.
- Syntax is: #define ⟨var_name⟩ ⟨value⟩ , no semicolon or equals operator.
- All #define directives are globals, there is no thing as a local #define.
- #define can be used for char and string constants.

```c
#define BEEP 'a'
#define TEE 'T'
#define ESC '\033'
#define OPPS "Now you have done it"
```

- Another way to create constant is to use the `const` keyword. This means data defined as `const` will not be able to change, its another equivalent for #define. `const` is a read-only value, you can use consts in calculations an whatever as long as you don't alter the value, this is not allowed. const can replace magic numbers.

- Const is more flexible to #define, it lets you declare a type and allows better control over whch parts of a program can use the constant, is to say that const can be local and global.
- Example: `const int MONTHS = 12;`
- Third way to declare constants are using enums.
- For initialization of a const string:

```c
const char message[] = "The end of the world is nigh.";
```

- Any attempt to modify it by doing a strcopy() will cause an error.

9.4 String functions

- C standard library includes functions that can operate on strings.

C standard functions on strings `#include <string.h>`:

- strlen(): getting length of string.
  - This functions does change the string, string is not parametrized as a constant.

```c
#include <stdio.h>
#include <string.h>
int main(){
    char my_string[] = "my string";
    printf("len: %d",strlen(my_string));
    return 0;
}
/* Output:
len: 9
*/
```

- strcpy() and strncpy(): copying one character string to another.
  - `char s[100]; s = "hello";` doesn't work in C, you'll have to use strcpy()
  - The strcpy() doesn't check the length of the character array, thus it's prone to errors such as out of bounds errors.
  - The strncpy() takes in an third argument, the maximum number of characters to copy.

```c
#include <stdio.h>
#include <string.h>
int main(){
    char src[50], dest[50];
    strcpy(src,"This is souce\n");
    strcpy(dest,"This is destination\n");
    printf("strcpy(): %s\nstrcpy(): %s",src,dest);
    return 0;
}
/* Output:
strcpy(): This is souce

strcpy(): This is destination

*/
```

- strcat() and strncat(): used for concatenation.
  - strncat() takes consideration of the size of the string, strcat() doesn't.

```c
#include <stdio.h>
#include <string.h>
int main()
{
    char src[50], dest[50];
    strcpy(src,"This is source");
    strcpy(dest,"This is destination");
    strncat(dest,src,(sizeof(src)/sizeof(char))+1);
    printf("Concatenated str: %s",dest);
    return 0;
}
/* Output:
Concatenated str: This is destinationThis is source
*/
```

- strcmp(): compare strings to see if they are equal.
  - str1 == str2 will compare whether the strings have the same address, not the actual content.
  - in strcmp(), if the result is 0 means that they are the same string, if the result is negative or positive they are not equal.
  - If the return is negative → str1 is less than str2.
  - If return is positive → str2 is less than str1.
  - You don't have to worry about the size of arrays being compared thus no strncmp() not for ensuring no buffer overflows, its for comparing substrings. For example, you want all strings that start in "astro" you'll use strncmp(), use strncmp() when you are interested in prefixes.

```c
#include <stdio.h>
#include <string.h>
int main()
{
    printf("strcmp(\"A\",\"A\")\n");
    printf(" %d\n",strcmp("A","A"));
    printf("strcmp(\"B\",\"b\")\n");
    printf(" %d\n",strcmp("B","b"));
    printf("strcmp(\"C\",\"c\")\n");
    printf(" %d\n",strcmp("C","C"));
    return 0;
}
/* Output:
strcmp("A","A")
 0
strcmp("B","b")
 -1
strcmp("C","c")
 0

*/
```

9.4.1 String functions examples

```c
#include <stdio.h>
#include <string.h>
int main()
{
    char s[] = "My name is David";
    printf("The length is: %d\n",strlen(s));
    char temp[strlen(s)+1];
    strncpy(temp,s,sizeof(temp)+1);
    printf("temp is: %s\n",temp);

    return 0;
}
/* Output:
The length is: 16
temp is: My name is David

*/
```

9.5 Searching, Tokenizing, and Analyzing Strings

- Searching a string: To find a single character or substring use: strchr() and strstr().
- Tokenizing: Sequence of characters within a string bounded by a delimeter (space,comma,period,etc); Breaking a sentence into words is called tokenizing. Use: strtok()
- Analyzing strings: islower(), isupper(), isalpha(), etc.

9.5.1 Pointers

- A pointer is a useful type of variable, related to memory. A pointer is a variable that stores an address, its value is the address of another location in memory that can contain value.

```c
int Number = 25;
int *pNumber = &Number;
```

- *pNumber is a pointer, it points to address of Number.

Variable	Address	Value
Number	1000	25
pNumber	1004	1000

9.5.2 Searching a string for a character

- strchr() function searches a given string for a specified character. The first argument is the string to search (given as an address) and the second is the character to look for.
- The function will return a pointer to the first position in the string where the character is found. The address to this position in memory and it is of type char* or “pointer to char”.
- To store a value returned you must store it in a variable that can store the address of a character.
- If the character is not found, the function returns a null value. For a pointer null represents a pointer that is not pointing to anything.

```c
char str[] = "The quick brown fox";
char ch = 'q';
char *pGot_char = NULL;
pGot_Char = strchr(str,ch); // pointer will be pointed to the q address.
```

- In the example you can also use an int as a char to look for, it will be converted to ASCII equivalent.

9.5.3 Searching for a substring

- strstr() searches one string for the first occurrence of a substring. It returns a null pointer if non is found.
- Returns a pointer to the position in the first string where the substring is found.
- The search is case-sensitive.

```c
char text[] = "Every dog has his day";
char word[] = "dog";
char *pFound = NULL;
pFound = strstr(text,word);
```

9.5.4 Tokenizing a string

- Token is a sequence of characters within a string that is bound by a delimiter, the delimiter can be anything, best if the delimiter is something meaningful.
- strtok(): first argument is the string to be tokenized, a string containing all the possible delimiter characters.

```c
#include <stdio.h>
#include <string.h>
int main()
{
    char str[80] = "Hello-how-are-you";
    const char s[2] = "-";
    char *token;
    token = strtok(str,s);

    while (token != NULL){
        printf(" %s\n",token);
        token = strtok(NULL, s);
    }
    return 0;
}
/* Output:
 Hello
 how
 are
 you

*/
```

9.5.5 Analyzing strings

- `islower()`: lower case.
- `isupper()`: upper case.
- `isalpha()`: lower or upper case.
- `isalnum()`: upper case or lower case or a digit.
- `iscntrl()`: control character.
- `isprint()`: any printing character including a space.
- `isgraph()`: any printing character except a space.
- `isdigit()`: decimal digit (0-9)
- `isxdigit()`: hexadecimal digit (0-9,A-F,a-f)
- `isblank()`: standard blank characters, space and `\t`
- `isspace()`: white space character such as: `space, \n, \t, \v, \r, \f`
- `ispunct()`: printing character for shitch isspace is isalnum return false.

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define BUFF 100
int main()
{
    char buff[BUFF]; // input buffer
    int nLetters = 0; // # of letters to input
    int nDigits = 0; // # of digits in input
    int nPunct = 0;  // # of punctuation characters
    printf("Enter an string of less than %d characters: ",BUFF);
    scanf("%s",buff); // read a string to a buffer.
    getchar();
    int i = 0;
    while (buff[i]){
        if (isalpha(buff[i])){
            ++nLetters;
        } else if (isdigit(buff[i])){
            ++nDigits;
        } else if (ispunct(buff[i])){
            ++nPunct;
        }
        ++i;
    }
    printf("\nYour string contained %d letter, %d digits and %d punctuation
    ↪  characters.\n",nLetters, nDigits, nPunct);
    return 0;
}
/* Output:
Enter an string of less than 100 characters:
↪  asdakdl81289asdnaskd1289njadjasdas..asd,as.d,asd.ad

```

```
Your string contained 36 letter, 9 digits and 6 punctuation characters.

*/
```

9.6 Converting strings

- Use toupper() and tolower().
- Syntax is: `for (int i = 0; (buff[i]=(char)toupper(buff[i])) != '\0'; ++i);`
- This loop will cover the entire string in the buff array to uppercase by stepping through the strung one character at a time.
- toupper() returns a type int, thats why we convert it.

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define size 100
int main()
{
    char text[size], substring[40];
    printf("Enter the string to be searched (less than %d characters):
    ↪  \n",strlen(substring));
    scanf("%s",text);
    getchar();
    printf("Enter the string sought (less than %d characters): \n",strlen(substring));
    scanf("%s",substring);
    getchar();
    printf("\nFirst string entered: \n%s\n",text);
    printf("Second string entered\n%s\n",substring);

    for (int i = 0; (text[i] = (char)toupper(text[i])) != '\0'; ++i);
    for (int i = 0; (substring[i] = (char)toupper(substring[i])) != '\0'; ++i);

    printf("The second string %s found in the first.\n",((strstr(text,substring) ==
    ↪  NULL) ? "was not" : "was"));

    return 0;
}
/* Output:
First string entered:
hello
Second string entered
llo
The second string was found in the first.

*/
```

This is all included in the stdio.h.

Function	Returns
`atof()`	A value of type double that is produced from the string argument. Infinity as a double value is recognized from the strings “inf” or “INFINITY” where any character can be in uppercase or lowercase and “not a number” is recognized from the string “NAN” in uppercase or lowercase.
`atoi()`	A value of type int that is produced from the string argument.
`atol()`	A value of type long that is produced from the string argument.
`atoll()`	A value of type long long that is produced from the string argument.
`strtod()`	A value of type double is produced from the initial part of the string specified by the first argument. The second argument is a pointer to a variable, ptr say, of type char* in which the fucntion will store the address of the first character following the substring that was converted to the double value. If no string was found that could be converted to type double, the variable ptr will contain the address passed as the first argument.
`strof()`	A value of type float. In all other respects it works as strtod().
`strtold()`	A value of type long double. In all other respects it works as strtod().

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
int main()
{
    double value = 0;
    char str[] = "3.5 2.5 1.26";    // The string to be converted
    char *pstr = str;               // pointer to the string to be converted
    char *ptr = NULL;               // pointer to char position after convertion
     ↪
    while (true){
        value = strtod(pstr,&ptr);  // Converting starting at pstr
        if (pstr == ptr) break;     // pstr stored of no convertion, so we are done.
        else {
            printf(" %f",value);    // output resultant value
            pstr = ptr;             // store start for the next convertion
        }
    }
    return 0;
}
/* Output:
 3.500000 2.500000 1.260000
*/
```

9.7 Challenge - Understanding char arrays

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int  length(const char str[]);
bool compare(const char str1[], const char str2[]);
void concatenate(char result[], char str1[], char str2[]);

int main()
{
    char str1[] = "Hello World";
    char str2[] = "Hello Worl";
    char result[length(str1) + length(str2) + 1];
    printf("%d\n",length(str1));
    printf("%d\n",compare(str1,str2));
    concatenate(result,str1,str2);
    printf("%s",result);

    return 0;
}


int  length(const char str[]){
    int i = 0;
    while (str[i] != '\0'){
        i++;
    }
    return i + 1;
}
void concatenate(char result[], char str1[], char str2[]){
    int i,j;
    for (i = 0; str1[i] != '\0'; ++i) result[i] = str1[i];
    for (j = 0; str2[j] != '\0'; ++j) result[i + j] = str2[j];
    result[i + j] = '\0';
}
bool compare(const char str1[], const char str2[]){
    int i = 0;
    if (length(str1) == length(str2)){
        while (str1[i] != '\0' && str2[i] != '\0'){
            if (str1[i] != str2[i] ){
                return false;
            }
            ++i;
        }
        return true;
    } else {
        return false;
    }
}
/* Output:
12
```

```
0
Hello WorldHello Worl
*/
```

9.7.1 Reverse a string

```c
#include <stdio.h>
#include <string.h>
char reverse(char reversed[], const char str[]);

int main()
{
    char str[] = "Hello world";
    char reversed[strlen(str)];
    reverse(reversed,str);
    printf("%s",reversed);
    return 0;
}

char reverse(char reversed[], const char str[]){
    int i,j;
    for (i = 0, j = strlen(str) - 1; i < strlen(str);){
        reversed[i] = str[j];
        i++;
        j--;
    }
    reversed[i] = '\0';
}

/* Output:
dlrow olleH
*/
```

Chapter 10

Debugging

10.1 Overview

- Debugging is the process of finding and fixing errors in a program, usually logic or compiler/syntax errors.
- For syntax errors, understand what the compiler is telling you.
- For always focus on fixing the first problem detected.
- Debugging can vary in complexity.
- It's a tremendous skill for programming.
- Debugging saves time and money.
- Maintenance phase is the most expensive phase of the software life cycle, when windows releases a new operating system they have the responsibility to fix bugs that may arise, this is why automatic updates are pushed to you computer sometimes.
- Understand, bugs are unavoidable, it's ok.

10.1.1 Common problems

- Logical errors: it compiles but it doesn't function as intended, example: having a loop do more iterations as it is intended. These are caught by the programmer on execution.
- Syntax errors: errors related to not following the rules of the language, example: not putting a semi-colon at the end of a statement. These are caught by the compiler.
- Memory corruption: not freeing memory can lead to this.
  - Memory not being used
  - Memory leeks
  - Dangling references
- Performance / Scalability: as the users or features become greater in number, the app crashes.
- Lack of cohesion: a function is doing too much, areas not used.
- Tight coupling (dependencies): your algorithm is dependent on a lot of things.

In writing code, you always want to strive for high cohesion and low coupling.

10.1.2 Debugging process

- Understand the problem: test, understand requirements.
- Reproduce the problem:
  - Sometimes very difficult as problems can be intermittent or only happen in very rare circumstances.
  - Parallel process or threading problems.
- Simplify the problem / divide and conquer / isolate the source.
  - Remove parts of the original test case.
  - Comment out the code / back out changes
  - Turn a large program into a lot of small programs (unit testing).
- Identify origin of the problem (in the code):
  - Use debugging tools if necessary.
- Solve the problem:
  - Experience and practice
  - Sometimes includes redesign or refactor of code.
  - Test like crazy, verify.

10.1.3 Techniques and tools

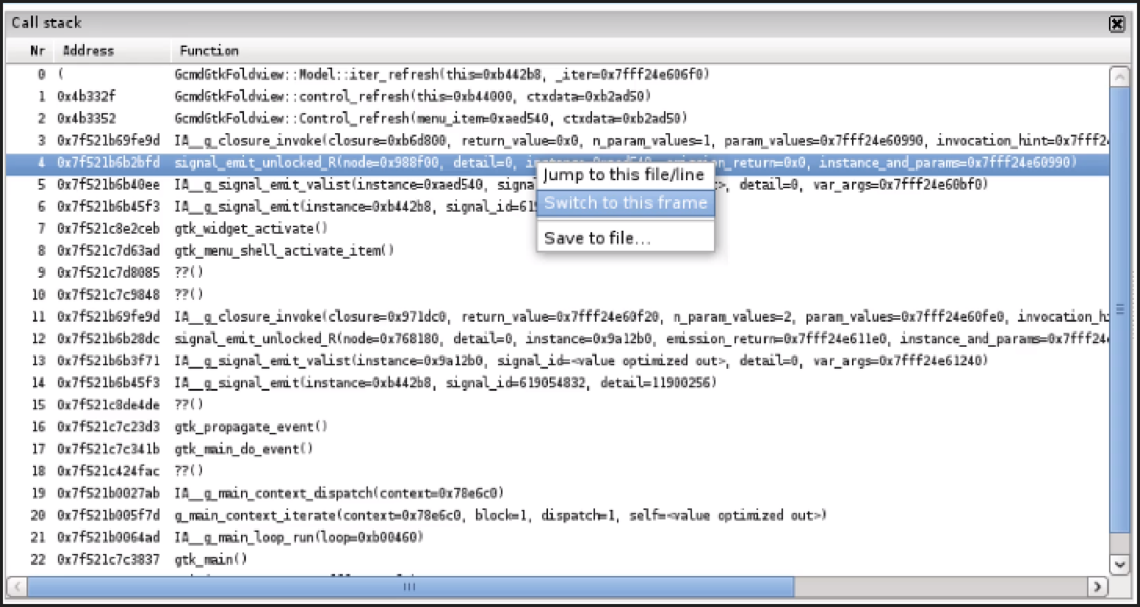
- Tracing / using print statements:
  - Output values of variables at certain points of a program
  - Show the flow of execution
- Debuggers: monitor the execution of a program, stop it, restart it, set breakpoints and watch variables in memory.
- Log files: can be used for analysis, add "good" log statements to your code, useful for un-reproducible bugs.
- Monitoring software: tun-time analysis of memory usage, network traffic, thread and object information.
- Exception handling: C doesn't support that.
- Static analyzers: analyze the source code for specific set of known problems, analyze the code without it running:
  - Semantic checker, does not analyze syntax.
  - Can detect things like uninitialized variables, memory leaks, unreachable code, deadlocks or race condition.
  - It's the intellisense.
- Test suites: run a set of comprehensive system end-to-end tests.
- Debugging the program after it has crashed:
  - Analyze the call stacks
  - Analyze memory dump (core file generated)

10.1.4 Preventing errors

- Writing high quality code.
  - Using good variable names
  - Using efficient implementations
- Unit test: automatically executed when compiling:
  - Helps avoid regression
  - Finds errors in new code before it is delivered
  - TDD (Test Driven Development)
  - If you unit testing is very large, this is an indicator of poor quality code.
- Provide good documentation and proper planning (write down design on paper and utilize pseudo code).
- Work in steps and constantly test after each step.
  - Avoid too many changes at once.
  - When making changes, apply them incrementally. Add one change, then test thoroughly before starting the next step.
  - Helps reduce the possible sources of bogs, limits problem set.

10.2 Understanding the call stack

- A stack trace (call stack) is generated whenever your app crashes because of a fatal error, a stack is a data structure, it's useful because you can see what steps your program took to end up crashing.
- A stack trace shows a list of function calls that lead to the error.
  - Includes the filenames and line numbers of the code that cause the exception or error to occur.
  - Top of the stack contains the last call that caused the error (nested calls).
  - Bottom of the stack contains the first call that started the chain of calls to cause the error.
  - You need to find the call in your application that is causing the crash.
  - It contains the line and the function that produced the error. Useful information for debugging.
- A programmer can also dump the stack trace, if you know that an error is going to happen you can dump it.
- You can build your program for release or with debugging information.



```
Nr Address        Function
 0 (              GcmdGtkFoldview::Model::iter_refresh(this=0xb442b8, _iter=0x7fff24e606f0)
 1 0x4b332f       GcmdGtkFoldview::control_refresh(this=0xb44000, ctxdata=0xb2ad50)
 2 0x4b3352       GcmdGtkFoldview::Control_refresh(menu_item=0xaed540, ctxdata=0xb2ad50)
 3 0x7f521b69fe9d IA__g_closure_invoke(closure=0xb6d800, return_value=0x0, n_param_values=1, param_values=0x7fff24e60990, invocation_hint=0x7fff2
 4 0x7f521b6b2bfd signal_emit_unlocked_R(node=0x988f00, detail=0, i                    n_return=0x0, instance_and_params=0x7fff24e60990)
 5 0x7f521b6b40ee IA__g_signal_emit_valist(instance=0xaed540, signa                    >, detail=0, var_args=0x7fff24e60bf0)
 6 0x7f521b6b45f3 IA__g_signal_emit(instance=0xb442b8, signal_id=61
 7 0x7f521c8e2ceb gtk_widget_activate()
 8 0x7f521c7d63ad gtk_menu_shell_activate_item()
 9 0x7f521c7d8085 ??()
10 0x7f521c7c9848 ??()
11 0x7f521b69fe9d IA__g_closure_invoke(closure=0x971dc0, return_value=0x7fff24e60f20, n_param_values=2, param_values=0x7fff24e60fe0, invocation_h
12 0x7f521b6b28dc signal_emit_unlocked_R(node=0x768180, detail=0, instance=0x9a12b0, emission_return=0x7fff24e611e0, instance_and_params=0x7fff24
13 0x7f521b6b3f71 IA__g_signal_emit_valist(instance=0x9a12b0, signal_id=<value optimized out>, detail=0, var_args=0x7fff24e61240)
14 0x7f521b6b45f3 IA__g_signal_emit(instance=0xb442b8, signal_id=619054832, detail=11900256)
15 0x7f521c8de4de ??()
16 0x7f521c7c23d3 gtk_propagate_event()
17 0x7f521c7c341b gtk_main_do_event()
18 0x7f521c424fac ??()
19 0x7f521b0027ab IA__g_main_context_dispatch(context=0x78e6c0)
20 0x7f521b005f7d g_main_context_iterate(context=0x78e6c0, block=1, dispatch=1, self=<value optimized out>)
21 0x7f521b0064ad IA__g_main_loop_run(loop=0xb00460)
22 0x7f521c7c3837 gtk_main()
```

- The question marks are unknown steps, on release builds the call stacks are empty, and in builds with debugging information they are populated, in releases this information is excluded so that the executable is lighter.

10.3 Code blocks debugger

- Enable the -g flag
- Disable optimization flags.

10.4 Common C mistakes

- Misplacing a semi-colon, example: `if (j == 100); j = 0;`, j will not be counted as part of the if statement.
- Confusing the = with the == .
- Omitting function prototype declaration.
- Failing to include the header file that include the definition for a C-programming library function.
- Confusing a character constant and a character string.
- Using the wrong bounds for an array.
- Confusing the operator -¿ with the operator . when referencing structure members.
  - The . is for structure variables.
  - The -¿ is used for structure pointer variables.
- Omitting the & in scanf() for non-pointer variables.
- Using a pointer variable before it's initialized.
- Omitting the break statement at the end of a case in a switch statement.
- Inserting a semicolon at the end of a preprocessor definition.
- Omitting a closing parenthesis or closing quotation marks on any statement.

10.5 Understanding compiler errors and warnings

- Use the -Wall flag to turn everything on.
- The compiler will show two types of problems: errors and warnings.
- In the -Wall flag will not allow you to compile the executable if there are any warnings.
- Types of errors:
  - variable undeclared: not declaring a variable before using it.
  - warning: implicit declaration of function. You have not placed the function prototype. Or a warning appears saying that the function is used in the code but no previous information has been given about it.
  - warning: control reaches end of non-void function: you forgot a return statement in a function.
  - warning: unused variable: variables not used.
  - undefined reference to “”: linking is not posible because there is a function with no definition.
  - error: conflicting types: different function prototypes and function declarations.
- Run-time errors:
  - Array out of bounds, that are produced in run time.
- Google the error if you don’t understand it.

Chapter 11

Pointers

11.1 Overview

- Are similar to the concept of indirection.

- Running errands is a concept of indirection, when you ask someone to do something, and they do it that's indirection.

- In programming languages, indirection is the ability to reference something using a name, reference, or container, instead of the value itself.

- The most common form of indirection is the act of manipulating a value through its memory address.

- A pointer provides an indirect means of accessing the value of a particular data item:

  - A variable whose value is a memory address.
  - Its value is the address of another location in memory that can contain value.

- It makes sense to use pointers in the same respect to the initial example of the utility of addresses, C pointers are one of the most powerful tools, pointers are the most complicated concept in C.

- The compiler must know the type of data stored in the variable to which it points:

  - Needs to how much memory is occupied or how to handle the contents of memory to which it points.
  - Every pointer will be associated with a specific variable type.
  - It can be used only to point to variables of that type.

- Pointers of type pointer int can only point to type int variables.

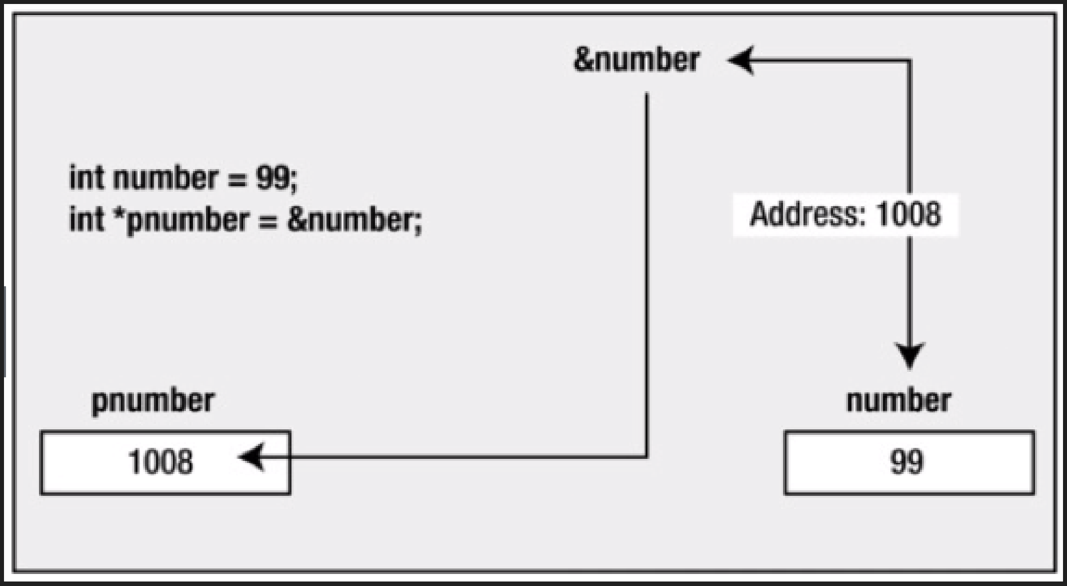


Figure 11.1: Taken from beginning C, Horton.

- A pointer has a value to an address, the value of &number is the address where number is located. This value is used to initialize pnumber in the second statement.
- What a pointer holds is the value of an address to a variable.

11.1.1 Why use pointers?

- Accessing data by means of only variables is very limiting.
  - With pointers, you can access any location, you can treat any position of memory as a variable for example, and in addition perform arithmetic with pointers.
- Pointers in C make it easier to use arrays and strings.
- Pointers allow you to refer to the same space in memory from multiple locations:
  - This means that you can update memory in one location and the change can be seen from another location in you program.
  - Can also save space by being able to share components in you data structures.
- Pointers allow functions to modify data passed to them as variables:
  - Pass by reference: by arguments to function in way they can be changed by function.
- Can also be used to optimize a program to run faster or use less memory than it would otherwise use.
- Pointers allow us to get multiple values from the function.
  - A function can only return one value, but by passing arguments as pointers we can get more than one values from the pointer.
- With pointers dynamic memory can be created according to the program use:
  - We can save memory from static (compile time) declarations.
- Pointers allow us to design and develop complex data structures like stack, queue, or linked lists.
- Pointers provide direct memory access.
- Allows encapsulating global variable in functions as well.

11.2 Defining pointers

- A pointer is not declared the same way as a normal variable, you can't say: `pointer ptr;`.
- It is not enough to say that a variable is a pointer:
  - You can also have to specify the kind of variable to which the pointer points.
  - Different variable types take up different amounts of storage in memory.
  - Some pointer operations require knowledge of that storage size.
- You declare a pointer to a variable of type int with syntax: `int *pnumber;`
- The type of the variable with the name pnumber is `int*`:
  - Can store the addresses of any variable of type int.
- Syntax is: ⟨data type⟩ * ⟨variable name⟩ ;
- Types matter.
- The space between the * and the pointer name is optional:
  - Programmers use the space in declaration and omit it when dereferencing a variable.
- The value of a pointer is an address, and it is represented internally as an unsigned integer on most systems.
  - However, you shouldn't think of a pointer as an integer type.
  - There are things you can do with integers that you can not do with pointers, and vice versa.
  - You can multiply one integer by another, but you can not multiply one pointer by another.
- A pointer is really a new type, not an integer type.
  - %p represents the format specifier for pointers.
- The previous declarations create the variable but do not initialize it:
  - Dangerous when not initialized.
  - You should always initialize a pointer when you declare it.

11.2.1 NULL Pointers

- You can initialize a pointer so that it does not point to anything. `int *pnumber = NULL;`
- NULL is a constant that is defined in the standard library, it is the equivalent of zero for a pointer.
- NULL is a value that is guaranteed not to point to any location in memory:
  - Means that it implicitly prevents the accidental overwriting of memory by using a pointer that does not point to anything specific.
- NULL is not `\0` character in strings they are diferent.
- Add an #include directive for stddef.h to your source file.

11.2.2 Address of operator

- If you want to initialize your variable with the address of a variable you have already declared:
  - Use the address of operator &.

```
int number = 99;
int *pnumber = &number;
```

- The initial value of pnumber is the address of the variable number.
  - The declaration of number must precede the declaration of the pointer that stores its address.
  - Compiler must have already allocated space and thus address for number to use it to initialize pnumber.

11.2.3 Precautions

- There is nothing special about the declaration of a pointer:
  - You can declare regular variables and pointers in the same statement: `double value, *pVal, fnum;`
- Only the second variable pVal is a pointer.
- `int *p,q;` this declaration is of a pointer of type int* and a variable q of type int; a common mistake is to think that both p and q are pointers.
- Also, a good idea to use names beginning with p as pointer names.

11.3 Accessing pointers

11.3.1 Access pointer values

- You can use the indirection operator *, to access the value of the variable pointed to by a pointer; also referred to as the deference operator because you use it to "dereference" a pointer.

```
int number = 15; // to get this value
int *pnumber = &number;
int result = 0;
```

- To get this value: the pointer variable contains the address of the variable number:
  - You can use this in an expression to calculate a new value for result.

```
result = *pointer + 5; // result is 20
```

- The expression *pointer will evaluate to the value stored at the address contained in the pointer:
  - The value stored in number, 15, so result will be set to 15 + 5, which is 20.
- The indirection operator, *, is also the symbol for multiplication, and it is used to specify pointer types:
  - Depending on where the asterisk appears, the compiler will understand whether its should interpret it as an indirection operator, as multiplication sign, or as part of a type specification.
  - Context determines what it means in any instance.

- Dereferencing is the process of which through a pointer I'm able to obtain the value stored in the variable of which the pointer is pointing to.

```c
#include <stdio.h>
int main()
{
    int count = 10,x;
    int *int_pointer;

    int_pointer = &count;
    x = *int_pointer;

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

    return 0;
}
/* Output:
count = 10, x= 10
*/

```

11.3.2 Displaying a pointer's value

- To output the address of a variable, you use the output format specifier %p:
  - Outputs a pointer value as a memory address in hexadecimal form.

```c
#include <stdio.h>
int main()
{
    int number = 0;
    int *pnumber = NULL; // good practice to initialize it to null.
    number = 10;
    pnumber = &number;
    printf("pnumber's   value: %p\n",pnumber);
    // print the address of number
    printf("number's  address: %p\n",&number); // the address of number
    printf("pnumber's address: %p\n",(void*)&pnumber); // address of the pointer
    // the (void*) is to get around a warning.
    return 0;
}
/* Output:
pnumber's   value: 0xffffcbfc
number's  address: 0xffffcbfc
pnumber's address: 0xffffcbf0

*/


```

- Pointers occupy 8 bytes (depending on you machine) and the addresses have 16 hexadecimal digits:
  - If a machine has 64-bit operating system and my compiler supports 64-bit addresses, some compilers only support 32-bit addressing, in which case addresses will be 32-bit addresses.

- Remember, a pointer itself has an address, just like any other variable:
  - You use %p as the conversion specifier to display an address.
- You use the & (address of) operator to reference the address that the pnumber variable occupies.
- The cast to void* is to prevent a possible warning from the compiler:
  - The %p specification expects the value to be kind of pointer type, but the type of &pnumber is “pointer to a pointer to int”.

11.3.3 Displaying a number of bytes you are using

- You use the sizeof operator to obtain the number of bytes a pointer occupies:
  - On my machine this shows that a pointer occupies 8 bytes.
  - A memory address on my machine is 64 bits.
- You may get a compiler warning when using sizeof this way:
  - size_t is an implementation-defined integer type.
  - To prevent the warning, you could cast the argument to type int like this:

```c
printf("pnumber's size: %d bytes\n",(int)sizeof(pnumber));
```

```c
#include <stdio.h>
int main()
{
    int number = 0; // var of type int
    int *pnumber = NULL; // a pointer that can point to type int
    number = 10;
    printf("number's address: %p\n",&number); // output the address
    printf("number's   value: %d\n\n",number); // output the value

    pnumber = &number; // store address of number in pnumber.

    printf("pnumber's address: %p\n",(void*)&pnumber); // output the address
    printf("pnumbe's     size: %zd bytes\n",sizeof(pnumber)); // output the size
    printf("pnumber's   value: %p\n",pnumber); // output the value (an address)
    printf("value pointed to: %d\n",*pnumber); // value at the address, dereferencion
     ↪  with *
    return 0;
}
/* Output:
number's address: 0xffffcbfc
number's   value: 10

pnumber's address: 0xffffcbf0
pnumbe's     size: 8 bytes
pnumber's   value: 0xffffcbfc
value pointed to: 10

*/
```

11.4 Challenge

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int num = 150;
    int *pNum = NULL;
    pNum = &num;
    printf("num address is:  %p\n", &num);
    printf("Address of pNum: %p\n", &pNum);
    printf("value of the pNum: %p\n", pNum);
    printf("value of what pNum is pointing to: %d\n", *pNum);
    return 0;
}
/* Output:
num address is:  0xffffcbfc
Address of pNum: 0xffffcbf0
value of the pNum: 0xffffcbfc
value of what pNum is pointing to: 150

*/
```

11.5 Overview

- C offers several basic operations you can perform on pointers.
- You can assign an address to a pointer:
  - Assigned value can be an array name, a variable preceded by address operator &, or another second pointer.
- You can also dereference a pointer:
  - The * operator gives the value stored in the pointed-to location.
- You can take a pointer address:
  - The & operator tells where the pointer itself is stored.
- You can perform pointer arithmetic:
  - Use the + operator to add an integer to a pointer or a pointer to an integer (integer is multiplied by the number of bytes in the pointed-to type and added to the original address)
  - Increment a pointer by one (useful in arrays when moving to the next element)
  - Use the operator to subtract an integer from a pointer (integer is multiplied by the number of bytes in the pointed-t type and subtracted from the original address).
  - Decrementing a pointer by one (useful in arrays when going back to the previous element) this means when you're iterating an array and you don't want to use the index, you can increment it by one.
- You can find the difference between two pointers:

  - You do this for two pointers to elements that are in the same array to find out how far apart the elements are. Example: you have a pointer to index 5 and a pointer to index 15 in an array you can subtract the pointers and find the difference.

- You can use the relational operators to compare the values of two pointers:
  - Pointers must be the same type.
  - This means you can find out if an address is greater than another.

- Remember, there are two forms of subtraction:
  - You can subtract one pointer from another to get an **integer**.
  - You can subtract an integer from a pointer and get a **pointer**.

- Be careful when you use incrementing or decrementing pointers and causing an array "out of bounds" error:
  - Computer does not keep track of whether a pointer still points to an array element or to another arbitrary point in memory.
  - If you increment too much in an array using pointer arithmetic you will get an array "out of bounds", if you decrement the array too much you will get an "under bounds" error, this will be apparent in debugging since these errors are that occur in runtime, not compile time.

- The value referenced by a pointer can be used in arithmetic expressions:
  - If a variable is defined to be if type "pointer to integer" then it is evaluated using the rules of integer arithmetic.

```c
int number = 0;
int *pnumber = NULL;
number = 10;
pnumber = &number;
*pnumber += 25; // The value pointed to is incremented to 25
```

  - Increment the value of the number variable by 25.

  - * indicates you are accessing the contents to which the variable called pnumber is pointing to.

- If a pointer pointes to a variable x:
  - That pointer has been defined to be a pointer to the same data type as is x.
  - Using the *pointer in an expression is identical to the use of x in the same expression.

- A pointer can contain the address of any variable of the appropriate type:
  - You can use one pointer variable to change the values of many variables.
  - As long as they are of type compatible with the pointer type.

```c
#include <stdio.h>
int main()
{
    long num1 = 0L;
    long num2 = 0L;
    long *pnum = NULL;
    pnum = &num1;
    *pnum = 2L;
    ++num2;
    num2 += *pnum;
    pnum = &num2;
    ++*pnum;
    printf("num1 = %ld\n",num1);
    printf("num2 = %ld\n",num2);
    printf("*pnum = %ld\n",*pnum);
    printf("pnum = %p\n",pnum);
    printf("*pnum + num2 = %ld\n",*pnum+num2);
    return 0;
}
/* Output:
num1 = 2
num2 = 4
*pnum = 4
pnum = 0xffffcbe8
*pnum + num2 = 8

*/
```

11.5.1 When reciecing input

- When we have used scanf() to input values, we used the & operator to obtain the address of a variable:
  - On the variable os that is to store the input.
  - We used & on all variables except the character array because the array name is a a character pointer.
- scanf()'s second argument is a pointer, so whenecer you have a pointer that already contains an address, you can use the pointer name as an argument for scanf().

```c
int value = 0;
int *pvalue = &value;
printf("Input an integer: ");
scanf("%d",pvalue);
printf("You entered %d.\n",value);
```

11.5.2 Testing for NULL

- There is one rule you should burn into you memory.
  - Do not dereferene an uninitialized pointer.

```c
int *pt; // uninitialized pointer.
*pt = 5 // terrible error.
```

- The second line means store the value 5 in the location to hoch pt points:
  - pt has random value, there is no knowing ehre 5 will be placed.
  - It can be be placed anywhere in memory, this might be harmless or might overwrite data or code, or it might cause the program to crash.
- Creating a pointer only allocates memory to store the pointer itself:
  - It does not allocate memory to store data, only to hold the pointer itself, youb can't dereference it.

  - Before you use a pointer, it should be assigned a memory location that has already been allocated.
    - Assign the address of an existing variable to the pointer.
    - Or you can use dynamic memory allocation with the malloc() function to allocate memory first.
  - So if you create a variable always initialize it to NULL and test your code for null pointers.

- You can test for NULL to make sure you don't have a dereferencer and your program doesn't crash:
  - We already know that when declaring a pointer that does not point to anything, we should initialize it to NULL. NULL points to nothing. `int *pvalue = NULL;`
  - NULL is a special symbol in C that represents the pointer equivalent to 0 with ordinary numbers.
  - Another way to set a poiter to NULL is to assign it to zero. `int *pvalue = 0;`
  - Because NULL is the equivalent of zero, if you want to test whether pvlue is NULL, you can do this `(pvalue == NULL)` OR `if (!pvalue) {<code>;}`.
  - You want to check for NULL before you dereference a pointer:
    - Often when pointers are passed to functions, functions use pointers as parameters, if the caller is passing in bad data that doesn't have memory allocated it can crash you code, this you must do always in functions, always have these checks.
    - To avoid crashes.

11.6 Pointes and const

- When we use the const modifier on a variable or array it tells the compiler that the contents of the variable/array will not be changed by the program.
- With pointers, we have to consider two things when using const modifiers:
  1. Whether the pointer will be changed
  2. Whether the value that the pointer points to will be changed.

11.6.1 Pointers to constants

- You can use the const keyword when you declare a pointer to indicate that the value pointed to must be changed.
- Syntax is: const ⟨type⟩ * ⟨name⟩ = ⟨address⟩ ;

```c
long value = 9999L;
const long *pvalue = &value; // defines pointer to a cosntant.
```

- You have declared the value pointed to by pvalue to be cosnt.
  - The compiler will check for any statements that attempt to modify the value pointed to by the pvalue and flag such statements as an error.
- The following statement will now result in an error message from the compiler:

```c
*pvalue = 8888L; // Error - attempt to change const location.
```

- We would still be able to change value, we just can't change *pvalue. Value is defined as a long not a constant so value can be changed but you can't change the pointer to a constant. You applied const to the pointer.

```c
value = 7777L; // allowed
```

- The value pointer to has changed, but you did not use the pointer to make the change, you used the variable.
- The pointer itself is not a constant, it's a pointer to constant, we declared the pointer to be of type pointer to constant, you can still change what it points to.

```c
long number = 8888L;
pvalue = &number; // OK - Changing the address in pvalue.
```

- Will change the address stored in pvalue to point to number.
  - Still, cannot use the pointer to change the value that is stored.
  - You can change the address stored in the pointer as much as you like.
  - Using the pointer to change the value pointed to is not allowed, even after you have reached the address stored in the pointer.

11.6.2 Constant pointers

- You might also want to ensure that the address stored in the pointer cannot be changed.
- You can do this by using the const keyword in the declaration of the pointer.
- Syntax is: ⟨type⟩ * ⟨const⟩ ⟨name⟩ = ⟨constant address⟩ ;

```c
int count = 43;
int *const pcount = &count; // defines constant pointer
```

- The above ensures that a pointer always points to the same thing.
  - It indicates that the address stored must not be changed.
  - Compiler will check that you do not inadvertently attempt to change what the pointer points to elsewhere in your code.

```c
int item = 34;
pcount = &item; // Error - attempt to change a constant pointer.
```

- Its all about where you place the const keyword, either before the type or after the type.

```c
const int * // value can not be changed.
int *const // pointer address cannot change
```

- You can still change the value that pcount points to using pcount.

```c
*pcount = 345; // Ok - changes the value of count
```

- References the value stored in count through the pointer and changes its value to 345.
- You can create a constant that points to a value that is also a constant:

```c
int item = 25;
const int * const pitem = &item;
```

- The pitem is a constant pointer to a constant so everything is fixed, you cannot:
  1. Change the address stored in pitem.
  2. Use pitem to modify what it points to.
- You can still change the value of item directly:
  - If you wanted to make everything not change, you could specify item as const as well. That will make it so that the pointer and the variable cannot change.

11.7 Void pointers

- Remember a void function, a void function was a function that did not return anything.
- The type name void means absence of any type.
- A pointer of type void* can contain the address of a data item on **any** type. This allows for a lot of flexibility.
- void* is often used as a parameter type or return value type with functions that deal with data in type-independent way.
- Any kind of pointer can be passed around as a value of type void* :
  - The void pointer does not know what type of object it is pointing to, so, it cannot be dereference directly (this is the part that isn't flexible, you can only use it if you know what to cast it).
  - The void pointer must first be explicitly cast to another pointer type before it is dereferenced.
- The address of a variable of type int can be stored in a pointer variable of type void*.
- When you want to access the integer value at the address stored in the void* pointer, you must first cast the pointer to type int*.
- To dereference it the syntax is: (⟨data type⟩ *) ⟨void *pointer⟩

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int i = 10;
    float f = 2.34;
    char ch = 'k';
    void *vptr;
    vptr = &i;
    printf("Value of i = %d\n",*(int *)vptr);
    vptr = &f;
    printf("Value of f = %f\n",*(float *)vptr);
    vptr = &ch;
    printf("Value of ch = %c\n",*(char *)vptr);
    return 0;
}
/* Output:
Value of i = 10
Value of f = 2.340000
Value of ch = k

*/
```

11.8 Pointers and arrays

- There is a strong relationship between pointers and arrays.
- An array is a collection of objects of the same type that you can refer to using a single name.
- A pointer is a variable that has as its value a memory address that can reference another variable or constant of a given type:
  - You can use a pointer to hold the address of different variables at different times (must be same type).
  - You must have the same type.
- Arrays and pointers seem quite different, but, they are very closely related and can sometimes be used interchangeably.
- One of the most common uses of pointers in C is as pointers to arrays.
- The main reason for using pointers to arrays are ones of notational convenience and of program efficiency.
- Pointers to arrays generally results in code that uses less memory and executes faster.
- Use pointers over arrays.
- If you have an array of 100 integers:

```c
int values [100];
```

- You can define a pointer called valuesPtr, which can be used to access the integers contained in this array:

```c
int *valuesPtr;
```

- When you define a pointer that is used to point to the elements of an array, you do not designate the pointer as type "pointer to array"
  - You designate the pointer as pointing to the type of element that is contained in the array.
- To set the valuesPtr to point to the first element in the values array, you write:

```c
valuesPtr = values; // this is going to point to the first address in the array
```

  - Note we don't use the & operator, this is because arrays are themselves pointers.
- The address operator is not used:
  - The C compiler treats the appearance of an array name without a subscript as a pointer to the array.
  - Specifying values without a subscript has the effect of producing a pointer to the first element of values.

11.8.1 Arrays and pointers

- An equivalent way of producing a pointer to the start of values is to apply the address operator to the first element of the array. You can use this one:

```
valuesPtr = &values[0]; // this is maybe clearer, since you don't know that values
↪   is an array.
```

or

```
valuesPtr = values; // shortcut, unclearer.
```

- Either one is fine and matter of programmer preference.

11.8.2 Summary

- The two expressions `ar[i]` and `*(ar+i)` are equivalent in meaning.
  - Both work if `ar` is the name of an array, and both work if `ar` is a pointer variable.
  - Using an expression such as `ar++` only works if `ar` is a pointer variable.

11.9 Pointer Arithmetic

- Don't be mislead by the name "arithmetic", itself just means adding or subtracting to a pointer.
- The real power of using pointers to arrays comes into play when you want to sequence through the elements of an array.
- Because each element of an array is an address, you can iterate through them using pointer arithmetic.

```
*valuesPtr // can be used to access the first intever to the values array, that
↪   is, values[0]
```

- To reference values[3] through the valuesPtr variable, you can add 3 to valuesPtr and then apply the indirection operator.

```
*(valuesPtr + 3)
```

- This is going to multiply the bites times the integers. This is essentially going to point to the third element in the array.
- The expression, `*(valuesPtr+i)` can be used to access the value contained in values[i].
  - To set values[10] to 27, you coud fo the following.

```
values[10] = 27;
// or, using valuesPtr, you could
*(valuesPtr + 10) = 27;
```

- Pointer arithmetic is notationally rigorous, however, it allows for a faster and more memory efficient way to access and modify array elements.
- To set valuesPtr to point to the seccond element of the values array, you can apply the address of operator to values[1] and assign the result o valuesPtr.

```c
valuesPtr = &values[1];
```

- it valuesPtr points to values[0], you can set it to point to values[1]by simply adding 1 to the values of valuesPtr.

```c
valuesPtr += 1;
```

- This is a perfectly valid expression in C can be used for pointers to any data type.
- The increment and decrement operators ++ and – are particularly useful when dealing with pointers.
  - Using the increment operator on a pointer has the same effect as adding one to the pointer.
  - Using the decrement operator has the same effect as subtracting one from the pointer.

```c
++valuesPtr;
```

- Sets valuesPtr pointing to the next integer in the values array (values[1]).

```c
--valuesPtr;
```

- Sets valuesPtr pointing to the previous integer in the values array, assuming that valuesPtr was not pointing to the beginning of the values array.
- Be careful with: Out of bounds errors.

11.9.1 Example

```c
#include <stdio.h>
#include <stdlib.h>


int main(void)
{
    int arraySum(int array[], const int n);
    int values[10] = {3,7,-9,3,6,-1,7,9,1,-5};
    printf("Sum: %i\n",arraySum(values,10));
    return 0;
}


int arraySum(int array[], const int n){
    int sum = 0, *ptr; // declaring variables
    int *const arrayEnd = array + n; // locating the end of array

    for (ptr = array /*Alternatively: ptr = &array[0]*/; ptr < arrayEnd; ++ptr )
        sum += *ptr; // adding to the value of the array

    return sum;
}
/* Output:
Sum: 21

*/
```

- To pass an array to a function, you simply specify the name of the array.
- To produce a pointer to an array, you need only to specify the name of the array.
- This implies that in the call to the arraySum() function, what was passed to the function was actually a pointer to the array values:
  - This explains why you are able to change the elements of an array from within a function considering a function can only change variables inside the function scope.
  - An array is nothing more than a pointer.
  - This is why we pass in the elements for the scanf() function.
- SO, why the formal parameter inside the function is not declared to be a pointer?

```c
int arraySum(int *array, const in n);
```

  - The above is perfectly valid.
  - Pointers and arrays are intimately related in C.
  - This is why you can declare arrays to be of type "array of ints" inside the arraySum function or to be of type "pointer to int".
  - This is why you see many functions accept arrays as parameters, the array is in itself a pointer.
- If you are going to be using index numbers to reference the elements of an array that is passed to a function, declare the corresponding formal parameter to be an array.
  - More correctly reflects the use of the array by the function.
- In terms of notation, for simplicity and less code and clarity (if you understand pointers) you might want to use pointers instead of indexes.

11.9.2 Summary

```c
int urn[3];
int *ptr1, *ptr2;
```

Valid	Invalid
ptr1++;	urn++;
ptr2 = ptr1 + 2;	ptr2 = ptr2 + ptr1;
ptr2 = urn + 1	ptr2 = urn *ptr1;

- Functions that process arrays actually use pointers as arguments.
- You have a choice between array notation and pointer notation for writing array-processing functions.
- Using array notation makes it more obvious that the function is working with arrays.
  - Array notation has more familiar look to programmers versed in FORTRAN, Pascal, Modula-2, or BASIC.
- Other programmers might be more accustomed to working with pointers and might find the pointer notation more natural:
  - Closer to machine language and, with some compilers, leads to more efficient code.

11.10 Examples of pointer arithmetic

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    char multiple[] = "a string";
    char *p = multiple; // multiple is already a pointer.

    for (int i = 0; i < strnlen(multiple, sizeof(multiple)); ++i){
        printf(
            "multiple[%d] = %c *(p+%d) = %c &multiple[%d] = %p p+%d 0 %p\n",
            i, multiple[i],i,*(p+i),i,&multiple[i],i,p+i
        );
    }
    return 0;
}
/* Output:
multiple[0] = a *(p+0) = a &multiple[0] = 0xffffcbd7 p+0 0 0xffffcbd7
multiple[1] =   *(p+1) =   &multiple[1] = 0xffffcbd8 p+1 0 0xffffcbd8
multiple[2] = s *(p+2) = s &multiple[2] = 0xffffcbd9 p+2 0 0xffffcbd9
multiple[3] = t *(p+3) = t &multiple[3] = 0xffffcbda p+3 0 0xffffcbda
multiple[4] = r *(p+4) = r &multiple[4] = 0xffffcbdb p+4 0 0xffffcbdb
multiple[5] = i *(p+5) = i &multiple[5] = 0xffffcbdc p+5 0 0xffffcbdc
multiple[6] = n *(p+6) = n &multiple[6] = 0xffffcbdd p+6 0 0xffffcbdd
multiple[7] = g *(p+7) = g &multiple[7] = 0xffffcbde p+7 0 0xffffcbde

*/
```

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
    long multiple[] = {15L,25L,35L,45L};
    long *p = multiple;

    for (int i = 0; i < (sizeof(multiple)/sizeof(multiple[0])); ++i){
        printf(
            "address p+%d (&multiple[%d]): %llu  *(p+%d)  value: %d\n",
            i,i,(unsigned long long)(p+i), i, *(p+i)
        );
    }
    printf("\nType long occupies %d bytes\n", (int)sizeof(long));
    return 0;
}
/* Output:
address p+0 (&multiple[0]): 4294953936  *(p+0)  value: 15
address p+1 (&multiple[1]): 4294953944  *(p+1)  value: 25
address p+2 (&multiple[2]): 4294953952  *(p+2)  value: 35
address p+3 (&multiple[3]): 4294953960  *(p+3)  value: 45

```

```
Type long occupies 8 bytes

*/
```

11.11 Pointers and strings

11.11.1 Overview

- We know how arrays relate to pointers and the concept of pointer arithmetic.
- These concepts can be very useful when applied to character arrays (strings).
- One of the most common applications of using a pointer to an array is as pointer to a character string.
  - The reasons are one of notational convenience and efficiency.
  - Using a variable of type pointer to char to reference a string gives you a lot of flexibility.

```c
#include <stdio.h>
#include <stdlib.h>

void copyStringArr(char to[], char from[]);
void copyStringPtr(char to[], char from[]);

int main()
{
    char from[] = "hello";
    char to[6];
    copyStringArr(to,from);
    printf("copyStringArr: %s\n",to);
    copyStringPtr(to,from);
    printf("copyStringArr: %s\n",to);
    return 0;
}
void copyStringArr(char to[], char from[]){
    int i;
    for (i = 0; from[i] != '\0'; ++i ){
        to[i] = from[i];
    }
    to[i] = '\0';
}
// More eficient
void copyStringPtr(char to[], char from[]){
    for (; *from != '\0'; ++from, ++to){
        *to = *from;
    }
    *to = '\0';
}
/* Output:
copyStringArr: hello
copyStringArr: hello

*/
```

11.11.2 char arrays as pointers

- If you have an array of characters called text, you could similarly define a pointer to be used to point to elements in text.

```c
char *textPtr;
```

- If textPtr is set to point to the beginning of an array chars called text.

```c
++textPtr;
```

- The above sets textPtr pointing to the next element character in text, which is text[1].

```c
--textPtr;
```

- The above sets textPtr pointing to the previous character in text, assuming that textPtr was not pointing to the beginning of text prior to the execution of this statement.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h> // for the strlen()

void copyString(char *to, char *from);

int main()
{
    char str1[] = "string to be copied";
    char str2[strlen(str1)];
    copyString(str2,str1);
    printf("copyString: %s\n",str2);
    return 0;
}
void copyString(char *to, char *from){
    while (*from) // the null character is evaluated as a falsy value.
        *to++ = *from++; // assign and increment in the same line, this is a post
        ↪  incrementation operator.
    *to = '\0'; // null character.
}
/* Output:
copyString: string to be copied

*/
```

11.12 Challenge Counting characters in a string

```c
#include <stdio.h>
#include <stdlib.h>

int str_lenght(const char *str);

int main()
{
    printf("str_lenght: %d\n",str_lenght("hello"));
    printf("str_lenght: %d\n",str_lenght("01234"));
    printf("str_lenght: %d\n",str_lenght("this is a string"));
    printf("str_lenght: %d\n",str_lenght("what is the lenght?"));
    printf("str_lenght: %d\n",str_lenght(""));
    return 0;
}
int str_lenght(const char *str){
    char const *begining = str;
    while (*str){
        ++str;
    }
    return (str - begining);
}
/* Output:
str_lenght: 5
str_lenght: 5
str_lenght: 16
str_lenght: 19
str_lenght: 0

*/
```

11.13 Pass by reference

- There are a few different ways you can pass data to a function.
  - Pass by value.
  - Pass by reference
- Java and C don't directly employ pass by reference, but it simulates it.
- Pass by value is when a function copies the actual value of an argument into the formal parameter of the function.
  - Changes made to the parameter inside the function have no effect on the argument.
  - It's a copy, no changes are made outside.
- C programming uses call by value to pass arguments:
  - Means the code within a function cannot alter the arguments used to call the function.
  - There are no changes in the values, though they had been changed inside the function.
  - Almost as in they are local variables, that is the reason why until this point you needed to create global data if you wanted functions to change values outside their scope.
- However, when you pass in an address, you can mimic pass by reference.

```c
#include <stdio.h>
#include <stdlib.h>

void swap(int x, int y);

int main()
{
    /* local variable definition */
    int a = 100;
    int b = 200;

    printf("Before swap, value of a: %d\n",a);
    printf("Before swap, value of b: %d\n",b);
    /* calling a function to swap the values */
    swap(a,b);
    printf("After swap, value of a: %d\n",a);
    printf("After swap, value of b: %d\n",b);

    return 0;
}
/* function definition to swap the values */
void swap(int x, int y){
    int temp;
    temp = x; /*save the value of x */
    x = y; /* put temp into y */
    y = temp; /* put temp into y */
    return;
}
/* Output:
Before swap, value of a: 100
Before swap, value of b: 200
After swap, value of a: 100
After swap, value of b: 200

*/
```

- As you notice the variables don't change, they are passed by value.

11.13.1 Passing data using copies of pointers

- Pointers and functions get along quite well together:
  - You can pass a pointer as an argument to a function and you can also have a function return a pointer as it's result.
- Pass by reference copies the address of an argument into the formal parameter:
  - The address is used to access the actual argument used in the call.
  - This means the changes made to the parameter within the function, it will affect the passed argument.
- To pass a value by reference, argument pointer are passed to the functions just like any other value:
  - You need to declare the function parameters as pointer types.
  - Changes inside the function are reflected outside the function as well, no need for global variables.
  - Unlike call by value where the changes do not reflect outside the function.

```c
#include <stdio.h>
#include <stdlib.h>

void swap(int *x, int *y);

int main()
```

```c
{
    /* local variable definition */
    int a = 100;
    int b = 200;

    printf("Before swap, value of a: %d\n",a);
    printf("Before swap, value of b: %d\n",b);
    /* calling a function to swap the values */
    swap(&a,&b); // needs to take addresses.
    printf("After swap, value of a: %d\n",a);
    printf("After swap, value of b: %d\n",b);

    return 0;
}
/* function definition to swap the values */
void swap(int *x, int *y){
    int temp;
    temp = *x; /* save the value of x */
    *x = *y; /* put temp into y */
    *y = temp; /* put temp into y */
    return;
}
/* Output:
Before swap, value of a: 100
Before swap, value of b: 200
After swap, value of a: 200
After swap, value of b: 100

*/
```

- Now the code above can change variables as if global variables were used.

11.13.2 Summary of syntax

- You can communicate two kinds of information about a variable to a function.

-
  ```c
  function1(x);
  ```
  - You can transmit the value of x and the function must be declared with the same type as x.
    ```c
    int function1(int num);
    ```

-
  ```c
  function2(&x);
  ```
  - You transmit the address of x and requires the function definition to include a pointer to the correct type.
    ```c
    int function2(int *ptr);
    ```

11.13.3 const pointer parameters

- You can qualify a function parameter using the const keyword:
  - Indicates that the function will treat the argument that is passed for this parameter as a constant.

  - Only useful when the parameter is a pointer.
  - When it's a value its not necessary because the value outside the function is anyway impossible to change without pointers, the only reason you want to use pointers is because you want outside variables to change.

- You apply the const keyword to a parameter that is a pointer to specify that a function will not change the value to which the argument points to.

```
bool SendMessage(const char* pmessage){
    // Code to send the message
    return true;
}
```

  - The type of the parameter, pmessage, is a pointer to a const char.
    - It is the char value that's const, not its address.
    - You could specify the pointer itself as const too, but tis makes little sense because the address is passed by value.
      - You cannot change the original pointer in the calling function.

- The compiler knows that an argument that is a pointer to constant data will be safe. You can't accidentally modify what the value is pointing to.

- If you pass a pointer to constant data as the argument for a parameter then the parameter must be a used like in the above. Remember that the function prototypes will need to be declared the same way.

11.13.4 Returning pointers from a function

- Return a pointer from a function is a particularly powerful capability.
  - It provides a way for you to return not just a single value, but a whole set of values.

- You would have to declare a function returning a pointer.

```
int *myfunc(){
    /* Code */
}
```

- Be careful though, there are specific hazards related to returning a pointer.
  - Use local variables to avoid interfering with the variable that the argument points to.
  - It is somewhat problematic when you are returning pointers because if you modify that pointer inside that function and then return it, it could be something that you did not intend to do.
  - Always use local variables if you don't want to affect a variable that your argument points to.

11.13.5 Pass by reference in C

- These mechanisms and techniques can be employed in the C language to mimic pass by reference using pointers and addresses.

11.14 Challenge Using pointers as parameters

```c
#include <stdio.h>
#include <stdlib.h>

void square(int *const x);

int main()
{
    int x = 10;
    printf("Before: %d\n",x);
    square(&x);
    printf("After:  %d\n",x);
    return 0;
}
void square(int *const x){ // The adress is constant.
    *x = (*x) * (*x);
}
/* Output:
Before: 10
After:  100

*/
```

11.15 Dynamic memory allocation

- This is directly related to pointers, you can only employ this on pointers.
- Whenever you define a variable in C, the compiler automatically allocates the correct amount of storage for you based on the data type. This memory allocation occurs on pointers as well. When you declare a pointer, memory is allocated for you so that it stores a pointer type.
- Up until this point whenever you assign data to a pointer, we've always assigned an existing variable.
  - If we wanted to assign data to a pointer we would use the & operator, on a variable, that variable already had memory allocated to for it.
  - What if you wanted to create a pointer, but you didn't want to assign if using the "address of" operator but instead dynamically allocate memory?
- It is frequently desirable to be able to dynamically allocate storage while a program is running.
- If you have a program that is designed to read in a set of data from a file into an array in memory, you have three choices:
  1. Define the array to contain the maximum number of possible elements at compile time. (Fixed size)
  2. Use a variable-lenght array to dimension the size of the array at runtime. (Added in C99, but it just means that the array can have expressions or variables instead of a constant, still kind of fixed size).
  3. Allocate the array dynamically using one of C's memory allocation routines. (The best choice, dynamically allocate memory as you need it).
- With the first approach, you have to define you array to contain the maximum number of elements that would be read into the array.

```c
int dataArray[1000];
```

- What this means though is that the data file cannot contain more than 1,000 elements, if it does, your program will not work.
  - If it is larger than 1,000 elements you must go back to the program, change the size to be larger and recompile it.
  - No matter what value you select, you always have the change of running into the same problem again in the future.
- Using the dynamic memory allocation functions, you can get memory storage as you need it.
  - This approach enables you to allocate memory as the program is executing.
  - Advantages: you won't create lots of memory that you might not use, you are going to only use as much as you need, and if you do need more you can allocate more, no need to go back to the source code and recompile it.
- Dynamic memory allocation depends on the concept of a pointer and provides a strong incentive to use pointers in you code.
  - Remember C is meant to be an efficient language, using low amounts of memory.
  - This provides a use case for pointers, allows to limit the amount of memory you use based on the need.
- Dynamic memory allocation allows memory for storing data to be allocated when your program executes:
  - Allocating memory dynamically is possible only because you have pointers available.
- The majority of production programs will use dynamic memory allocation.
- Allocating data dynamically allows you to create pointers at runtime that are just enough to hold the amount of data you require for the task. (Not wasting memory).

11.15.1 Heap vs Stack

- Two ways you can create memory in a program, two different data structures, you can store data on the stack and on the heap.
- Dynamic memory allocation reserves space in a memory area caller the heap.
- Essentially heap allows for more change, you can arbitrarily change the size of data objects and it sticks around a lot longer, through the entirety of your program.
- The stack is another place where memory is allocated: (more limiting)
  - Function arguments and local variables in a function are stored here.
  - When the execution of a function ends, the space allocated to store arguments and local variables is freed.
  - Everything related to functions is created on the stack.
  - Memory stored on the stack, does not stay around very long, because everything gets deleted after the program terminates.
  - This is one of the reasons why when you dynamically allocate memory it happens on the heap not on the stack. Because you control when you use it and delete it; the data stored on the stack on the other hand, automatically gets created and automatically gets deleted.

- The memory in the heap is different in that it is controlled by you:
  - When you allocate memory on the heap, it is up to you to keep track of when the memory you have allocated is no longer required.
  - You must free the space you have allocated to allow it to be reused.
  - Other languages manage the heap automatically using a garbage collector and other such tools.
  - In embedded systems you must be efficient because memory is a scarce resource and thus you have the responsibility to free the space allocated in the heap after it's used.

11.16 malloc, calloc, realloc

11.16.1 malloc

- The simplest and most common way to allocate memory is using the standard library function that allocates memory at runtime, this function is called `malloc()`:
  - Need to include the `stdlib.h` header file.
  - You specify the number of bytes of memory that you want allocated as the argument.
  - Returns the address of the first byte of memory that it allocated.
  - Because you get an address returned, a pointer is the only place to put it.

```c
int *pNumber = (int*)malloc(100);
```

- The code above demonstrates the usage of the malloc function, it's important to remember that the malloc function needs to be explicitly cast to the desired type.
- In the above, you have requested 100 bytes of memory and assigned the address of this memory block to pNumber.
  - pNumber will point to the first int location at the beginning of the 100 bytes that were allocated.
  - Can hold 25 int values on my computer because they require 4 bytes each.
  - Assumes that type int requires 4 bytes. This code will not work in some systems because not every one has int to be 4 bytes, resolve this by using the sizeof operator.

```c
int *pNumber = (int*)malloc(25*sizeof(int));
```

- The argument to malloc() above is clearly indicating that sufficient bytes for accommodating 25 values of type int should be made available.
- Also notice the cast `(int*)`, which converts the address returned by the function to type pointer to int.
  - malloc returns a pointer of type to void, so you have to cast it.
- You can request any number of bytes.
- If the memory that you requested can not be allocated for any reason:
  - malloc() returns a pointer with the value NULL.
  - It is always a good idea to check any dynamic memory request immediately using an if statement to make sure the memory is actually there before you try to use it.

```c
int *pNumber = (int*)malloc(25*sizeof(int));
if (!pNumber) { // remember NULL is a falsy value
    // < code to deal with memory allocation failure ... >
}
```

- What you should do in the event your program is not able to allocate memory, you should abort.
- Your program will crash anyways but you won't know why, display a message if this happens to save debugging time.

- You can at least display a message and terminate the program.
  - Much better than allowing the program to continue and crash when it uses a NULL address to store something.

11.16.2 Releasing memory

- When you allocate memory dynamically, you should always release the memory when it is no longer required.
- Memory that you allocate on the heap will be automatically released when your program ends:
  - Better to explicitly release the memory when you are done with it, even if it's just before you exit from the program.
  - Always release the memory, this is a rule.
- A memory leak occurs when you allocate some memory dynamically and you do not retain the reference to it, so you are unable to release the memory.
  - Often occurs within loops.
  - Because you do not release the memory when it is no longer required, your program consumes more and more of the available memory on each loop iteration and eventually may occupy it all.
- To free memory that you have allocated dynamically, you must still have access to the address that references the block of memory.
- To release the memory for a block of dynamically allocated memory whose address you have stored in a pointer, use the `free()` function:

```c
free(pNumber); // in the standard library.
pNumber = NULL;
```

  - The free() function has a formal parameter of type void*, so you can pass a pointer of any type as the argument.

- As long as pNumber contains the address that was returned when the memory was allocated, the entire block of memory will be freed for further use.
- You should always set the pointer to NULL after the memory that it points to has been freed.
  - Set the pointer to NULL to signify that that pointer is no longer pointing to anything, that memory has been freed.

11.16.3 calloc

- calloc has a couple of advantages to malloc, however its similar.
- the `calloc()` function offers a couple of advanteges over malloc().
  - It allocates memory as number of elements of a given size.
  - It initializes the memory that is allocated so that all bytes are zero.
- This is the main advantage to using calloc over malloc, you always want to initialize your data because in memory there can be data in the address allocated that you don't have any clue about. Always initialize your data with something.
- calloc() function requires two argument values:
  - Number of data items for which space is required.
  - Size of each data item.
- It's declared in the `stdlib.h` header file, just like malloc().

```c
int *pNumber = (int*)calloc(75,sizeof(int));
```

  - Here you say don't actually need to multiply by the size, it does it for you.
- The return value will be null if it was not possible to allocate the memory requested.
  - Very similar to using malloc(), but the big plus is that you know the memory area will be initialized to 0, this is the main advantage.
  - Again it's a good idea to check using an if statement if the value returned is NULL, so that you can abort everything in that case.

11.16.4 realloc

- The `realloc()` enables you to reuse or extend memory that you previously allocated using the malloc() or calloc().
  - This is a way to dynamically change how much memory is allocated, while you run your program.
  - You might use your program and figure that you need more memory, you can just call realloc and problem solved.
- realloc() expects two arguments:
  - A pointer containing an address that was previously returned by a call to malloc()or calloc().
  - The size in bytes of the new memory that you want allocated.
- This will allocate the amount of memory you specify by the second argument:
  - Transfers the contents of the previously allocated memory referenced by the pointer, (it automatically knows how much memory you allocated before because you passed in the pointer, then it will actually add to that the new needed memory), that you supply as the first argument to the newly allocated memory (the one you allocated with the malloc and calloc).
  - Returns a void* pointer to the new memory or NULL if the operation fails for some reason. In this case you should also abort.
- The most important feature of this operation is that realloc() preserves the contents of the original memory area:
  - The contents maybe fragmented (because you might not have been able to do it in sequence) but you still have all that memory that you can assign to.

11.16.5 Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    char *str = NULL; // no memory allocated to it
    /* intial memory allocation */
    str = (char*)malloc(strlen("jason")*sizeof(char)); // creates memory for 6 chars
    strcpy(str,"jason");
    printf("String = %s, Address = %p\n",str,str);

    /* Whoops, we didn't allocate enough memory to store
    ".com", no problem, use realloc */

    /* reallocating memory */
    str = (char*)realloc(str,strlen("jason.com")*sizeof(char)); // creates an aditional
    ↪  4 bytes
    strcat(str,".com");
    printf("String = %s, Address = %p\n", str, str);

    /* free the used memory */
    free(str); // always call free when you call malloc

    return 0;
}
/* Output:
String = jason, Address = 0x8000284d0
String = jason.com, Address = 0x8000284d0

*/
```

11.16.6 Guidelines

- Avoid allocating lots of small amounts of memory, it's not efficient.
  - Allocating memory on the heap carries some overhead with it.
  - Allocating many small blocks of memory will carry mush more overhead than allocating fewer larger blocks.
  - Allocate the memory that you need, but don't do it every two statements. Do large chunks.
- Only hang on to the memory as long as you need it.
  - As soon as you are finished with a block of memory on the heap, release the memory.
- Always ensure that you provide for releasing memory that you have allocated:
  - Decide where in your code you will release the memory when you write the code that allocates it.
  - Make sure you have access to that pointer to release the memory associated to it, if you need to make a pointer global you should, if you must release it in that function you must.
  - Try to reduce coupling as much as possible, don't make a "free function", don't put other functions in charge of releasing memory, release it as you can best to do this within a function that used that memory.

- Make sure you do not inadvertently overwrite the address of memory you have allocated on the heap before you have released it:
  - This will cause a memory leak.
    - If you try to write too many bytes to it, or have overflows you may have issues.
  - Be especially careful when allocating memory within a loop.
    - This can cause problems such as buffer overflows.

11.17 Challenge - Using Dynamic Memory

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int bytes = 0;
    char *str = NULL;
    printf("Enter the number of bytes: \n");
    scanf("%d",&bytes);
    str = (char *) malloc(bytes*sizeof(char));

    if (str != NULL){
        printf("Enter some text: \n");
        scanf(" ");
        gets(str);
        printf("Inputted text is: %s\n",str);
    }

    free(str);
    str = NULL;
    return 0;
}
/* Output:
Enter the number of bytes:
10
Enter some text:
DavidC
Inputted text is: DavidC

*/
```

Chapter 12

Structures

12.1 Overview

- Structures in C provide another tool for grouping elements together.
  - A powerful concept that you will use in many C programs that you develop.
- Similar to object-oriented programming, structures are thought of as an object but only with data members, no methods.
  - You can group data members together, and those can represent essentially data attributes.
- Suppose you want to store a data inside a program.
  - We could create variables for month, day, and year to store the date.

```c
int month = 9, day = 25, year = 2015;
```

- Suppose your program also needs to store the date of purchase of a particular item.
  - You must keep track of three separate variables for each date that you use in the program.
  - These variables are logically related and should be grouped together.
- It would be much better if you could somehow group these sets of three variables
  - This is precisely what the structure in C allows you to do.
  - Allows us to group together different variables to one name, you can then group different variables of that name, and access the elements in that name (called members).

12.1.1 Creating a structure

- A structure declaration describes how a structure is put together.
  - What elements are inside the structure, what members are contained in the structure.
- The `struct` keyword enables you to define a collection of variables of various types called a structure that you can treat as a single unit.

```c
struct date {
    int month;
    int day;
    int year;
}
```

  - Any data type can populate a struct, you can even put pointers in there.

- The above statement defines what a data structure looks like to the C compiler.
  - There is no memory allocation for this declaration.
- The variable names within the date structure, month, day, and year, are called members or fields:
  - Members of the structure appear between the braces that follow the struct tag name date.
  - This is not an array.

12.1.2 Using a structure

- The definition of date defines a new typa in the language.
  - Variables can now be declared to be of type struct date.
  - You can think of it as an enum, except an enum can only hold one value, structures can hold many values. This is another data type.

```c
struct date today;
```

- You can now declare more variables of type struct date.

```c
struct date purchaseDate;
```

- The above statement declares a variable to be of type struct date:
  - Memory is now allocated for the variables above.
  - Memory is now allocated for three integer values for each variable.
- Be certain you understand the difference between defining a structure and declaring variables of the particular structure type.

12.1.3 Accessing memebers in a struct

- Now that you know how to define a structure and declare structure variables, you need to be able to refer to the members of a structure.
- A structure variable name is not a pointer.
  - You need special syntax to access the members.
- You can refer to a member of a structure by writing the variable name followed by the period, followed by the member variable name.
  - The period between the structure variable name and the member name is called the member selection operator.
  - There are no spaces permitted between the variable name, the period, and the member name.
  - Syntax is: ⟨variable name⟩ . ⟨member name⟩
- To set the value of the day in the variable today to 25, you write:
  - `today.day = 25;`
  - `today.year = 2015;`
- To test the value of month to see i it is equal to 12:

```c
if (today.month == 12){
    nextMonth = 1;
}
```

- You use this syntax as if there were normal variables, following the rules of the type.

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    struct date {
        int month;
        int day;
        int year;
    };

    struct date today;
    today.month = 9;
    today.day   = 25;
    today.year  = 2015;

    printf("Today's date is %i/%i/%.2i.\n", today.month, today.day, today.year %
    ↪   100);
    return 0;
}
/* Output:
Today's date is 9/25/15.

*/
```

12.1.4 Structures in expressions

- When it comes to the evaluation of expressions, structure members follow the same rules as ordinary variables do:
  - Division of an integer structure member by another integer is performed as an integer division.

    ```c
    century = today.year / 100 + 1;
    ```

  - Follow the rules of the type when using a member.

12.1.5 Defining the structure and variable at the same time

- You do have some flexibility in defining a strucure.
  - It is valid o declare a variable to be of a particular structure type at the same time that the structure is defined. You can also define no name structures or anonymous structures.
  - Include the variable name (or names) before the terminating semicolon of the structure definition.
  - You can also assign initial values to the variables in the normal fashion.
- Declaration of struct and variable at the same time:

```c
struct date {
    int month;
```

```
    int day;
    int year;
} today; // also creating a variable named today.
```

- In the above, an instance of the structure, called today (we refer to that as an instance of the structure, to instantiate a structure in a variable), is declared at the same time that the structure is defined:
  - Today is a variable of type date.
  - Right after you do the today you can actually initialize those variables, by using the brackets and initializing the variables.

12.1.6 Un-named structures

- You also do not have to give the stucture a tag name.
  - If all the variables of a particular structure type are defined when the structure is defined, the structure name can be committed.
  - This means that the structure is going to be used just once.

```
/* No struct name */
struct { // structure declaration and ...
    int day;
    int year;
    int month;
} today; // ... structure variable declaration combined
```

- A disadvantage of the above is that you can no longer define further instances of the structure in another statement.
  - All the variables of this structure type you want in your program must be defined in one statement.

Three ways of declaring structures

1. Define the structure with a tag name and not creating a variable. Create the variable in a separate instance when the memory is allocated.
2. Define the structure and a variable at the same time, this allows to immediate usage of the declared variable, and future instances of the structures are possible.
3. Create an unnamed struct with no name, and just a variable at the bottom.

12.1.7 Intializing structures

- Intializing structures is similar to initializing arrays:
  - The elements are listed inside a pair of braces, with each element separated by a comma.
  - The initial values listed inside the curly braces must be constant expressions.

```
struct date today = {7,2,2015};
```

- Just like an array initialization, fewer values might be listed than are contained in the structure:

```
struct date date1 = {12,10};
```

- The above sets date.month to 12, and date1.day to 10 but gives no initial value to date.year.
- You can also specify the member names in the initialization list.
  - Enables you to initialize the members in any order, or to only initialize specified members.

```c
.member = value;
```

```c
struct date date1 = {.month = 12, .day = 10};
```

- Sets the year member fo the date structuere variable to 2015:

```c
struct date today = {.year = 2015};
```

12.1.8 Assignment with compound literals

- You can assign one or more to a structure in a single statement using what is known as compound literals.

```c
today = (struct date){9,25,2015};
```

- This statement can appear anywhere in the program:
  - It is not a declaration statement.
  - The type cast operator is used to tell the compiler the type of the expression.
  - The list of values follows the cast and are to be assigned to the members of the structure, in order.
  - Listed in the same way as if you were initializing a structure variable. The advantage to take here is that you don't have to initialize the individual members in separate lines, you can do it all on one line.
- You can also specify values using the .member notation:

```c
today = (struct date){.month = 9, .day = 25, .year = 2015};
```

- The advantage of using this approach is that the arguments can appear in any order.

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    struct date {
        int month;
        int day;
        int year;
    } today ;
    today = (struct date){.month = 29,.day = 06,.year = 2020};

    printf("Today: %d/%d/%d",today.month,today.day,today.year);

    return 0;
}
/* Output:
Today: 29/6/2020
*/
```

12.2 Structures and arrays

- You have seen how useful a structure is in enabling you to logically group related elements together.
  - For example, it is only necessary to keep track of one variable, instead of three, for each date that is used by the program.
  - To handle 10 different dates in a program, you only have to keep track of 10 different variables, instead of 30.
  - Structs are similar to arrays in that they group variables, but the advantage is that they can hold different data types.
- A better method for handling the 10 different dates involves the combination of two powerful features of the C programming language.
  - Structures and arrays.
  - It is perfectly valid to define an array of structures.
  - The concept of an array of structures is a very powerful and important one in C.
- Declaring an array of structures is like declaring any other kind of array.

```
struct date myDates[10];
```

- Defines an array called myDates, which consists of 10 elements:
  - Each element inside the array is defined to be of type struct date.
- To identify members of an array of structures, you apply the same rule used for individual structures.
  - Follow the structure name with the dot operator and then with the member name.
- Referencing a particular structure element inside the array is quite natural:
  - To set the second date inside the myDates array to August 8,1086:

```
myDates[1].month = 8;
myDates[1].day   = 8;
myDates[1].year  = 1986;
```

  - The above references the second element in an array of structs.

12.2.1 Intializing an array of structures

- Initialization of arrays containing structures is similar to initialization of multidimensional arrays.

```
struct myDates[5] = { {12,10,1975}, {12,30,1980}, {11,15,2005}};
```

- Sets the first three dates in the array myDate to 12/10/1975, 12/30/1950, and 11/15/2005, the other two will remain uninitialized.
- The inner pairs of braces are optional.

```
struct myDates[5] = { 12,10,1975, 12,30,1980, 11,15,2005};
```

  - This is perfectly valid, however it reduces readability to you program because you only have commas.
- If you wanted to initialize just the third element of the array to the specified value. Do this:

```c
struct date myDate[5] = { [2] = {12,10,1975}};
```

- If you just wanted to set the month and day of the secong element of the myDates array to 12 and 30:

```c
struct date myDates[5] = { [1].month = 12, [1].day = 30};
```

12.2.2 Structures containing arrays

- It is also possible to define structures that contain arrays as members:
  - Most common use is to set up an array of characters inside a structure.
- Suppose you wanted to define a structure called month that contains as its members the number of days in the month as well as a three-character abbreviation for the month name:

```c
struct month {
    int numberOfDays;
    char name[3];
};
```

  - This is not allocating any memory, it's just defining it.
- This sets up a month structure that contains an integer member called numberOfDays and a character member called name:
  - Member name is actually an array of three characters.
- You can now define a variable to be of type struct month and set the proper fields inside aMonth for January.

```c
struct month aMonth;
aMonth.numberOfDays = 31;
aMonth.name[0] = 'J';
aMonth.name[1] = 'a';
aMonth.name[2] = 'n';
// you can use strcpy as well.
```

- You can also initialize this variable to the same values like this:

```c
struct month aMonth = {31,{'J','a','n'}};
```

- You can set up 12-month structures inside an array to represent each month of the year:

```c
struct month months[12];
```

12.3 Nested structures

- C allows you to define a structure that itself contains other structures as one or more of its members.
  - This means you can have structures inside of structures.
- You have seen how it is possible to logically group the month, day, and year into a structure called date:

  - How about grouping the hours, minutes, and seconds into a structure called time.

```c
struct time {
    int hours;
    int minutes;
    int seconds;
};
```

- In some applications, you might have the need to group both a date and time together.
  - You might need to set up a list of events that are to occur at a particular date and time.
- You want to have a convenient way to associate both the date and the time together.
  - Define a new structure, called, for example dateAndTime, which contains as its members two elements.
  - Date and time.

```c
struct dateAndTime {
    struct date sdate;
    struct time stime;
};
```

- The first member of this structure is of type struct date and is called sdate.
- The second member of the dateAndTime structure os of type struct time and is called stime.
- Variables can now be defined to be of type struct dateAndTime.

```c
struct dateAndTime event;
```

12.3.1 Accessing members on a nested structure

- To reference the date structure of the variable event, the syntax is the same as referencing any member.

```c
event.sdate
```

- To reference a particular member inside one of these structures, a period followed by the member name is tacked on the end.
  - The below statement sets the moth of the date structure contained within an event to October, and adds one to the second contained within the time structure.

```c
event.sdate.month = 10;
++event.stime.seconds;
```

- The event variable can be initialized just like normal:
  - Sets the date in the variable event to February 1, 2015, and sets the time to 3:30:00.
  - Very similar to a multidimensional array.

```c
struct dateAndTime event = {{2,1,2015},{3,30,0}}
```

- You can use members' name in the initialization:

```c
struct dateAndTime event = {
    {.month = 2, .day = 1, .year= 2015},
    {.hour = 3, .minute = 30 .seconds = 0}
};
```

12.3.2 An array of nested structures

- It is also posible to set up an array of dateAndTime structures.

```c
struct dateAndTime events[100];
```

- The array events is declared to contain 100 elements of type struct dateAndTime:
  - The fouth dateAndTime contained within the array is referenced in the usual index way as events[3].
- To set the first time in the array to noon:

```c
events[0].stime.hour = 12;
events[0].stime.minutes = 0;
events[0].stime.seconds = 0;
```

12.3.3 Declare a structure within a structure

- You can define the Date structure within the time structure definition.

```c
struct Time {
    struct Date { // un-named structure
        int day;
        int month;
        int year;
    } dob;
    int hour;
    int minutes;
    int seconds;
};
```

  - We have a nested structure, but we defined it with the nested structure inside.
- Notice: the declaration is enclosed within the scope of the Time structure definition:
  - It does not exist outside it.
  - It becomes impossible to declare a Date variable external to the Time structure.
  - The only way you can use the Date structure in this example is by first instantiating a Time structure and then using the Date structure.
  - You would only do this when you need the Date to never be by itself.
  - This can be looked at as a local structure within a structure. The Date structure in this case has a local scope and has no outside reference to the outside of the Time struct.

12.4 Structures and pointers

- You can declare structures to be pointers, and you can have pointers inside the structures.
- C allows for pointers to structures.
- Pointers to structures are easier to manipulate than structures themselves.
- In some older implementations, a structure cannot be passed as an argument to a function, but a pointer to a structure can, another reason why you want a structure to be declared as a pointer.
- Even if you can pass a structure as an argument, passing a pointer is more efficient.
    - Remember if you pass a pointer as a parameter in a function, you can imitate pass by reference, however if you pass in a structure as parameter, structures can be large and can make your program inefficient.
- Many data representations use structures containing pointers to other structures.
- Much more efficient to pass in an address than all the elements of a structure.

12.4.1 Declaring a struct as a pointer

- You can define a variable to be a pointer to a struct.

```
struct date *datePtr;
```

- This does not allocate to store the structure, this only allocates memory to store the pointer, so you can pass in an address of an existing structure or allocate memory for it somewhere else, or initialize it.

- The variable datePtr can be assigned just like other pointers:
    - You can set it to point to todaysDate with the assignment statement.

```
datePtr = &todaysDate;
```

- You can then indirectly access any of the members of the date structure pointed to by datePtr.

```
(*datePtr).day = 21;
```

- The above has the effect of setting the day of the structure pointed to by datePtr to 21:
    - Parentheses are required because the structure member operator "." has precedence than the indirection operator "*".

12.4.2 Using structus as pointers

- To test the value of month stored in the date structure pointed to by datePtr.

```
if ((*datePtr).month == 12) {
    // < Code >;
}
```

- Pointers to structures are so often used in C that a special operator exists
    - The structure pointer operator -¿, which is the dash followed by the greater than sign, permits:

```c
(*x).y
```

- to be more clearly expressed as:

```c
x->y
```

- The previous if statement can be conveniently written as:

```c
if (datePtr->month == 12){
    // < Code >;
}
```

  - Think of it intuitively as: datePtr is pointing to the member month, at the same time its being dereferenced and compared to 12.

- If this were a nested structure, you would have to dereference using the structure pointer operator, and then accessing the further data members with the "." operator.

```c
#include <stdio.h>
#include <stdlib.h>

struct date { // you can create it as a global, no memory is allocated for this,
↪  its just a definition.
    int month;
    int day;
    int year;
};

int main()
{

    struct date today, *datePtr;
    datePtr = &today;

    datePtr->month = 9;
    datePtr->day = 25;
    datePtr->year = 2015;

    printf("Today's date is %i %i
    ↪  %.2i.\n",datePtr->month,datePtr->day,datePtr->year);

    return 0;
}
/* Output:
Today's date is 9 25 2015.

*/
```

Structures containing pointers

- A pointer also can be a member of a structure.

```c
struct intPtrs{
    int *p1;
    int *p2;
}
```

- A structure called intPtrs is defined to contain two integer pointers:
  - The first one called p1.
  - The second one p2.
- You can define a variable of type struct intPtrs:

```
struct intPtrs pointers;
```

- The variable pointers can now be used just like other structs.
  - Pointers itself is not a pointer, but a structure variable that has two pointers as its members.

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

struct intPtrs{
    int *p1;
    int *p2;
};

int main()
{
    struct intPtrs pointers;
    int i1 = 100, i2;

    pointers.p1 = &i1;
    pointers.p2 = &i2;
    *pointers.p2 = -97;

    printf("i1 = %i, *pointers.p1 = %i\n",i1,*pointers.p1);
    printf("i2 = %i, *pointers.p2 = %i\n",i2,*pointers.p2);
    return 0;
}
/* Output:
i1 = 100, *pointers.p1 = 100
i2 = -97, *pointers.p2 = -97

*/
```

12.4.3 Character arrays or character pointers?

- Which one suits me better?

```
struct names {
    char first[20];
    char last[20];
};
// OR
struct pnames {
    char *first;
```

```c
    char *last;
};
```

- You can do both, however, you need to understand what is happening here.
- If we say:

```c
struct names veep = {"Talia", "Summers"};
struct pnames treas = {"Brad", "Fallingjaw"};
printf("%s and %s", veep.first, treas.first);
```

- The struct names variable veep:
  - Strings are stored inside the structure.
  - Structure has allocated a total of 40 bytes to hold the two names.
  -
- The struct pnames variable traes:
  - Strings are stored wherever the compiler stores string constants.
  - The structuer holds the two addresses, which takes a total of 16 bytes on out system.
  - The struct pnames structre allocates no space to store strings.
  - It can be used only with strings that have had space allocated for them elsewhere.
    * Such as string constants or strings in arrays.
  - You need to use malloc() or calloc().
- The pointers in a pnames structure should be used only to manage strings that were created and allocated elsewhere in the program.
- One instance in which it makes sense to use a pointer on a structure to handle a string is if you are dynamically allocating that memory.
  - Use a pointer to store the address.
  - Has the advantage that you can ask malloc() to allocate just the amount of space that is needed for a string.

```c
struct namect {
    char *fname; // using pointers instead of arrays.
    char *lname;
    int letters;
};
```

- Understand that the two strings are not stored in the structure:
  - They are stored in the chunk of memory managed by malloc().
  - The addresses of the two strings are stored in the structure.
  - Addresses are what string-handing functions typically work with.
- Example:

```c
void getinfo(struct namect *pst) {
    char temp[SLEN];
    printf("Please enter your first name.\n");

    // allocate memory to hold name.
    pst->fname = (char*)malloc(strlen(temp)+1);

    // copy name to allocated memory.
    strcpy(pst->fname,temp);
    printf("Please enter your last name.\n");
    s_gets(temp,SLEN);
    pst->lname = (char*)malloc(strlen(temp)+1);
    strcpy(pst->lname,temp);
}
```

12.5 Structures and functions

- After declaring a structure named Family, how do we pass this structure as an argument to a function?

```c
struct Family {
    char name[20];
    int age;
    char father[20];
    char mother[20];
};

bool siblings(struct Family memeber1, struct Family member2){
    if (strcmp(member1.mother,member2.mother) == 0) {
        return true;
    } else {
        return false;
    }
}
```

- This function has two parameters, each of which is a structure.

12.5.1 Pointers to structures as function arguments

- You whould use a pointer to a structure as an argument:
  - It can take quite a bit of time to copy large structures as arguments, as well as requiring whatever amount of memory to store the copy of the structure.
  - Pointers to structures avoid the memory consumption and the copying time (this approach only copies the pointer argument).

```c
bool siblings(struct Family *pmember1, struct Family *pmember2){
    if (strcmp(pmember1->mother,pmember2->mother) == 0){ // you can use the ->
    ↪   since they are pointers.
        return true;
    } else {
        return false;
    }
}
```

- You can also use the const modifier to not allow any modification of the members of the struct (what the struct is pointing to).

```
bool siblings(struct Family const *pmember1, struct Family const *pmember2){
    if (strcmp(pmember1->mother,pmember2->mother) == 0){ // you can use the ->
    ↪  since they are pointers.
        return true;
    } else {
        return false;
    }
}
```

- You can also use the const modifier to not allow any modification of the pointer address.
  - Any attempt to change those structures will cause an error message during compilation.

```
bool siblings(struct Family *const pmember1, struct Family *const pmember2){ //
↪  you are passing a copy of the address, thus this is redundant, but there is
↪  nothing wrong in leaving it like this.
    if (strcmp(pmember1->mother,pmember2->mother) == 0){ // you can use the ->
    ↪  since they are pointers.
        return true;
    } else {
        return false;
    }
}
```

- The indirection operator in each parameter definition is now in front of the const keyword.
  - Not in front of the parameter name.
  - You cannot modify the addresses stored in the pointers.
  - It is the pointers that are protected here, not the structures to which they point to. The const is after the asterisk.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>


struct Family {
    char name[20];
    int age;
    char father[20];
    char mother[20];
};

bool siblings(struct Family *const pmember1, struct Family *const pmember2);
int main()
{
    struct Family var1 = {"David",20,"dad1","Mom1"};
```

```c
    struct Family var2 = {"David",20,"dad1","Mom1"};
    struct Family *pvar1 = &var1;
    struct Family *pvar2 = &var2;
    printf("%s",siblings(pvar1,pvar2)? "true": "false");

    return 0;
}
bool siblings(struct Family *const pmember1, struct Family *const pmember2){
    if (strcmp(pmember1->mother,pmember2->mother) == 0){ // you can use the -> since
    ↪  they are pointers.
        return true;
    } else {
        return false;
    }
}
/* Output:
true
*/
```

12.5.2 Returning a structure from a function

- The function prototype has to indicate this return value in the normal way.

```c
struct Date my_funct(void);
```

- This is a function prototype for a function taking no arguments that returns a structure of type Date.
- This can potentially allow you to return more than one value from a function, and more than one data type as well.
- It is more convenient to return a pointer to a structure.
  - When returning a pointer to a structure, it should be created on the heap.
  - Because all pointers are created on the heap.
  - This means it will be around for longer, and you can create it and delete it at an arbitrary time.
- Example:

```c
#include <stdio.h>
#include <stdlib.h>
#define FUNDLEN 25

struct funds {
    char bank[FUNDLEN];
    double bankfund;
    char save[FUNDLEN];
    double savefund;
};

double sum(struct funds moolah);

int main()
{
    struct funds stan = {
        "Garlig-Melon-Bank",
        4032.27,
        "Lucky's Savings and Loan",
        8543.94
    };
    printf("Stan has a total of $%.2f.\n",sum(stan));
    return 0;
}
double sum(struct funds moolah){
    return (moolah.bankfund + moolah.savefund);
}
/* Output:
Stan has a total of $12576.21.

*/
```

12.5.3 Reminder

- You should always use pointers when passing structures to a function:
  - It works on older, as well as newer versions of C, and it is quick (just a single address is copied).
  - Some versions of C don't allow passing in structures.
- However, you have less protection for your data.
  - Some operations in the called function could inadvertently affect data in the original structure.
  - Use the const qualifier to solve this problem, use the const before the data type, not after the asterisk.
- Advantages of passing structures as arguments:
  - The function works with copies of the original data, which is safer than working with the original data.
  - The programming style tends to be clearer.
- Main disadvantages to passing structures as arguments:
  - Older implementations of C might not handle the code.
  - Wastes time and space.
  - Especially wasteful to pass large structures to a function that uses only one or two members of the structure.
- Programmers use structure pointers as function arguments for reasons of efficiency and use const when necessary to make it safer.
- Passing structures by value is most often done for structures that are small.

12.6 Challenge - Declaring and initializing a structure

```c
#include <stdio.h>
#include <stdlib.h>

struct employee {
    char name[30];
    char hireDate[20];
    float salary;
};

int main()
{
    struct employee employee1 = {
        "David Corzo",
        "2020/07/02",
        4500
    };
    printf("Name: %s\nHiredate: %s\nSalary: $%.2f"
            ,employee1.name, employee1.hireDate, employee1.salary);
    return 0;
}
/* Output:
Name: David Corzo
Hiredate: 2020/07/02
Salary: $4500.00
*/
```

```c
#include <stdio.h>
#include <stdlib.h>

struct item {
    char *itemName;
    int quantity;
    float price;
    float amount;
};

void read_item(struct item *pitem);
void print_item(struct item *pitem);

int main(void)
{
    struct item it;
    struct item *pit = &it;
    read_item(pit);
    print_item(pit);

    // delete what you malloced
    free(pit->itemName); // the structure will be deleted in the heap
    return 0;
}

void read_item(struct item *pit){
    printf("Enter the item name: \n");
    pit->itemName = (char*)malloc(30*sizeof(char));
    scanf("%[^\n]%*c",pit->itemName); // strings don't need &

    if (pit->itemName == NULL) // check for memory allocated
        exit(-1);

    printf("Enter the quantity: \n");
    scanf("%d",&pit->quantity);
    // printf("%d",pit->quantity);

    printf("Enter the price: \n");
    scanf("%f",&pit->price);
    // printf("%f",pit->price);

    pit->amount = pit->price * (float)pit->quantity;
    // printf("%f",pit->amount);
}

void print_item(struct item *pit){
    printf("\nITEM: \n");
    printf("itemName: %s\n",pit->itemName);
    printf("quantity: %d\n",pit->quantity);
    printf("price: $%.2f\n",pit->price);
    printf("amount: $%.2f\n",pit->amount);
}
/* Output:
Enter the item name:
Milk cartons
Enter the quantity:
600
Enter the price:
14.55

ITEM:
itemName: Milk cartons
quantity: 600
price: $14.55
amount: $8730.00

*/
```

Chapter 13

File input and output

13.1 Overview

- Up until this point, all data that our prograam accesses is via memory (RAM):
  - Scope and variety of applications you can create is limited.
  - Using memory goes away when the program goes away.
  - Specially on embedded systems memory is scarce.
  - You want to have this ability for reading and writing persistent data on a hard drive.
- All serious business applications require more data than would fit into main memory:
  - It also depends on the ability to process data that is persistent and stored on an external device such as a disk.
- C provides many functions in the header file stdio.h for writing to and reading from external devices.
  - The external device you would use for storing and retrieving data is typically a disk drive.
  - However, the library will work with virtually any external storage device.
- With all the example up to this point, any data the user enters is lost once the program ends, this is called volatile memory meaning the data is not persistent.
  - If the user wants to run the program with the same data, he or she must enter it again each time.
  - Very inconvenient and limits programming.
  - Referred to as volatile memory.

13.1.1 Files

- Programs need to store data on permanent storage:
  - This is non-volatile (it stays around).
  - Continues to be maintained after the computer is turned off.
- A file can store non-volatile data and is usually stored on a disk or solid-state device:
  - A named section of storage.
  - stdio.h is a file containing useful information.
- C views a file as a continuous sequence of bytes.

  - A file inside a Linux or windows operating system is stored a certain way, but as far as C as a programming language goes, it's just a sequence of bytes.
  - Each byte can be read individually.
  - Corresponds to the file structure in the Unix environment.

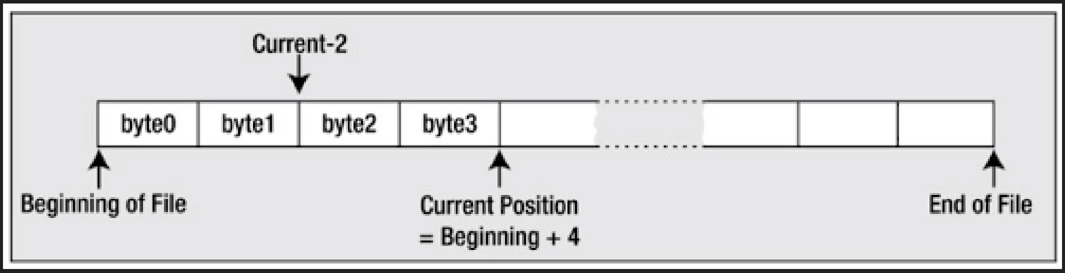


Figure 13.1: Taken from Beginning C,Horton

- A file has a beginning and an end, as well as a current position (defined as so many bytes from the beginning).
- The current position is where any file action (read/write) will take place.
  - You can move the current position to any point in the file (even the end).
  - This is relevant because we want to seek the beginning position to start writing the file, and seek the end to stop.

13.1.2 Text and binary files

- There are two ways of writing data to a stream that represents a file.
  - Text
  - Binary
- Text data is written as a sequence of characters organized as lines (each line ends with a newline).
  - Text data can vary from system to system depending on how the system interprets the bytes, for example on Windows DOS every line has a carriage return (`\r`) and a newline (`\n`) so when you read and write to a file you need to take this in to account.
- Binary data is written as a series of bytes exactly as they appear in memory.
  - Image date, music encoding non-readable.
  - These are not readable because they are just bytes, if you were to open them in a text editor you would not see anything you understand.
- You can write any data you like to a file.
  - Once a file has been written, it just consists of a series of bytes.
- You have to understand the format of the file in order to read it.
  - A sequence if 12 bytes in a binary file could be 12 characters, 12 8-bit signed integers, 12 8-bit unsigned integers, etc. This is significant because for example Windows might have extra characters when if writes data.
  - In binary mode, each and every byte of the file is accessible. This makes it extra tricky.

13.1.3 Streams

- A stream:
  - Can represent a file, but a stream is a more generic term that can represent any kind of input.
  - It can represent the keyboard, the console, a file, a socket (if you are using networking).
- C programs automatically open three files on your behaf:
  - Standard input: the normal input device for our system, usually your keyboard.
    - You use this when you do scanf()
    - You can actually use scanf() to read from files if you change the input stream.
  - Standard output: usually your display screen.
    - This is usually via the console.
  - Standard error: usually also your display screen.
    - You have to redirect it in order to see it.
- You can also redirect files on the system to be recognized as a standard input output. This is an operating system concept. Sending output to a file is an example of this, get input from a file.
- Standard input is the file that is read by getchar() and scanf().
- Standard output is used by putchar(), puts(), and printf().
- The purpose of the standard error output file is to provide a logically distinct place to send error messages.
  - To access it you need special commands.
- A stream is an abstract representation of any external source or destination for data.
  - The keyboard, the command line on your display, and files on a disk are all examples of things you can work with as streams. This will become even more significant when you program in C++.
  - The C library provides functions for reading and writing to or from data streams.
    - You use the same input/output functions for reading and writing any external device that is mapped to a stream.
    - This is why you can use scanf to read from files. It all depends on which stream we use.

13.2 Accessing files

- Files on disk have a name and the rules for namign files are determined by your operating system.
  - You may have to adjust the names depending on what OS your program is running.
- A program references a file through a file pointer (or stream pointer, since it works on more than a file):
  - You associate a file pointer with a fie programmatically when the program is run. You will create a pointer variable of type FILE in your program. We use these file pointers as a more effective way to refer to the file, because the name can cause ambiguity, the pointer will refer to what's actually being pointed to.
  - Pointers can be reused to point to different files on different occasions.
  - A file pointer should really be referred to as a stream pointer.
- A file pointer points to a struct of type FILE that represents a stream, you can actually look at the struct of type file to see what is inside it.

  - Contains information about the file:
    - Whether you want to read or write or update the file.
    - The addresses of the buffer in memory to be sued for data.
    - A pointer to the current position in the file for the next operation.
  - The above is all set via input/output file operations.
    - When you open a file it needs to know if it's a binary or a text file, so that methods such as seek adapt to the file type.

- If you want to use several files simultaneously in a program, you need a separate file pointer for each file.
  - There is a limit to the number of files you can have open at one time.
    - Defined as FOPEN_MAX in stdio.h.
    - It depends as well on the operating system you are running on.
  - It's not a good idea to have more than one file opened if it's not absolutely necessary, files are saved in persistent memory and copying all of that from persistent memory to RAM takes more time and can make you programs unresponsive.
  - You can only reuse a pointer if you are reading or writing at different times.

13.2.1 Open name

- You can associate a specific external file name with an internal file pointer variable though a process refered to as opening a file.
  - Via the `fopen()` function:
    - This function returns the file pointer for specific external file.
- The fopen() function is defined in stdio.h

```
FILE *fopen(const char * restrict name, const char * restrict mode);
```

  - This is usually your first operation while writing and reading files.
- The function fopen() takes as a first argument a pointer to a string that is the name of the external file you want to process.
  - You can specify the name explicitly or use a char pointer that contains the address of the character string that defines the file name.
  - You can obtain the file name through the command line, as input from the user, or defined as constant in you program.
- The second argument to the fopen() function is a character string that represents the file mode.
  - Specifies what to do with the file.
  - A file mode specification is a character string between double quotes.
- Assuming the call to fopen() is successful, the function returns a pointer of type FILE* that you can use to reference the file in further input/output operations using other functions in the library.
- If the file cannot be opened for some reason, fopen() returns NULL.
  - Just like with malloc, check for NULL in files.
- The second parameter can follow any of the following conventions.

Mode	Description
"w"	Open a text file for *write* operations. If the file exists, its current contents are discarded.
"a"	Open a text file for *append* operations. All writes are to the end of the file.
"r"	Open a text file for read operations.
Adding a + to the option adds the feature of creating the file if it doesn't exist, except for the r.	
"w+"	Open a text file for update (reading and writing), first truncating the file to zero lenght if it exists or creating the file if it doesn't exist.
"a+"	Open a text file for update (reading and writing) appending to the end of the existing file, or creating the file if it doesn't yet exists.
"r+"	Open a text file for update (for both reading and writing).

13.2.2 Write mode

- If you want to write an existing text file with the name myfile.txt

```c
FILE *pfile = NULL;
char *filename = "myfile.text";
pfile = fopen(filename,"w"); // open myfile.txt to write it
if (pfile == NULL)
    printf("Failed to open %s",filename);
```

- Opens the file and associates the file with the name myfile.txt with your file pointer pfile:
  - The mode as “w” means you can only write to the file.
  - You cannot read it.
- If a file with the name myfile.txt does not exist, the call to fopen() will create a new file with this name.
- If you only provide the file name without any path specification, the file is assumed to be in the current directory:
  - You can also specify a string that is the full path and name for the file.
  - If you want to access a file somewhere else, you have to specify the entire directory, also called the absolute path. Just having the name is called a relative path.
  - Relative paths are better because its not dependent on a specified file. You can also move backwards in directories using “../”. In absolute path is usually like “`C:\User...`” and if you ever moved that file your program wouldn't be able to open it.
- On opening a file for writing, the file length is truncated to zero and the position will be at the beginning of any existing data for the first operation.
  - Any data that was previously written to the file will be lost and overwritten by any write operations.
  - Without providing a + you can only do one operation, reading or writing but not both.

13.2.3 Append mode

- If you want to add to an existing text file rather than overwrite it:
  - Specify mode “a”.
  - The append mode operation.
- This positions the file at the end of any previously written data.
  - If the file does not exist, a new will be created.

```c
pFile = fopen("myfile.txt","a"); // Open myfile.txt to add to it
```

- Do not forget that you should test the return value for null each time you use it.
- When you open a file in append mode:
  - All write operations will be at the end of the data in the file on each write operation.
  - All write operations append data to the file and you cannot update the existing contents in this mode, it’s all at the end.

13.2.4 Read mode

- If you want to read a file:
  - Open it with mode argument as “r”.
  - You can not write to this file.

```c
pFile = fopen("myfile.txt","r");
```

- This positions the file to the beginning of the data.
- If you are going to read the file:
  - If must already exist, there is no point on reading an empty file.
- If you try to open a file for reading and it doesn’t exist, fopen() will return a NULL pointer.
- You always want to check the value returned from fopen().

13.2.5 Renaming a file

- Renaming a file is very easy.
  - Use the rename() function.
  - It takes two parameters, pointers of the old name and new name.

```c
int rename(const char *oldname, const har *newname);
```

- The integer that is returned will be 0 if the name change was successful and nonzero otherwise.
- The file must not be opened when you call rename(), otherwise the operation will fail.

```c
if (rename("C:\\temp\\myfile.txt","C:\\temp\\myfile_copy.txt")){
    printf("Failed to rename file.");
} else {
    printf("File renamed successfully.");
}
```

  - You know it's a Windows directory because of the `C:\\`.
  - Also in order to provide \ in C, you must provide two, this is because the backslash is used for special escape sequences, thus you need to escape the backslash itself with another one as so: \\

- This will change the name of myfile.txt in the temp directory on drive C to myfile_copy.txt.

- If the file path is incorrect or the file does not exist, the renaming operation will fail.

13.2.6 Closing a file

- When you have finished with a file, you need to tell the operating system so that it can free up the file:
  - You can use it by the fclose() function.
  - Remember there is a limit to how many files you can have opened at a time.

- fclose() accepts a file pointer as an argument.
  - Returns EOF (int) of an error occurs.
    - EOF is a special character called the end-of-file character.
    - Defined in stdio.h as a negative integer that is usually equivalent to the value -1.
  - 0 if successful.

```
fclose(pfile); // Close the file associated with pfile.
pfile = NULL;
```

- The result of calling fclose() is that the connection between the pointer, pfile, and the physical file is broken.
  - pfile can no longer be used to access the file.

- If the file was being written, the current contents of the output buffer are written to the file to ensure that the data is not lost. So you don't have to worry about your data, it will be written even if you call the fclose() while it's being written, it will write it first and then close it.
  - This is called thread safe, meaning multiple points of execution in your program.
  - This process is thread safe because even if another thread is writing your file, fclose() will not close it until the other thread is finished.

- It is good programming practice to close a file as soon as you have finished with it.
  - This protects against output data loss.

- You must also close a file before attempting to rename it or remove it.

13.2.7 Deleting a file

- You can delete a file by invoking the remove() function:
  - Declared in stdio.h
  - Just like all the other ones.

```
remove("myfile.txt");
```

- Will delete the file that has the name myfile.txt from the current directory.

- The file must be closed in order for this operation to work.

- The file cannot be open when you try to delete it, just like rename.
- You should always double check with operations that delete files.
  - You could wreck your system if you do not.
  - Deleting files can cause problems so be careful.
- Everything is based on that file pointer.

13.3 Reading for a file

13.3.1 Reading characters from a text file

- The fgetc() function reads a character from a text file that has been opened for reading.
  - We need to open the file first using the "r" or "r+" mode.
- Takes a file pointer as its only argument and returns the character read as type int.

```c
int mchar = fgetc(pfile); // Reads a character into mchar with pfile a File pointer
```

- The mchar is type int because EOF will be returned if the end of the file has been reached.
- The function getc(), which is equivalent to fgetc(), is also available.
  - Requires an argument of type FILE* and returns the character read as type int.
  - Virtually identical to fgetc().
  - Only difference between them is that getc() may be implemented as a macro, whereas fgetc() is a function.
  - fgetc() has more protection in it.
- You can read the contents of a file again when necessary.
  - The rewind() function positions the file that is specified by the file pointer argument at the begining.

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE *fp;
    int c;

    fp = fopen("file.txt","r");
    if (fp == NULL){
        perror("Error in opening file.");
        return (-1);
    }

    // read a single char
    while ((c = fgetc(fp)) != EOF){
        printf("%c",c);
    }

    fclose(fp);

    fp = NULL;

    return 0;
}
/* Output:
hello
from
a
file
:)

*/


```

13.3.2 Reading a string from a text file

- You can use the fgets() function to read from any file or stream.

```c
char *fgets(char *str, int nchars, FILE *stream);
```

- The function reads a string into memory area pointed by str, from the file specified by the stream.
  - Characters are read until either a `'\n'` is read or nchars-1 characters have been read from the stream, whichever occurs first.
  - I a newline character is read, it's retained in the string.
    * a `\0` character will be appended to the end of the string.
  - If there is no error, fgets() returns the pointer, str.
  - If there is an error, NULL is returned.
  - Reading EOF causes NULL to be returned.

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE *fp;
    int c;
    char str[10];

    fp = fopen("file.txt","r");

    if (fp == NULL){
        perror("Error in opening file.");
        return (-1);
    }

    if (fgets(str,10,fp) != NULL){
        /* writing content to stdout */
        printf("%s",str);
    }
    return 0;
}
```

```
/* Output:
hello

*/
```

13.3.3 Reading formatted input from a file

- You can get formatted input from a file by using the standard fscanf() function.

```
int fscanf(FILE *stream, const char *format, ...);
```

  - This is formatted input, formatted input is data that follows some convention such as character delimiters.

- The first argument to this function is the pointer to a FILE object that identifies the stream.
- The second argument to this function is the format:
  - A C string that contains one or more of the following items.
    - White space character.
    - Non-white spate character.
    - Format specifiers.
    - Usage is similar to scanf, but, from a file.
- The function returns the number of input items successfully matched and assigned.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char str1[10], str2[10], str3[10];
    int year;
    FILE *fp;

    fp = fopen("ffile.txt","w+");
    if (fp != NULL){
        fputs("Hello how are 2020",fp);
    }

    rewind(fp);

    fscanf(fp,"%s %s %s %d", str1, str2, str3, &year);

    printf("Read string 1: |%s|\n",str1);
    printf("Read string 1: |%s|\n",str2);
    printf("Read string 1: |%s|\n",str3);
    printf("Read integer 1: |%d|\n",year);

    fclose(fp);

    return 0;
}
/* Output:
Read string 1: |Hello|
Read string 1: |how|
Read string 1: |are|
Read integer 1: |2020|

*/
```

13.4 Writing to a file

- The simplest write operations is provided by the function fputc().
  - Writes a single character to a text file.

```
int fputc(int ch, FILE *pfile);
```

- The function writes the character specified by the first argument to the file identified by the second argument (file pointer)
  - Returns the character that was written if successful.
  - Return EOF of failure.
- In practice, characters are not usually written to a physical file one by one:
  - This is extremely ineficient.
- The potc() function is equivalent to fputc():
  - Requires the same arguments and the return type is the same.
  - Difference between them is that putc() may be implemented in the standard library as a macro, whereas fputc() is a function.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE *fp;
    int ch;

    fp = fopen("file.txt","w+");

    for (ch = 33; ch <= 100; ch ++){
        fputc(ch,fp); // writes a single ascii char
    } // you can use the atoi function to write other data types as well.

    fclose(fp);

    fp = NULL;

    return 0;
}
/* Output: in file
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcd
*/
```

13.4.1 Writing a string to a file

- You can use the fputs() function to write to any file stream, it similar to the fgets().

```c
int fputs(const char *str, FILE *pfile);
```

- The first argument is a pointer to the character string that is to be written to the file.
- The second argument is the file pointer.
- This function will write characters from a string until it reaches the `\0` character.
  - Does not write the null terminator character to the file.
    - Can complicate reading back variable-lenght strings from a file that have been written by fputs().
    - Expecting to write a line of text that has a newline character at the end.
    - You need to explicitly add the null terminator to the file.
- This function tends to be more eficient than fputc() in writing strings.

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE *filePointer;

    filePointer = fopen("file.txt","w+");

    // write two lines.
    fputs("This is a string.",filePointer);
    fputc('\n',filePointer);
    fputs("Great",filePointer);
    fputc('\n',filePointer);

    fclose(filePointer);
    return 0;
}
/* Output: in file
This is a string.
Great

*/
```

13.4.2 Writing formatted output to a file

- The standard function for formatted output to a stream is fprint().

```c
int fprintf(FILE *stream, const char *format, ...);
```

- The first argument to this function is the pointer to a FILE object that identifies the stream.
- The second argument to this function is the format:
  - A C string that contains one or more of the following items.
    - White space character.

  * Non-white space character.
  * Format specifiers.
  * Usage is similar to printf, but to a file.

- If successful, the total number of characters written is returned otherwise, a negative number is returned.

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE *fp;

    fp = fopen("file.txt","w+");

    if (fp == NULL){
        printf("Error");
        exit(-1);
    }


    fprintf(fp, "%s %s %s %s %d", "Hello", "My", "Number", "is", 502);

    fclose(fp);

    fp = NULL;

    return 0;
}
/* Output: in file
Hello My Number is 502
*/
```

13.5 Finding your position in a file

- File positioning, for many applications, you need to access data in a file other than sequential order (meaning starting in the beginning and ending at the end, byte by byte).
  - You might need to go to the middle of the file for example.
- There are various functions that you can use to access data in random sequence.
- There are two aspects to file positioning:
  1. Finding out where you are in a file.
  2. Moving to a given point in a file.
- You can access a file at a random position regardless whether you opened the file.

13.5.1 Finding out where you are, ftell()

- You have two functions to tell you where you are in a file.
  - ftell()
  - fgetpos()

```c
long ftell(File *pfile);
```

- This function accepts a file pointer as an argument and returns a long integer value that specifies the current position in the file.

```c
long fpos = ftell(pfile);
```

- The fpos variable now holds the current position in the file and you can use this to return to this position at any subsequent time.
  - Value is the offset in bytes from the beginning of the file.
  - Basically how many bytes from the beginning of the file.

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE *fp = NULL;
    int len;
    fp = fopen("file.txt","r");

    if (fp == NULL){
        perror("Error opening file.");
        return (-1);
    }

    fseek(fp, 0, SEEK_END);

    len = ftell(fp);

    fclose(fp);

    fp = NULL;

    printf("Total size of file.txt = %d bytes",len);

    return 0;
}
/* Output: in file
Total size of file.txt = 24 bytes
*/
```

13.5.2 fgetpos()

- The second function providing information on the current file position is a little more complicated.

```c
int fgetpos(FILE *pfile, fpos_t * position);
```

- The first parameter is a file pointer.
- The second parameter is a pointer to a type that is defined in stdio.h:
  - fpos_t: a type that is able to record every position within a file.

- The fgetpos() function is designed to be used with the positioning function fsetpos().
- The fgetpos() function stores the current position and file state information for the file in position and returns 0 if the operation is successful.
  - Returns a non-zero integer value for failure.

```c
fpos_t here;
fgetpos(pfile,&here);
```

- The above records the current file position in the variable here.
- You must declare a variable of type fpos_t.
  - You cannot declare a pointer of type fpos_t because there will not be any memory allocated to store the position data.

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE *fp = NULL;
    fpos_t position;

    fp = fopen("file.txt","w+");
    fgetpos(fp, &position);
    fputs("Hello, World!",fp);

    fclose(fp);

    return 0;
}
/* Output: in file
Hello, World!
*/
```

13.5.3 Setting a position in a file

- As a complement to ftell(), you also have fseek() function.

```c
int fseek(FILE *pfile, long offset, int origin);
```

- The first parameter is a pointer to the file you are repositioning.
- The second and third parameter define where you want to go in the file.
  - The second parameter is an offset from a reference point specified by the third parameter.
  - Reference point can be one of three values that are specified by the predefined names.
    - SEEK_SET: defines the beggining of a file.
    - SEEK_CUR: defines the current position in the file.
    - SEEK_END: defines the end of the file.
- For a text mode file, the second argument must be a value returned by ftell().
- The third argument for text mode files must be SEEK_SET.

  - For text files, all operations with fseek() are performed with reference to the beggining of the file.
  - For binary files, the offset argument is simple a relative byte count.
    * Can therefore supply positive or negative values for the offset when the reference point is specified as SEEK_CUR

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE *fp = NULL;

    fp = fopen("data.txt","r+");

    fseek(fp,7,SEEK_SET);
    fputs("@",fp);


    fclose(fp);
    return 0;
}
/* Input: in file
012345678910 <- insert on 7 a @
*/
/* Output:
0123456@8910 <- insert on 7 a @
*/


```

13.5.4 fsetpos()

- You have the fsetpos() function to go with fgetpos().

```c
int fsetpos(FILE *pfile, const fpos_t * position);
```

- The first parameter is a pointer to the open file.
- The second is a pointer of the fpos_t type.
  - The position that is stored at the address was obtained by calling fgetpos().

```c
fsetpos(pfile, &here);
```

- The variable here was previously set by a call to fgetpos().
- The fsetpos() returns a non-zero value on error or0 when it succeeds.
- This function is designed to work with a value that is returned by fgetpos():
  - You can only use it to get to a place in a file that you have been before.
  - fseek() allows you to do to any position just by specifying the appropriate offset.

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE *fp = NULL;
    fpos_t position;

    fp = fopen("file.txt","w+");
    fgetpos(fp, &position);

    fputs("Hello, World!",fp);

    fsetpos(fp, &position);

    fputs("This is going to override previous content",fp);
    fclose(fp);

    fp = NULL;
    return 0;
}
/* Output: in file
This is going to override previous content
*/
```

13.5.5 Interesting insights

- Whenever you seek for a byte using fseek() using a negative number such as `fseek(fp,-10,SEEK_END);` you are really referring to the following procedure.

  $$\text{byte in question (second param)} = \text{ total number of bytes } + (\text{ specified number })$$

  Thus in the above example we would have: (assuming 812 bytes as total number of bytes)

  $$\begin{aligned}\text{byte in question (second param)} &= 812 + (-10)\\ &= 802\end{aligned}$$

13.6 Challenge Find the number of lines in a file

```c
#include <stdio.h>
#include <stdlib.h>

#define FILENAME "lines.txt"

int main()
{
    FILE *fp = NULL;
    char ch;
    int lines = 0;

    fp = fopen(FILENAME,"r");

    if (fp == NULL){
        perror("Error opening file.");
        exit(-1);
    }

    while ((ch=fgetc(fp)) != EOF){
        if (ch == '\n')
            lines++;
    }

    fclose(fp);

    fp = NULL;

    printf("Total number of lines are: %d",lines);

    return 0;
}
/* Input: in file
Lorem ipsum dolor sit amet, consectetur adipiscing
elit, sed do eiusmod tempor incididunt ut labore et
dolore magna aliqua. Molestie at elementum eu
facilisis sed odio morbi quis commodo.
Egestas congue quisque egestas diam in arcu cursus
euismod quis. Quis enim lobortis scelerisque
fermentum dui faucibus in ornare. At elementum
eu facilisis sed odio. Amet mauris commodo quis
imperdiet massa tincidunt. Ultricies tristique
nulla aliquet enim tortor at. Bibendum ut tristique
et egestas quis. Donec enim diam vulputate ut.
Fames ac turpis egestas maecenas pharetra convallis
posuere morbi. Consectetur adipiscing elit ut
aliquam. Ut sem viverra aliquet eget sit amet
tellus cras. Cras pulvinar mattis nunc sed. Viverra
suspendisse potenti nullam ac tortor vitae purus
faucibus.
*/
/* Output: in file
Total number of lines are: 17
*/
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#define FILENAME "upper.txt"
#define TEMPORARY "temp.txt"

int main()
{
    FILE *fp = NULL, *fptemp = NULL;
    char ch;

    // Reading in information from file
    fp = fopen(FILENAME,"r");
    if (fp == NULL){
        perror("Error on reading file: ");
        printf("%s",FILENAME);
        exit(-1);
    }

    // Creation of temporary file.
    fptemp = fopen(TEMPORARY,"w+");

    while ((ch=fgetc(fp)) != EOF){
        if (islower(ch))
            ch = toupper(ch);
        fputc(ch,fptemp);
    }

    fclose(fp);
    fclose(fptemp);

    // Remove the FILENAME
    remove(FILENAME);

    // Rename temp file to FILENAME
    rename(TEMPORARY,FILENAME);

    // Remove the temp file.
    remove(TEMPORARY);

    fp = fopen(FILENAME,"r");

    while ((ch=fgetc(fp)) != EOF){
        printf("%c",ch);
    }

    return 0;
}
/* Input: in file
this is lowecase
THIS IS UPPERCASE?
is everything uppercase?

*/
/* Output:
THIS IS LOWECASE
THIS IS UPPERCASE?
IS EVERYTHING UPPERCASE?

*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
```

```c

#define FILENAME "lines.txt"

int main()
{
    FILE *fp = NULL;
    char ch;
    int i = 0, end = 0;

    fp = fopen(FILENAME,"r");

    if (fp == NULL){
        perror("Error opening: ");
        printf("%s",FILENAME);
    }

    // move the file pointer to the end
    fseek(fp,0,SEEK_END);

    // figure out the complete byte size
    end = ftell(fp);

    // print the reverse
    while (i < end){
        i++;
        fseek(fp,-i,SEEK_END);
        printf("%c",fgetc(fp));
    }

    // Close The file
    fclose(fp);
    fp = NULL;

    return 0;
}
/* Input:
Lorem ipsum dolor sit amet, consectetur adipiscing
elit, sed do eiusmod tempor incididunt ut labore et
dolore magna aliqua. Molestie at elementum eu
facilisis sed odio morbi quis commodo.
Egestas congue quisque egestas diam in arcu cursus
euismod quis. Quis enim lobortis scelerisque
fermentum dui faucibus in ornare. At elementum
eu facilisis sed odio. Amet mauris commodo quis
imperdiet massa tincidunt. Ultricies tristique
nulla aliquet enim tortor at. Bibendum ut tristique
et egestas quis. Donec enim diam vulputate ut.
Fames ac turpis egestas maecenas pharetra convallis
posuere morbi. Consectetur adipiscing elit ut
aliquam. Ut sem viverra aliquet eget sit amet
tellus cras. Cras pulvinar mattis nunc sed. Viverra
suspendisse potenti nullam ac tortor vitae purus
faucibus.

```

```
*/
/* Output:

.subicuaf
surup eativ rotrot ca mallun itnetop essidnepsus
arreviV .des cnun sittam ranivlup sarC .sarc sullet
tema tis tege teuqila arreviv mes tU .mauqila
tu tile gnicsipida rutetcesnoC .ibrom ereusop
sillavnoc arterahp saneceam satsege siprut ca semaF
.tu etatupluv maid mine cenoD .siuq satsege te
euqitsirt tu mudnebiB .ta rotrot mine teuqila allun
euqitsirt seicirtlU .tnudicnit assam teidrepmi
siuq odommoc siruam temA .oido des sisilicaf ue
mutnemele tA .eranro ni subicuaf iud mutnemref
euqsirelecs sitrobol mine siuQ .siuq domsiue
susruc ucra ni maid satsege euqsiuq eugnoc satsegE
.odommoc siuq ibrom oido des sisilicaf
ue mutnemele ta eitseloM .auqila angam erolod
te erobal tu tnudidicni ropmet domsuie od des ,tile
gnicsipida rutetcesnoc ,tema tis rolod muspi meroL
*/
```

Chapter 14

The standard C library

14.1 Standard header files

- The C standard library is basically all the functionality of the C programming language.

14.1.1 strdef.h

Define	Meaning
`NULL`	A null pointer constant.
`offsetof` (structure member)	The offset in bytes of the member *member* from the start of the structure *structure*; the typeof the result is *size_t*
`ptrdiff_t`	The type of integer produced by subtracting two pointers.
`size_t`	The type of integer produced by the *sizeof* operator.
`wchar_t`	The type if the integer required to hold a wide character.

Taken from Programming in C, Kochan

14.1.2 limits.h

- `<limits.h>`: contains various implementation-defined limits for character and integer data types.

Define	Meaning
CHAR_BIT	Number of bits in a `char` (8).
CHAR_MAX	Maximum value for object of type `char` (127 if sign extension is done n chars, 255 otherwise).
CHAR_MIN	Minimum value for object of type `char` (-127 if sign extension is done on chars, 0 otherwise).
SCHAR_MAX	Maximum value for object of type `signed char` (127).
SCHAR_MIN	Minimum value for object of type `signed char` (-127).
UCHAR_MAX	Maximum value for object of type `unsigned char` (255).
SHRT_MAX	Maximum value for object of type `short int` (32767).
SHRT_MIN	Minimum value for object of type `short int` (-32767).
USHRT_MAX	Maximum value for object of type `unsigned short int` (65535).
INT_MAX	Maximum value for object of type `int` (32767).
INT_MIN	Minimum value for object of type `int` (-32767).
UINT_MAX	Maximum value for object of type unsined int (65535).
LONG_MAX	Maximum value for object of type `long int` (2,147,483,647).
LONG_MIN	Minimum value for object of type `long int` (-2,147,483,647).
ULONG_MAX	Maximum value for object of type `unsigned long int` (4,294,967,295).
LLONG_MAX	Maximum value for object of type `long long int` (9,223,372,036,854,775,807).
LLONG_MIN	Minimum value for object of type `long long int` (9,223,372,036,854,775,807).
ULLONG_MAX	Maximum value for object of type `unsigned long long int` (18,446,744,073,709,551,615).

Taken from Programming in C, Kochan

14.1.3 stdbool.h

- `<stdbool.h>`: file contains definitions for working with boolean variables (type _Bool).

Define	Meaning
`bool`	Substitute name for the basic `_Bool` data type.
`true`	Defined as `1`.
`false`	Defined as `0`.

14.2 Various functions

- Reminder of some various functions.

14.2.1 String function

- To use any of these, you need to include the header file `<string.h>` .
- `char *strcat(s1,s1)`
  - Concatenates the character string s2 to the end of s1, placing a null character at the end of the final string. The function returns s1.
- `char *strchr(s,c)`
  - Searches the string s for the first occurrence of the character c, If it is found, a pointer to the character is returned; otherwise, a null pointer is returned.
- `int strcmp(s1, s1)`

  - Compares strings s1 and s2 and returns a value less than zero if s1 is less than s2, equal to zero if s1 is equal to s2, and greater than zero if s1 is greater than s2.
  - Greater means in alphabetic precedence of the ascii table, a is 97 and b is 98, if we were to send a and b to this function it will return a negative because a is less than b.

- `char *strcpy(s1, s2)`
  - Copies the string s2 to s1, returning s1.
  - Integers and doubles don't need a function to reference them, strings do.
- `size_t strlen(s)`
  - Returns the number of characters in s, excluding the null character `\0`.
- `char *strncat(s1, s2, n)`
  - Copies s2 to the end of s1 until either the null character is reached or n characters have been copied, whichever occurs first. Returns s1.
  - Safer for buffer overflows.
- `int strncmp(s1, s2, n)`
  - Performs the same function as strcmp, except that at most n characters from strings are compared.
  - Safer for buffer overflows.
- `char *strncpy(s1, s2, n)`
  - Copies s2 to s1 until the null character is reached or n characters have been copied, whichever occurs first. Returns s1.
  - Safer for buffer overflows.
- `char *strrchr(s, c)`
  - Searches the string s for the last occurrence of the character c. If found, a pointer to the character in s is returned; otherwise, the null pointer is returned.
  - Safer for buffer overflows.
- `char *strstr(s1, s2)`
  - Searches the string s1 for the first occurrence of the string s2. If found a pointer to the tart of where s2 is located inside s1 is returned; otherwise, if s2 is not located inside s1, the null pointer is returned.
- `char *strtok(s1, s2)`
  - Breaks the string s1 into tokens based on delimiter characters in s2.
  - Helps with parsing data.

14.2.2 Character functions

s

- To use these character fucntions, you must include the file `<ctype.h>`
- The character functions we used were: we used to check or test a character, there are also convention functions.
  - isalnum

    - isalpha
    - isblank
    - iscntrl
    - isdigit
    - isgraph
    - islower
    - isspace
    - ispunct
    - isupper
    - isxdigit

- `int tolower(c)`
    - Returns the lowercase equivalent of c. If c is not an uppercase letter, c itself is returned.
- `int toupper(c)`
    - Returns the uppercase equivalent of c. If c is not a lowercase letter, c itself was returned.

I/O functions

- To use the most common I/O functions from the C library you should include the header file `<stdio.h>` .
- Included in this file are declarations for the I/O functions and definitions for the names EOF, NULL, stdin, stdout, stderr (all constant values), and FILE.
- `int fclose(filePtr)`
    - Closes the file identified by filePtr and returns zero if the close was successful, or returns EOF if an error occurs.
- `int feof(filePtr)`
    - Returns nonzero if the identified file has reached the end of the file and returns zero otherwise.
    - We can use this instead of comparing to a constant EOF.
- `int fflush(filePtr)`
    - fflushes (writes) any data from internal buffers to the indicated file, returning zero on success and the value EOF if an error occurs.
- `int fgetc(filePtr)`
    - Returns the next character from the file identified by the filePtr, or the value of EOF if an end-of-file condition occurs.
    - Remember that this function returns an int.
- `int fgetpos(filePtr, fpos)`
    - Gets the current position for the file associated wirh filePtr, storing it into fpor_t (defined in `<stdio.h>`) variable pointed to by fpos. fgetpos returns zero on success, and returns nonzero on failure.
- `char *fgets(buffer, i, filePtr)`

  - Reads characters from the indicated file, until either $i - 1$ characeters are read or a newline character is read, whichever occurs first.

- `FILE *fopen(fileName, accessMode)`
  - Opens the specified file with the indicated access mode.
- `int fprintf(filePtr, format, ...)`
  - Writes the specified arguments to the file identified by filePtr, according to the format specified by the character string format.
- `int fputc(c,filePtr)`
  - Writes the value of c to the file identified by the filePtr, returning c if the write is successful, and the value EOF otherwise.
- `int fputs(buffer, filePtr)`
  - Writes the characters in the array pointers to by buffer to the indicated file until the terminating null character in buffer is reached.
- `int fscanf(filePtr, format, ...)`
  - Reads data items from the file identified by filePtr, according to the format specified by the character string format.
- `int fseek(filePtr, offset, mode)`
  - Positions the indicated file to a point that is offset (a long int) bytes from the beginning of the file, from the current position in the file, or from the end of the file, depending upon the value of mode (an integer.)
  - Allows you to move to a position.
- `int fsetpos(filePtr, fpos)`
  - Sets the current file position for the file associated with filePtr to the value pointed to by fpos, which is of type fpos_t (defined in `<stdio.h>`). Returns zero on succes, and non-zero on failure.
- `lont ftell (filePtr)`
  - Returns the relative offset in bytes of the current position in the file identified by filePtr, or -1L on error.
- `int printf(format, ...)`
  - Writes the specified arguments to stdout, according tot he format specified by the character string.
  - Returns the number of characters written.
- `int remove(fileNam)`
  - Removes the specified file. A non-zero value is returned on failure.
  - The file pointer has to be closed.
- `int rename(fileName1, fileName2)`
  - Renames the file fileName1 to fileName2, returning a non-zero result on failure.
- `int scanf(format, ...)`
  - Reads items from stdin according to the format specified by the string format.

14.2.3 Conversion functions

- To use these functions that convert character strings to numbers you must include the header file `<stdlib.h>`, look at the stdlib header file for more functions.
- `double atof(s)`
  - Converts the string pointed to by s into an int, returning the result.
- `int atoi(s)`
  - Converts the string pointed to by s into an int, returning the result.
- `int atol(s)`
  - Converts the string pointed to by s into a long int, returning the result.
- `int atoll(s)`
  - Converts the string pointed to by s into a long long int, returning the result.

14.2.4 Dynamic memory allocation function

- To use these functions that allocate and free memory dynamically you must include the `<stdlib.h>` header file.
- `void *calloc(n,size)`
  - Allocates contiguous space for n items of data, where each item is size bytes in length. The allocated space is initially set to all zeroes. On success, a pointer to the allocated space is returned; on failure, the null pointer is returned.
  - This is preferred over malloc(). It initializes it as well as allocating it.
- `void free(pointer)`
  - Returns a block of memory pointed to by pointer that was previously allocated by a calloc(), malloc(), or realloc() call.
- `void *malloc(size)`
  - Allocates contiguous space of size bytes, returning a pointer to the beginning of the allocated block if successful, and the null pointer otherwise.
- `void *realloc(pointer, size)`
  - Changes the size of a previously allocated block to size bytes, returning a pointer to the new block (which might have moved), or null pointer of an error occurs.

14.3 Math functions

- To use common math functions you must include the `<math.h>` header file and link to the math library.
- A lot of different functions here are the main ones, consult the actual header file for more.
- `double acosh(x)`
  - Returns the hyperbolic arc cosine of x, $x \geq 1$
- `double asin(x)`

  - Returns the arcsine of x as an angle expressed in radians in the range $[-\pi/2, \pi/2]$. x is in the range $[-1, 1]$

- `double atan(x)`
  - Returns the arctangent of x as an angle expressed in radians in the range $[-\pi/2, \pi/2]$
- `double ceil(x)`
  - Returns the smallest integer value grater than or equal to x. Note that the value is returned as a double.
- `double cos(r)`
  - Returns the cosine of r.
- `double floor(x)`
  - Returns the largest integer value less than or equal to x. Note that the value is returned as double.
- `double log(x)`
  - Returns the natural logarithm of x, $x \geq 0$
- `double nan`
  - Returns a NaN, if possible, according to the content specified by the string pointed to by s.
- `double pow(x,y)`
  - Returns x,y. If x is less than zero, y must be an integer. If x is equal to zero, y must be greater than zero.
- `double remainder(x,y)`
  - Returns the remainder of x divided by y.
  - You can use this instead of the % operator.
- `double round(x)`
  - Returns the value of x rounded to the nearest integer in floating-point format. Halfway values are always rounded away from zero (so 0.5 always rounds to 1.0).
- `double sin(r)`
  - Returns the sine of r.
- `double sqrt(x)`
  - Returns the square root of x, $x \geq 0$.
- `double tan(r)`
  - Returns the tangent of r.
- And so many more ...
  - Check out the complex arithmetic library.

14.4 Utility functions

- To use these routines, include the header file `<stdlib.h>`
- Here are some of the functions.
- `int abs(n)`
  - Returns the absolute value of its argument n.
- `void exit(n)`
  - Terminates program execution, closing any returning the exit status specified by its int argument n.
  - `EXIT_SUCCESS` and `EXIT_FAILURES`, defined in `<stdio.h>`.
  - Other related routines in the library that you might want to refer to are abort and atexit.
    - abort is usually invoked when you have a fatal error, the good thing about abort is that it will generate a core file and this is useful for debugging.
- `char *getenv(s)`
  - Returns a pointer to the value of the environment variable pointed to by s, or a null pointer if the variable doesn't exist.
    - There is a get environment variable, an environment variable is like a global variable for the operating system.
    - You can set an environment variable and then use it in you program.
  - Used to get environment variables.
- `void qsort(arr, n, size, comp_fn)`
  - Sort the data array pointed to by the void pointer arr.
  - There are n elements in the array, each size bytes in length. n and size are of type size_t.
  - The fourth argument is of type "pointer to function that returns int and that takes two void pointers as arguments."
  - qsort calls this function whenever it needs to compare two elements int the array, passing it pointer to the elements to compare.
- `int rand(void)`
  - Returns a random number in the range `[',RAND_MAX]`, where `RAND_MAX` is defined in `<stdlib.h>` and has a minimum value of 32767.
- `void srand(seed)`
  - Seed the random number generator to the unsigned int value seed.
- `int system(s)`
  - Gives the command contained in the character array pointed to by s to the system for execution, returning a system-defined value.
  - `system("mkdir /usr/tmp/data);`

14.4.1 Assert library

- The assert library, supported by the `<assert.h>` header file, is a small one designed to help with debugging.
- It consists of a macro named assert()
  - It takes as its argument an integer expression.
  - If the expression evaluates as false (non-zero), the assert() macro writes an error message to the standard error stream (stderr) and calls the abort() function, which terminates the program.
  - This is a valuable function from debugging.

```
z = x * x - y * y; // should be greater + */
assert(z >= 0); // asserts that z is greater or equal to 0
```

14.4.2 Other useful header files

- time.h: defines macros and functions supporting operations with dates and times.
- errno.h: defines macros for the reporting of errors.
- locale.h: defines functions and macros to assist with formatting data such as monetary units for different countries.
- signal.h: defines facilities with conditions that arise during a program execution, including error conditions.
- stdarg.h: defines facilities that enable a variable number of arguments to be passed to a function.

Chapter 15

Conclusion

15.1 Further topics to study

- More on data types:
  - Defining your own types (typedef). Create your own data types.
- More on the preprocessor:
  - String concatenation.
  - You can optimize a lot of your code.
- More on void*'s:
  - A lot more.
- Static libraries and shared objects.
  - Reuse code.
- macros
  - Some of the functions we used are macros.
- unions:
  - Used in data structures.
- Function pointers.
- Advanced pointers.
  - Pointers to pointers.
- Variable arguments to functions (variadic functions).
- Dynamic linking (dlm_open).
  - Open libraries on run-time.
- Signals, forking and iner-process communications.
  - Signals it's when two processes want to communicate, forking is creating another executing thread.
- Threading and concurrency:
  - Asynchronous execution.

- Sockets:
  - Data from computer to computer.
- setjmp and longjmp for restoring state.
- More on memory management and fragmentation.
  - Avoid fragmentation.
- More and making your program portable.
- Interfacing with kernel modules (drivers and ioctls).
  - Create drivers for example.
- More on compiler and linker flags.
- Advanced use of gdb:
  - Debugging and more insight while using it.
  - Understanding or interpreting the core files.
- Profiling and tracing tools (gprof, dtrace, strace):
  - Checking for dead code, tracing and other things.
- Memory debugging tools such as valgrind.
  - Find memory links and other memory related errors.

15.2 Course summary

- Code Blocks: IDE.
- Basic operators: logical, arithmetic and assignment.
- Conditional statements: making decisions (if, switch).
- Repeating code: looping (for, while, do-while).
- Arrays: defining and initializing, multidimensional.
- Functions: declaration and use, arguments and parameters, call by value vs. call by reference.
- Debugging: call stack, common mistakes, understanding compiler messages.
- Structs: initializing, nested structures.
- Character strings: basics, arrays of chars, character operations.
- Pointers: definition and use, using with functions and arrays, malloc, pointer arithmetic.
- The preprocessor: #define, #include.
- Input and output: command line arguments scanf, printf.
- File Input/Output: reading and writing to a file, fgetc, fgets, fputc, fgetc, fseek, etc.
- Standard C library: string functions, math functions, utility functions, standard header files.

15.2.1 Course outcomes

- You are now able to write beginner C programs.
- You are now able to write efficient C programs.
  - Modular.
  - Low coupling.
  - Naming conventions, indentation.
- You are now able to find and fix errors:
  - You now understand compiler messages.
  - You know how to use a debugger.
- You now understand fundamental aspects of the C programming language.