Contents

[Fundamental Concepts 3](#_Toc525746509)

[What Are Microservices 3](#_Toc525746510)

[5 secrets to make you a better coder 3](#_Toc525746511)

[Programming Language C 5](#_Toc525746512)

[Drawing a Shape 5](#_Toc525746513)

[We can use printf to draw a triangle on the console: 5](#_Toc525746514)

[Variables and types 5](#_Toc525746515)

[Data Types 5](#_Toc525746516)

[Defining Variables 5](#_Toc525746517)

[Arrays 6](#_Toc525746518)

[Strings 8](#_Toc525746519)

[Defining Strings 8](#_Toc525746520)

[String Length with strlen() 8](#_Toc525746521)

[String Comparison with strncmp() 8](#_Toc525746522)

[String Concatenation with strncat 9](#_Toc525746523)

[For Loops 9](#_Toc525746524)

[Getting User Input 9](#_Toc525746525)

[While Loops 10](#_Toc525746526)

[Loop Directives (break & continue) 10](#_Toc525746527)

[There are two important loop directives that are used in conjunction with all loop types in C – the break and continue directives. 10](#_Toc525746528)

[Functions 10](#_Toc525746529)

[Exercise 12](#_Toc525746530)

[Accessing Files 12](#_Toc525746531)

[Static 12](#_Toc525746532)

[What is a static variable? 12](#_Toc525746533)

[What is a static function? 13](#_Toc525746534)

[Static vs Global? 13](#_Toc525746535)

[Exercise 14](#_Toc525746536)

[Pointers 14](#_Toc525746537)

[What is a pointer? 14](#_Toc525746538)

[Memory Address 14](#_Toc525746539)

[Strings as pointers 14](#_Toc525746540)

[Dereferencing 15](#_Toc525746541)

[Exercise 15](#_Toc525746542)

[Structures 17](#_Toc525746543)

[Typedefs 17](#_Toc525746544)

[Exercise 18](#_Toc525746545)

[Function Arguments by Reference 18](#_Toc525746546)

[Pointers to structures 19](#_Toc525746547)

[Exercise 20](#_Toc525746548)

[Dynamic Allocation 20](#_Toc525746549)

[Exercise 21](#_Toc525746550)

[Arrays and Pointers 22](#_Toc525746551)

[Dynamic Memory Allocation for Arrays 23](#_Toc525746552)

[Recursion 24](#_Toc525746553)

[Exercise 25](#_Toc525746554)

[Linked Lists 25](#_Toc525746555)

[Introduction 25](#_Toc525746556)

[What is a Linked List? 26](#_Toc525746557)

[Iterating over a list 27](#_Toc525746558)

[Adding an item to the end of the list 27](#_Toc525746559)

[Adding an item to the beginning of the list (pushing to the list) 28](#_Toc525746560)

# Fundamental Concepts

## What Are Microservices

Microservice architecture is a pattern of organizing computer systems into services that can scale in demand. Back in 1990s, an internet would run a big monolithic program, on a server that the company maintain on premise. To serve an increase in traffic, a popular company would simply add more instances of the monolithic.

Monolithic architectures do have some positive features, a monolithic centralizes the code base, so it is in one place. Engineers can step through any part of the code when they are debugging. Also, user requests that are completely served by a monolithic, do not have to make many calls across network, which reduces the chance of network failures.

Most software companies have their code in a monolithic today, when those monolithic gets big, problems can start to occur. Centralized codes lead to tight coupling that is hard to breakup, if the program is too big, it will be impossible to run on a typical machine. Internet giants in the early 2000s began breaking up their application into service, instead of scaling monolithic application, a service oriented architecture could scale the parts of the application that were underload.

Operating system virtualization made service oriented architecture more economical. One server could host multiple virtualized operating system instances, and each of those instances could run a service. But this also meant that engineers had to handle more and more layers of infrastructure, the virtual machine host, the hyperviser layer, and the hardware itself. Failures become more complex, debugging got harder.

In 2006, Amazon web services launched the Oblastic Compute Cloud, EC2 allows programmers to rent virtual machines in Amazon’s data center. With the Amazon taking care of the failure on the hardware level, and the hyperviser level, programmers can focus on the virtual machine hosts themselves, where their application codes are running. But using an entire virtual machine to run a small piece of application code is wasteful. Containers allow a virtual machine to be sliced up into isolate file system regions. A container can be as large as the entire VM, or as small as your smallest service, hence the term microservice.

Microservices run in container, which run in a virtual machine, which run on a Hyperviser, which runs on a server, which sits in a rack, which sits in a data center, which is part of a network of data centers called the cloud. Containerized architecture lead to new problem, companies that ran thousands of microservices and containers on the cloud did not have a simple way of managing them. Kubernetes is an open source project from Google, that gives engineers a centralized system to for managing containers. Kubernetes also makes those service portable, creating a competitive tension between Amazon Web Services, and Google cloud Platform. Both of which can hold Kubernetes clusters.

The days of microservices are just getting started, software development has never been easier, and the two biggest companies in the cloud are compete for users. It’s going to keep getting easier and cheaper.

Why should we use C programming language?

1. C language has less number of keywords, as we know that in a standard version of C programming language there are only 32 keywords and 32 keywords are easy to remember.
2. C language has less number of standard libraries and library functions (those are well-established and standardized portable) but they can be used to design any kind of (almost) functions.
3. C language has rich set of data types with awesome features of type qualifiers like const, volateile etc.
4. We have already discussed in a previous post about low level and high level languages. C language is one of the most popular programming languages which are able to make low level applcations like device drivers, operating systems, firmware etc along with the high level application like desktop applications. Note that: due to this amazing feature C programming language is known as “Middle Level programming language”.
5. C language is very efficient, popular and well understood.
6. C language is much popular for embedded systems programming due to its flexibility.
7. Programs writtein in C programming language are easy to read, understand and edit.
8. C language is free, and you do not have to pay anything even if you are using C language for embedded systems. We can easily access any hardware device attached with the micro-controller (please, don’t confuse here, in embedded systems micro-controllers are like processors), for example: we can capture images through the camera, record videos by writing functions (known as device drivers) for it.
9. C language has other features like Portability, Modularity, and Structure Oriented. Since C programming language is very old language, still it is more popular due to its features.

Some basic rules of writing a C program

C program is the set of statement written in C programming language. Each program should follow the standards (rules) to make your code working everywhere.

1. Comments/Documentation

Comments are not required, but the comments are as important as the code. So it is first rule that you should remember to comment/document the code properly.

What can you document:

* Purpose of the code
* Name of the programmer, reviewer, code writing, editing date etc
* Resources (inputs) that will be used to run the code
* Explain each logic, what is that for and why you used it
* You can also write note for the developers, in which you can explain your logic/algorithm in details
* Each function declaration, definition should be documented
* Anything that you can explain, but there is no need to write unnecessary comments with the code that is already cleared.

1. Statement Termination by Semicolon (;)
2. 3. Tokens, Identifiers related rules

You must be aware about the C language tokens to differentiate what is keyword, what is identifier etc.

Some of the points that you must know:

* Keywords are the reserved words in the compilers and you cannot use them for other purposes
* C language is case sensitive programming language. Here, small case identifiers and uppercase identifiers are different.
* Same as keyword, library functions are also declared in the header files and you must have to include header file in the program. For example, if you are using sqrt() function to find the square root of any number, you will have to include math.h header file

C Tokens

C tokens are the smallest & individual words/characters, in other words, these are the smallest building blocks which are used to write a program.

C tokens are divided into the following categories:

1. Keywords
2. Identifiers
3. Variables
4. Constants
5. Literals
6. Operators
7. Special Character

Keywords

Keywords are the reserved words in the C Compilers, we can’t change the meanings of them, these are the words which have predefined meaning in the compilers.

C Langauge has 32 keywords, which are:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| auto | extern | sizeof | break | static | case | for | struct |
| goto | switch | const | if | typedef | continue | int | char |
| float | union | default | long | unsigned | do | void | register |
| double | return | volatile | else | short | whil | enum | signed |

Identifiers

An identifier is the name of variable, const, array, structure, functions etc. These are the user defined name which is used to define a name of particular memory location or particular data types:

Here are some of the rules to define an identifier:

* An identifier must start with a character or underscore
* Only 32 characters are allowed as an identifier name
* Spaces are not allowed in the identifier name
* A keyword cannot be used as the identifier name
* Characters, digits, underscores are allowed in identifier name
* An identifier must be meaningful or descriptive

Variables

Variables are names used to represent a memory location, which represent predefined or customize data type. A variable’s value can be changed any time during the program’s execution.

Constants

Constants are the name of those memory blocks, which are not changeable. A constant’s value is fixed during the program’s execution.

# Let Us See (Reference Book)

## Exercise on Functions & Pointers (page 201 - 227)

Attempt 1:

### [A] What would be the output of the following programs?

1. Output:

Only stupids use C?

Fools too use C!

1. Output: Infinite loop of “C to it that C survives”
2. Output: 100
3. Output: 4500

[B]

1. Returning two values are not possible without the use of pointers
2. The function was defined within main()
3. The formal arguments were not declared
4. Function was not declared using void, and it doesn’t have a return value
5. Function was defined within main()
6. The function that was used as the actual argument doesn’t have a return value

[C]

1. No, main() is not defined
2. .
3. False
4. False
5. True
6. False
7. True
8. True
9. True
10. False
11. True
12. False

[D]

1. Write a function to calculate the factorial value of any integer entered through the keybaord:

**#include** <stdio.h>

**int** **factorial**(**int** number) {

**int** x = number;

**if** (x == 1) {

**return** 1;

} **else** {

number = number \* factorial(x-1);

}

**return** number;

}

**int** **main**() {

**int** number;

**int** answer;

**printf**("Please enter a number:\n");

**scanf**("%d", &number);

answer = factorial(number);

**printf**("The factorial of %d is %d", number, answer);

}

How to write a function that takes pointers as arguments?!

Attempt 1:

#include <stdio.h>

void factorial(int \*pnumber, int \*panswer) {

int x = \*pnumber;

if (x == 1) {

printf("x = 1, function will now be exited\n");

} else {

\*panswer = \*pnumber \* factorial(\*&x, \*&panswer);

}

}

int main() {

int number = 0;

int answer = 0;

printf("Please enter a number:\n");

scanf("%d", &number);

factorial(&number, &answer);

printf("The factorial of %d is %d", number, answer);

}

1. Write a function power (a, b), to calculate the value of a raised to b.

**int** **power**(**int** base, **int** power) {

**int** answer = 1;

**for** (**int** i = 0; i < power; i++) {

answer \*= base;

**printf**("loop ran for %d times\n", i+1);

}

**return** answer;

}

**int** **powerRecur**(**int** base, **int** power) {

**int** answer = base;

**int** x = power;

**if** (x == 0) {

**return** 1;

} **else** **if** (x >= 1) {

answer \*= powerRecur(base, power-1);

}

**return** answer;

}

**int** **main**() {

**int** userInputBase = 0;

**int** userInputPower = 0;

**int** answer;

**int** answerRecur;

**printf**("Please input the base:");

**scanf**(" %d", &userInputBase);

**printf**("Please input the power:");

**scanf**(" %d", &userInputPower);

answer = power(userInputBase, userInputPower);

**printf**("\nThe power of %d to %d is %d.\n", userInputBase, userInputPower, answer);

answerRecur = powerRecur(userInputBase, userInputPower);

**printf**("\nThe power of %d to %d is %d.\n", userInputBase, userInputPower, answerRecur);

}

1. Write a general-purpose function to convert any given year into its roman equivalent. The following table shows the roman equivalents of decimal numbers (1998 = mdcccclxxxviii, 1525 = mdxxv):
   * 1 = i
   * 5 = v
   * 10 = x
   * 50 = l
   * 100 = c
   * 500 = d
   * 1000 = m

**#include** <stdio.h>

**#include** <stdlib.h>

**int** **main**() {

**char** userInputYear[] = "";

// char \*userInputYear1 = &userInputYear[0];

// char \*userInputYear2 = &userInputYear[1];

// char \*userInputYear3 = &userInputYear[2];

// char \*userInputYear4 = &userInputYear[3];

**char** userInputYear1;

**char** userInputYear2;

**char** userInputYear3;

**char** userInputYear4;

**printf**("Please into the year1: ");

**scanf**(" %c", &userInputYear1);

**printf**("Please into the year2: ");

**scanf**(" %c", &userInputYear2);

**printf**("Please into the year3: ");

**scanf**(" %c", &userInputYear3);

**printf**("Please into the year4: ");

**scanf**(" %c", &userInputYear4);

// scanf("%s", &userInputYear);

//

// printf("You have entered the year %s\n", userInputYear);

//// for (int i = 0; i < 4; i ++) {

//// printf("Digit %d = %c\n", i+1, userInputYear[i]);

//// }

//// printf("Digit 1 = %c\n", userInputYear[0]);

//// printf("Digit 2 = %c\n", userInputYear[1]);

//// printf("Digit 3 = %c\n", userInputYear[2]);

//// printf("Digit 4 = %c\n", userInputYear[3]);

//

**printf**("Digit 1 = %c\n", userInputYear1);

**printf**("Digit 2 = %c\n", userInputYear2);

**printf**("Digit 3 = %c\n", userInputYear3);

**printf**("Digit 4 = %c\n", userInputYear4);

**switch** (userInputYear1) {

**case** '1':

**printf**("m");

**break**;

**default**:

**printf**("userInputYear1 does not fit any specified case");

**break**;

}

}

Attempt 2

#include <stdio.h>

#include <stdlib.h>

int main() {

int userInputYear;

printf("Please input a year from 0 - 5000: ");

scanf("%d", &userInputYear);

switch (userInputYear) {

case 0:

printf("Roman equivalent of %d is 0\n");

break;

case 1000:

printf("m");

default:

printf("Please input a valid year.\n");

break;

}

}

1. Any year is entered through the keyboard. Write a function to determine whether the year is a gap year or not.
2. A positive integer is entered through the keyboard. Write a function to obtain the prime factors of this number
   * For example, prime factors of 24 are 2, 2, 2 and 3. Whereas prime factors of 35 are 5 and 7.

[E]

1. 3.14
2. Only he men use C! (c is empty, therefore it would also output 0 on the next line.

[F]

1. Write a function which receives a float and an int from main(), finds the product of these two and returns te product which is printed through main().
2. Write a function that receives 5 integers and returns the sum, average and standard deviation of these numbers. Call this function from main() and print the results in main().
3. Write a function that receives marks received by a student in 3 subjects and returns the average and percentage of these marks. Call this function from main() and print the results in main().

[G]

# Programming Language C

## Drawing a Shape

## We can use printf to draw a triangle on the console:

**printf**(" /|\n");

**printf**(" / |\n");

**printf**(" / |\n");

**printf**(" /\_\_\_|\n");

**return** 0;

## Variables and types

### Data Types

Integers – Whole numbers which can be either positive or negative. Defined using:

* Char (range from -128 to 127)
* Int
* Short
* Long (range from -2,147,483,648 to 2,147,483,647)
* Long long

Unsigned integers – whole numbers which can only be positive. Defined using:

* Unsigned char
* Unsigned int
* Unsigned short
* Unsigned long
* Unsigned long long

Floating point numbers – real numbers (numbers with fractions). Defined using:

* Float
* Double

C does not have a Boolean type. Usually, it is defined using the following notation:

#define BOOL char

#define FALSE 0

#define TRUE 1

C uses arrays of characters to define strings.

### Defining Variables

For numbers ,we will usually use the type int, which an integer in the size of a “word” the default numer size of the machine which your program is compiled on. On most computers today, it is a 32-bit number, which means the number can range from -2,147,483,648 to 2,147,483,647.

**int** a = 1;

printf(""%d"", a); // Outputs 1

### Arrays

Arrays can only have one type of variable, because they are implemented as a sequence of values in the computer’s memory. Because of that, accessing a specific array cell is very efficient.

You can initialize an array in the following way:

**int** nubmers[5];

numbers[0] = 1;

numbers[1] = 2;

numbers[2] = 3;

numbers[3] = 4;

numbers[4] = 5;

For multi-dimension arrays, it can be defined in the following ways:

**char** vowels[][5] = {

{‘A’, ‘E’, ‘I’, ‘O’, ‘U’},

{‘a’, ‘e’, ‘i’, ‘o’, ‘u’}

};

OR

**char** vowels[2][5] = {

{‘A’, ‘E’, ‘I’, ‘O’, ‘U’},

{‘a’, ‘e’, ‘i’, ‘o’, ‘u’}

};

The above two pieces of code will have the same result, because the compiler would already know tat there are two “dimensions”. However, it is necessary to specify the how many values (integers, characters, floats, etc) we will have in each dimension.



The following two pieces of code are identical to the computer:

**int** a[3][4] = {

{0, 1, 2, 3},

{4, 5, 6, 7},

{8, 9, 10, 11}

}

AND

**int** a[3][4] = {1,2,3,4,5,6,7,8,9,10,11};

The “[3]” which indicates the wanted row is also optional in both cases.

## Strings

### Defining Strings

Strings in C are actually arrays of characters. Although using pointers in C is an advance subject, fully explained later on, we will use pointers to a character array to define simple strings, in the following manner:

**char** \* name = "John Smith";

This method creates a string which we can only use for reading. If we wish to define a string which can be manipulated, we will need to define it as a local character array:

**char** name[] = "John Smith";

This notation is different because it allocates an array variable so we can manipulate it. The empty brackets notation [] tells the compiler to calculate the size of the array automatically. This is in fact the same as allocating it explicitly, adding one to the length of the string:

**char** name[] = "John Smith";

/\* Is the same as \*/

**char** name[11] = "John Smith";

The reason that we need to add one, although the string John Smith is exactly 10 characters long, is for the string termination: a special character (equal to 0) which indicates the end of the string. The end of the string is marked because the program does not know the length of the string – only the compiler knows it according to the code.

### String Length with strlen()

The function ‘strlen’ returns the length of the string which has to be passed as an argument:

**char** \* name = "Nikhil"";

printf("%d\n", strlen(name);

### String Comparison with strncmp()

The function strncmp compares between two strings, returning the number 0 if they are equal, or a different number if they are different. The arguments are the two strings to be compared, and the maximum comparison length. There is also an unsafe version of this function called strcmp, but it is not recommended to use it. For example:

**char** name = "John";

**if** (**strncmp**(name, "John", 4) == 0) {

**printf**("Hello, John\n");

} **else** {

**printf**("You are not John. Go away.\n");

}

The argument 4 is a “threshold”, if it is 1, only the first character is compared.

### String Concatenation with strncat

The function ‘strncat’ appends first n characters of src string sring to the destination string where n is min(n, length(src)); The arguments passed are destination string, source string, and n – maximum number of characters to be appended. For example:

**char** dest[20]="Hello";

**char** src[20]="World";

strncat(dest,src,3);

printf("%s\n",dest);

strncat(dest,src,20);

printf("%s\n",dest);

## Scanf() & Printf() Properties

### Format specifiers

|  |  |  |
| --- | --- | --- |
| Character | (char) | %c |
| Integer | (int) | %d |
| Unsigned Integer | (unsigned int) | %ld |
| Long | (long) | %ld |
| Unsigned Long | (unsigned long) | %lu |
| Float | (float) | %f |
| Double | (double) | %lf |
| Octal Value | (octal value) | %o |
| Hexadecimal Value | (hex value) | %x |
| String | (char[]) | %s |

Escape Sequence for printf() & scanf():

|  |  |
| --- | --- |
| \\ | \ |
| \” | “ |
| \’ | ‘ |
| \? | ? |
| \a | Alert |
| \b | Back Space |
| \n | New Line |
| \t | Horizontal Tab |
| \v | Vertical Tab |
| \r | Carriage Return |

Padding Integers with 0’s / Spaces

We can use %[n]d, where n is the number of characters, and –n would be fore right space padding.

For spacing with 0’s, %0[n]d can be used.

int main() {

int a = 10;

printf("\n\nLeft padding with spaces: a =%5d\n", a);

printf("\n\nLeft padding with spaces: a =%10d\n", a);

printf("\n\nLeft padding with spaces: a =%15d\n", a);

}

## Padding Float values with 0’s

Default float value prints 6 digits after precision

%05f – Equals to default

%08.3f – “.3f” prints 3 digits after precision, 08 prints 8 digits of 0’s including digits before and after the precision

%.2f – 2 digits after precision

String Formatting & Padding Strings with spaces

## For Loops

In order to print an array using For loop, it can be done in the following manner:

**char** array[] = "This is an array of string";

**int** arrayLength = **strlen**(array);

**for** (**int** i = 0; i < arrayLength; i++) {

**printf**("%c", array[i]);

}

## Getting User Input

### scanf

We can get user input using scanf:

**int** age;

**double** gpa;

**char** grade;

**int** i = 0;

**printf**("Enter your age: ");

**scanf**("%d", &age);

**while** (age <=0 && i < 5) {

**printf**("Please input your correct age.");

**scanf**("%d", &age);

i++;

}**if** (i > 4) {

**printf**("You don't even know your own age, goodbye.");

**return** 0;

}

**printf**("Enter your GPA: ");

**scanf**("%lf", &gpa);

**printf**("Enter your grade: ");

**scanf**("%c", &grade);

**scanf**("%c", &grade); //For some reason the first second last scanf is ignored

**if** (i < 5) {

**printf**("The age you have entered is %d \nThe GPA you have entered is %f \nThe grade you have entered is %c", age, gpa, grade);

}

### getch()

This function is used to read single character from standard input device without echoing, and does not require [return] key after input. This function is declared in the conio.h header file:

## getchar()

This function is used to read a single character from standard input device with echoing, and requires [return] key after input. This function is declared in stdio.h header file:

int main() {

char ch;

printf("Enter a character: ");

ch = getchar();

printf("The character that was entered was : %c\n", ch);

}

### putchar()

This function is used to print out a single character on the terminal:

## While Loops

While loops are similar to for loops, but have less less functionality. A while loop continues excuting the while block as long as the condition in the while remains true.

### Loop Directives (break & continue)

### There are two important loop directives that are used in conjunction with all loop types in C – the break and continue directives.

**int** n = 0;

**while** (1) {

n++;

**printf**("%d", n);

**if** (n == 10) {

**break**;

}

}

In the following code, the continue directive causes the printf command to be skipped, so that only even numbers are printed out:

**int** n = 0;

**while** (n < 10) {

n++;

**if** (n % 2 == 1) {

**continue**; //go back to the start of the while block

}

**printf**("The number %d is even \m", n); //we reach this code only if n is even

}

Exercise

**int** array[] = {1,2,3,4,5,6,7,8,9,0,11,12,13,14,15,16,17,18,19,20};

**int** i = 0;

**while** (1) {

i++;

**if** (array[i - 1] < 5) {

**continue**;

} **else** **if** (array[i - 1] > 10) {

**break**;

} **else** {

**printf**("The number is %d\n", n);

}

}

## Functions

C functions are simple, but because of how C works, the power of function is a bit limited.

* Functions receive either a fixed or variable amount of arguments.
* Functions can only return one value, or return no value.

In C, arguments are copied by value to functions, which means that we cannot change the arguments to affect their value outside of the function. To do that, we must use pointers, which are taught later on.

Functions are defined using the following syntax:

**int** **foo**(**int** bar) {

**printf**("%d", bar \* 2);

**return** bar \* 2;

}

**int** **main**() {

foo(1);

}

In C, functions must be first defied before they are used in the code. They can be either declared first and then implemented later on using a header file or in the begining of the C file, or they can be implemented in the order they are used (less preferable).

The correct way to use functions is as follows:

**int** **foo**(**int** bar);

**int** **main**() {

**printf**("The value of foo is %d", foo(1));

}

**int** **foo**(**int** bar) {

**return** bar \* 1;

}

We also can create functions that do not return a value by using the keyword void.

**void** **moo**() {

}

**int** **main**() {

moo();

}

### Exercise

**void** **print\_big**(**int** x);

**int** **main**() {

print\_big(50);

print\_big(40);

}

**void** **print\_big**(**int** x) {

**if** (x > 10) {

**printf**("%d is big\n", x);

}

}

### Accessing Files

We can create a file pointer for a file, which will point to the address of the file on the computer.

There are three basic file modes:

1. w -> Write (will overwrite all the contents of the specified file)
2. r -> Read
3. a -> Append (will add onto the contents of the specified file)

**int** **main**(){

FILE \* filepointer = **fopen**("employees.txt", "a"); //file mode: r = Read, w = write, a = append

**fprintf**(filepointer, "Test\n");

**fclose**(filepointer);

}

We can use fgets to read the file line by line.

**char** line[255];

FILE \* filepointer = **fopen**("employees.txt", "r"); //file mode: r = Read, w = write, a = append

**for** (**int** i = 0; i < 3; i++) {

**fgets**(line, 255, filepointer);

**printf**("%s", line);

}

**fclose**(filepointer);

## Static

Static is a keyword in the C programming language. It can be used with variables and functions.

### What is a static variable?

By default, variables are local to the scope in which they are defined. Variables can be declared as static to increase their scope up to file containing them. As a result, these variables can be accessed anywhere inside a file.

Consider the following scenario – we want to count the runners participating in a race:

**int** **runner**() {

**int** count = 0;

count++;

**return** count;

}

**int** **main**() {

**printf**("%d", runner()); //returns 1

**printf**("%d", runner()); //returns 1

}

We will see that count is not updated because it is removed from memory as soon as the function completes. If static is used, however:

**int** **runner**() {

**static int** count = 0;

count++;

**return** count;

}

**int** **main**() {

**printf**("%d", runner()); //returns 1

**printf**("%d", runner()); //returns 2

}

### What is a static function?

By default, functions are global in C. If we declare a function with static, the scope of that function is reduced to the file containing it.

The syntax looks like this:

**static** **void** **fun**(**void**) {

**printf**("I am a static function.");

}

**int** **main**() {

fun();

}

### Static vs Global?

While static variables have scope over the file containing them making them accessible only inside a given file, global variables can be accessed outside the file too.

### Exercise

**int** **sum** (**int** num) {

**static** **int** sum = 0;

sum += num;

**return** sum;

}

**int** **main**() {

**printf**("%d ",sum(55));

**printf**("%d ",sum(45));

**printf**("%d ",sum(50));

**return** 0;

}

## Pointers

Pointers are also variables and play a very important role in C programming language. They are used for several reasons, such as:

* Strings
* Dynamic memory allocation
* Sending function arguments by reference
* Building complicated data structur
* Pointing to functions
* Building special data structure (.e. Tree, Tries, etc...)

### What is a pointer?

A pointer is essentially a simple integer variable which holds a memory address that points to a value, instead of holding the actual value itself.

The computer’s memory is a sequential store of data, and a pointer points to a specific part of the memory. Our program can use pointers in such a way that the pointers point to a large amount of memory – depending on how much we decide to read from that point on.

### Memory Address

When a variable is created, the value of the variable is stored somewhere on the RAM, and it is identified using a memory address. When the compiler looks for the value, it uses the memory address to lookup the value.

The memory address of a variable can be printed using %p as shown below:

**int** age = 23;

**printf**("The memory address of age is %p", age);

### Strings as pointers

We’ve already discussed strings, but now we can dive in a bit deeper ad understand what strings in C really are (which are called C-Strings to differentiate them from other strings when mixed with C++)

The following line does three things:

**char** \* name = "John";

1. It allocates a local (stack) variable called name, which is a pointer to a single character.
2. It causes the string “John” to appear somewhere in the program memory (after it is compiled and executed, of course).
3. It initializes the name argument to point to where the  J character resides at (which is followed by the rest of the string in the memory).

If we try to access the name variable as an array, it will work, and will return the ordinal value of the character J, since the name variable actually points exactly to the beginning of the string.

Since we know that the memory is sequential, we can assume that i fwe move ahead in the memory to the next character, we’ll receive the next letter in the string, until we reach the end of the string, marked with a null terminator (the character with the ordinal value of 0, noted as \0).

### Dereferencing

Dereferencing is the act of referring to where the pointer points, instead of the memory address. We are already using dereferencing in arrays – but we just didn’t know it yet. The brackets operator – [0] for example, accesses the first item of the array. And since arrays are actually pointers, accessing the first item in the array is the same as dereferencing a pointer. Dereferencing a pointer is done using asterisk operator \*.

If we want to create an array that will point to a different variable in our stack, we can write the following code:

**int** a = 1;

**int** \* pointer\_to\_a = &a;

**printf**("The value a is %d\n", a);

**printf**("The value of a is also %d\n", \*pointer\_to\_a);

Notice that we used the & operator to point at the variable a, which we have just created.

We then referred to it using the dereferencing operator. We can also change the contents of the dereferenced variable:

**int** a = 1;

**int** \* pointer\_to\_a = &a;

a += 1;

\*pointer\_to\_a += 1;

**printf**("The value a is now %d\n", a);

### Exercise

**int** n = 10;

**int** \* pointer\_to\_n = &n;

\*pointer\_to\_n += 1;

**if** (pointer\_to\_n != &n) **return** 1;

**if** (\*pointer\_to\_n != 11) **return** 1;

**printf**("Done!\n");

**return** 0;

## Structures

C structures are special, large variables which contain several named variables inside. Structures are the basic foundation for objects and classes in C. Structures are used for:

* Serialization of data
* Passing multiple arguments in and out of functions through a single argument
* Data structures such as linked lists, binary trees, and more

The most basic example of structures are points, which are a single entity that contains two variables – x and y. Let’s define a point:

**struct** point {

**int** x;

**int** y;

};

Now, let’s define a new point, and use it. Assume the function draw receives a point and draws it on a screen. Without structs, using it would require two arguments – each for every coordinate:

**int** x;

**int** y;

draw(x, y);

Using structs, we can pass a point argument:

**struct** point p;

p.x = 10;

p.y = 5;

draw(p);

To access the point’s variables, we use the dot operator.

### Typedefs

Typedefs allow us to define types with a different name – which can come in handy when dealing with structs and pointers. In this case, we’d want to get rid of the long definition of a point structure. We can use the following syntax to remove the struct keyword from each time we want to define a new point:

**typedef** **struct** {

**int** x;

**int** y;

};

This will allow us to define a new point with only the following line:

point p;

Structures can also hold pointers – which allows them to hold strings, or pointers to other structures as well – which is their real power. For example, we can define a vehicle structure in the following manner:

**typedef** **struct** {

**char** \* brand;

**int** model;

} vehicle;

Since brand is a char pointer, the vehicle type can contain a string (which, in this case, indicates the brand of the vehicle).

In order to store a string in the structure, we will need to use the strcpy function (String Copy):

**struct** student {

**char** name[50];

**int** age;

};

**int** **main**(){

**struct** student student1;

**strcpy**(student1.name, "Billy");

student1.age = 23;

**printf**("%s", student1.name);

}

### Exercise

**struct** person {

**char** \* name;

**int** age;

};

**int** **main**() {

**struct** person billy;

billy.name = "Billy";

billy.age = 23;

**printf**("%s is %d years old.", billy);

}

**typedef** **struct** {

**char** \* name;

**int** age;

} person;

**int** **main**() {

person john;

john.name = "John";

john.age = 27;

**printf**("%s is %d years old.", john.name, john.age);

}

## Function Arguments by Reference

Function arguments are passed by value, which means they are copied in and out of functions. But what if we copied pointers to values instead of values themselves? This will enable us to give functions control over variables and structures of the parent functions, and not just a copy of them.

Let’s say we want to write a function which increments a number by one, called addone. This will not work:

**int** **addone**(**int** n) {

n++;

}

**int** **main**() {

**int** n;

**printf**("Before %d\n", n);

addone(n);

**printf**("After %d\n", n);

}

However, this will work:

**int** **addone**(**int** \* n) {

(\*n)++;

}

**int** **main**() {

**int** n;

**printf**("Before %d\n", n);

addone(&n);

**printf**("After %d\n", n);

}

The difference is that the second version of addone receives a pointer to the variable n as an argument, and then it can manipulate it, because it knows where it is in the memory.

Notice that when calling the addone function, we must pass a reference to the variable n, and not the variable itself – this is done so that the functions knows the address of the variable, and won’t just receive a copy of the variable itself.

### Pointers to structures

Let’s say we want to create a function which moves a point forward in both x and y directions, called move. Instead of sending two pointers, we can now send only one pointer to the function of the point structure:

**void** **move**(point \*p) {

(\*p).x++;

(\*p).y++;

}

However, if we wish to dereference a structure and access one of it’s internal members, we have a shorthand syntax for that, because this operation is widely used in data structures. We can rewrite this function using the following syntax:

**void** **move**(point \*p) {

p->x++;

p->y++;

}

## Exercise

**typedef** **struct** {

**char** \* name;

**int** age;

} person;

**void** **birthday**(person \* p) {

p->age++;

};

**int** **main**() {

person john;

john.name = "John";

john.age = 27;

**printf**("%s is %d years old.\n", john.name, john.age);

birthday(&john);

**printf**("Happy birthday! %s is now %d years old.\n", john.name, john.age);

**return** 0;

}

## Dynamic Allocation

Dynamic allocation of memory is a very important subject in C. It allows building complex data structures such as linked lists. **Allocating memory dynamically helps us to store data without initially knowing the size of the data in the same time we wrote the program.**

To allocate a chunk of memory dynamically, we have to have a pointer ready to store the location of the newly allocated memory. We can access memory that was allocated to us using that same pointer, and we can use that pointer to free the memory again, once we have finished using it.

Let’s assume we want to dynamically allocate a person structure. The person is defined like this:

**typedef** **struct** {

**char** \* name;

**int** age;

} person;

To allocate a new person in the myperson argument, we use the following syntax:

person \* myperson = **malloc**(**sizeof**(person));

This tells the compiler that we want to dynamically allocate just enough to hold a person struct in memory, and then return a pointer to the newly allocated data.

Note that sizeof is not an actual function, because the compiler interprets it and translates it to the actual memory size of the person struct.

To access the person’s members, we can use the -> notation:

myperson->name = "John";

myperson->age = 27;

After we are done using the dynamically allocated struct, we can release it using free:

**free**(myperson);

Note that the free does not delete the myperson variable itself, it simply releases the data that it points to. The myperson variable will still point to somewhere in the memory – but after calling myperson we are not allowed to access that area anymore. We must not use that pointer again until we allocate new data using it.

### Exercise

**typedef** **struct** {

**int** x;

**int** y;

} point;

**int** **main**() {

point \* mypoint = **malloc**(**sizeof**(point));

/\* Dynamically allocate a new point

struct which mypoint points to here \*/

mypoint->x = 10;

mypoint->y = 5 ;

**printf**("mypoint coordinates: %d, %d\n", mypoint->x, mypoint->y);

**free**(mypoint);

**return** 0;

}

### Arrays and Pointers

In a previous tutorial on Pointers, you learned that a pointer is a given data type that can store the address of any variables of that particular data type. For example, in the following code, the pointer variable pc stores the address of the character variable c:

**char** c = 'A';

**char** \*pc = &c;

**printf**("The value of c is %d\n", c);

**printf**("The memory address of c is %p\n", &c);

**printf**("The value of \*pc is %p\n", pc);

Here, c is a scalar variable that can store only a single value. However, you are already familiar with arrays that can hold multiple values of the same data type in a contiguously allocated memory block. So, you might wonder, can we have points to arrays too? Indeed we can.

Let us start with an example code and look at its outputs. We will discuss its behavior subsequently:

**char** vowels[] = {'A', 'E', 'I', 'O', 'U'};

**char** \*pvowels = &vowels;

**for** (**int** i = 0; i < 5; i++) {

**printf**("&vowels[%d]: %u, (pvowels + %d): %u, (vowels + %d): %u\n", i, &vowels[i], i, pvowels + i, i, vowels + i);

}

**for** (**int** i = 0; i < 5; i++) {

**printf**("vowels[%d]: %c, \*(pvowels + %d): %c, \*(vowels + %d): %c\n", i, vowels[i], i, \*(pvowels + i), i, \*(vowels + i));

}

Output:

&vowels[0]: 6356735, (pvowels + 0): 6356735, (vowels + 0): 6356735

&vowels[1]: 6356736, (pvowels + 1): 6356736, (vowels + 1): 6356736

&vowels[2]: 6356737, (pvowels + 2): 6356737, (vowels + 2): 6356737

&vowels[3]: 6356738, (pvowels + 3): 6356738, (vowels + 3): 6356738

&vowels[4]: 6356739, (pvowels + 4): 6356739, (vowels + 4): 6356739

vowels[0]: A, \*(pvowels + 0): A, \*(vowels + 0): A

vowels[1]: E, \*(pvowels + 1): E, \*(vowels + 1): E

vowels[2]: I, \*(pvowels + 2): I, \*(vowels + 2): I

vowels[3]: O, \*(pvowels + 3): O, \*(vowels + 3): O

vowels[4]: U, \*(pvowels + 4): U, \*(vowels + 4): U

As you rightly guessed, &vowels[i] gives the memory locatino of the /th (ith) element of th array vowels. Moreover, since this is a character rarray, each element occupies one byte so that the consecutive memory addresses are separated by a single byte. We also created a pointer, pvowels, and assigned the address of the array vowels to it. pvowels + i is a valid operation; although in general, this may not always be meaningful (explored further in Pointer Arithmetics). In particular, the output shown above indicates that &vowels[i] and pvowels + i are equivalent. Feel free to alter the data types of the array and pointer variables to test this out.

If you look carefully at the previous code, you will notice that we also used another apparently surprising notation: vowels + i. Moreover, pvowels + i and vowels + i returns the same thing – address of the /th (ith) element of the array vowels. Why is that so?

This is because the name of an array itself is a (constant) pointer to the first element of the array. In other words, the notations vowels, &vowels[0], and vowels + o all point to the same location.

You may go through Pointers and Arrays for a further detailed discussion on this topic.

### Dynamic Memory Allocation for Arrays

By now we know that we can traverse an array using pointers. Moreover, we also know that we can dynamically allocate (contiguous) memory using blocks pointers. These two aspects can be combined to dynamically allocate memory for an array. This is illustrated in the following code:

**int** n = 5;

**char** \*pvowels = (**char** \*) **malloc**(n \* **sizeof**(**char**));

**char** \*a = **malloc**;

pvowels[0] = 'A';

pvowels[1] = 'E';

\*(pvowels + 2) = 'I';

pvowels[3] = 'O';

\*(pvowels + 4) = 'U';

**for** (**int** i = 0; i < n; i++) {

**printf**("%c ", pvowels[i]);

}

**printf**("\n");

**free**(pvowels);

In the above code, we allocated five contiguous bytes of memory to store five characters. Subsequently, we used array notations to traverse the blocks of memory as if pvowels is an array. However, remember that pvowels actually is a pointer, and as noted in Pointers and Arrays, pointers and arrays, in general, are not the same thing.

So when is this useful? Remember that while declaring an array, the number of elements that it would contain must be known beforehand. Therefore, in some scenarios it might happen that the space allocated for an array is either too less than the desired space or too much more. However, by using dynamic memory allocation, one can allocate just as much memory as required by a program. Moreover, unused memory can be freed as soon as it is no longer required by invoking the free() function. On the down side, with dynamic memory allocation, one must responsibly call free() wherever relevant. Otherwise, memory leaks would occur.

We conclude this tutorial by looking at dynamic memory allocation for a two-dimension array. This can be generalized to n-dimensions in a similar way. Unlike one-dimension arrays, where we used a pointer, in this case we require a pointer to a pointer, as shown below.

**int** nrows = 2;

**int** ncols = 5;

**int** i, j;

// Allocate memory for nrows pointers

**char** \*\*pvowels = (**char** \*\*) **malloc**(nrows \* **sizeof**(**char** \*));

// For each row, allocate memory for ncols elements

pvowels[0] = (**char** \*) **malloc**(ncols \* **sizeof**(**char**));

pvowels[1] = (**char** \*) **malloc**(ncols \* **sizeof**(**char**));

pvowels[0][0] = 'A';

pvowels[0][1] = 'E';

pvowels[0][2] = 'I';

pvowels[0][3] = 'O';

pvowels[0][4] = 'U';

pvowels[1][0] = 'a';

pvowels[1][1] = 'e';

pvowels[1][2] = 'i';

pvowels[1][3] = 'o';

pvowels[1][4] = 'u';

**for** (i = 0; i < nrows; i++){

**for** (j = 0; j < ncols; j++) {

**printf**("%c", pvowels[i][j]);

}

}

**printf**("\n");

// Free individual rows

**free**(pvowels[0]);

**free**(pvowels[1]);

// Free the top-level pointer

**free**(pvowels);

## Recursion

Recursion occurs when a function contains within it a call to itself. Recursion can result in very neat, elegant code that is intuitive to follow. It can also result in a very large amount of memory being used if the recursion gets too deep.

Common examples of where recursion is used:

* Walking recursive data structures such as linked lists, binary tress, etc.
* Exploring possible scenarios in games such as chess

Recursion always consists of two main parts. A terminating case that indicates when teh recursion will finish and a call to itself that must make the process towards the terminating case.

For example, this function will perform multiplication by recursively adding:

**unsigned** **int** **multiply**(**unsigned** **int** x, **unsigned** y){

**if** (x == 1) {

**return** y;

} **else** **if** (x > 1) {

**return** y + multiply(x-1, y);

}

}

**int** **main**() {

**printf**("3 times 5 is %d", multiply(3,5));

**return** 0;

}

### Exercise

**int** **factorial**(**int** number);

**int** **main**() {

/\* testing code \*/

**printf**("1! = %i\n", factorial(1));

**printf**("3! = %i\n", factorial(3));

**printf**("5! = %i\n", factorial(5));

}

**int** **factorial**(**int** number){

**int** f = number;

**if**(number > 1){

f \*= factorial(number-1);

}

**return** f;

}

## Linked Lists

### Introduction

Linked lists are the best and simplest example of a dynamic data structure that uses pointers for its implementation. However, understanding pointers is crucial to understanding how linked lists work, so if you’ve skipped the pointers tutorial, you should go ack and redo it. You must also be familiar with dynamic memory allocation and structures.

Essentially, linked lists function as an array that can grow and shrink as needed, from any point in the array.

Linked lists have a few advantages over arrays:

1. Items can be added or removed from the middle of he list
2. There is no need to define an initial size

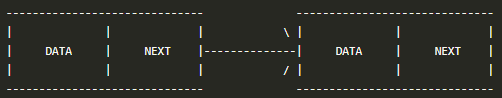
However, linked lists also have a few disadvantages:

1. There is no “random” access – it is impossible to reach the nth item in the array without first iterating over all items up until that item. This means we have to start from the beginning of the list and count how many times we advance in the list until we get to the desired item.
2. Dynamic memory allocation and pointers are required, which complicates the code and increases the risk of memory leaks and segment faults.
3. Linked lists have a much larger overhead over arrays, since linked list items are dynamically allocated (which is less efficient in memory usage) and each item in the list also must store an additional point.

### What is a Linked List?

A linked list is a set of dynamically allocated nodes, arranged in such a way that each node contains one value and one pointer. The pointer always points to the next member of the list. If the pointer is NULL, then it is the last node in the list.

A linked list is held using a local pointer variable which points to the first item of the list. If that pointer is also NULL, then the list is considered to be empty.



Let’s define a linked list node:

**typedef** **struct** node {

**int** val;

**struct** node \*next;

} note\_t;

Notice that we are defining the struct in a recursive manner, which is possible in C. Let’s name our node type node\_t.

Now we can use the nodes. Let’s create a local variable which points to the first item of the list (called head).

**typedef** **struct** node {

**int** val;

**struct** node \*next;

} node\_t;

**int** **main**() {

node\_t \* head = NULL;

head = **malloc**(**sizeof**(node\_t));

**if** (head == NULL) {

**return** 1;

}

head->val = 1;

head->next = NULL;

}

We’ve just created the first variable in the list. We must set the value, and the next item to be empty, if we want to finish populating the list. Notice that we should always check if malloc returned a NULL value or not.

To add a variable to the end of the list, we can just continue advancing to the next pointer.

**int** **main**() {

node\_t \* head = NULL;

head = **malloc**(**sizeof**(node\_t));

**if** (head == NULL) {

**return** 1;

}

head->val = 1;

head->next = **malloc**(**sizeof**(node\_t));

head->next->val = 2;

head->next->next = NULL;

}

### Iterating over a list

Let’s build a function that prints out all the items of a list. To do this, we need to use a current pointer that will keep track of the node we are currently printing. After printing the value of the node, we set the current pointer to the next node, and print again, until we’ve reached the end of the list (the next node is NULL).

**void** **print\_list** (node\_t \* head) {

node\_t \* current = head;

**while** (current != NULL) {

**printf**("%d\n", current->val);

current = current->next;

}

**printf**("\*\*\*print\_list() has reached the end of the list\*\*\*");

}

### Adding an item to the end of the list

To iterate over all the members of the linked list, we use a pointer called current. We set it to start from the head and then in each step, we advance the pointer to the next item in the list, until we reach the last item.

**void** **pushToEnd**(node\_t \* head, **int** val) {

node\_t \* current = head;

**while** (current->next != NULL) {

current = current->next;

}

current->next = **malloc**(**sizeof**(node\_t));

current->next->val = 3;

current->next->next = NULL;

} //use pushToEnd(head, 9) to call the function

The best use cases for linked lists are stacks and queues, which we will now implement:

### Adding an item to the beginning of the list (pushing to the list)

To add to the beginning of the list, we will need to do the following:

1. Create a new item and set its value
2. Link the new item to the point to the head of the list
3. Set the head of the list to be our new item

This will effectively create a new head to the list with a new value, and keep the rest of the list linked to it. Since we use a function to do this operation, we want to be able to modify the head variable. To do this, we must pass a pointer to the pointer variable (a double pointer) so we will be able to modify the pointer itself.

**void** **pushToFront**(node\_t \*\* head, **int** val) {

node\_t \* new\_node;

new\_node = **malloc**(**sizeof**(node\_t));

new\_node->val = val;

**printf**("node->val = %d\n", new\_node->val);

new\_node->next = \*head;

\*head = new\_node;

} //use pushToFront(&head, 9) to call the function

### Removing The First Item (Popping from the list)

To pop a variable, we will need to reverse this action:

* + 1. Take the next item that the head points to and save it
    2. Free the head item
    3. Set the head to be the next item that we’ve stored on the side

### Removing the last item of the list

Removing the last item from a list is a very similar to adding it to the end of the list, but with one big exception – since we have to change one item before the last item, we actually have to look two items ahead and see if the next item is the last one in the list:

**typedef** **struct** node {

**int** val;

**struct** node \* next;

} node\_t;

**void** **remove\_last**(node\_t \*head) {

**int** retval = 0;

//If there is only one item in the list, remove it

**if** (head->next == NULL) {

retval = head->val;

**free**(head);

**return** retval;

}

//Get to the second to last node in the list

node\_t \* current = head;

**while** (current->next->next != NULL) {

current = current->next;

}

//Now current points to the second to last item of the list, so let's remove current->next

retval = current->next->val;

**free**(current->next);

current-> = NULL;

**return** retval;

### Removing a specific item

To remove a specific item from the list, either by its index from the beginning of the list or by its value, we will need to go over all the items, continuously looking ahead to find out if we’ve reached the node before the item we wish to remove. This is because we need to change the location to where the previous node points to as well.

Here is the algorithm:

1. Iterate to the node before the node we wish to delete
2. Save the node we wish to delete in a temporary pointer
3. Set the previous node’s next pointer to point to the node after the node we wish to delete
4. Delete the node using the temporary pointer

There are a few edge cases we need to take care of, so make sure you understand the code.

### Code Archive For This Topic

**#include** <stdio.h>

**#include** <stdlib.h>

**typedef** **struct** node {

**int** val;

**struct** node \*next;

} node\_t;

**void** **print\_list** (node\_t \* head) {

**printf**("\*\*\*Printing the updated linked list now\*\*\*\n");

node\_t \* current = head;

**while** (current != NULL) {

**printf**("%d\n", current->val);

current = current->next;

}

}

**void** **pushToEnd**(node\_t \* head, **int** val) {

node\_t \* current = head;

**while** (current->next != NULL) {

current = current->next;

}

current->next = **malloc**(**sizeof**(node\_t));

current->next->val = val;

current->next->next = NULL;

}

**void** **pushToFront**(node\_t \*\* head, **int** val) {

node\_t \* new\_node;

new\_node = **malloc**(**sizeof**(node\_t));

new\_node->val = val;

**printf**("node->val = %d\n", new\_node->val);

new\_node->next = \*head;

\*head = new\_node;

}

**void** **removeLast**(node\_t \*head){

**int** retval = 0;

**if** (head -> next == NULL) {

retval = head->val;

**free**(head);

**return** retval;

}

node\_t \* current = head;

**while** (current->next->next != NULL) {

current = current->next;

}

retval = current->next->val;

**free**(current->next);

current->next = NULL;

**return** retval;

}

**int** **pop**(node\_t \*\* head) {

**int** retval = -1;

node\_t \* next\_node = NULL;

**if** (\*head == NULL) {

**return** -1;

}

next\_node = (\*head)->next;

retval = (\*head)->val;

**free**(\*head);

\*head = next\_node;

**return** retval;

}

**int** **removalByIndex**(node\_t \*\*head, **int** n) {

**int** retval = -1; //What is the purpose of setting retval to -1?

node\_t \* current = \*head;

node\_t \* temp\_node = NULL;

**if** (n == 0) {

**return** pop(head);

}

**for** (**int** i = 0; i < n-1; i ++) {

**if** (current->next == NULL){

**return** -1;

}

current = current->next;

}

temp\_node = current->next;

retval = temp\_node->val;

current->next = temp\_node->next;

**free**(temp\_node);

**return** retval;

}

//int removalByValue(node\_t \*\* head, int target) {

// int counter = 0;

// node\_t \* current = \*head;

// node\_t \* temp\_node = NULL;

// int retval = -1;

//

// while (current->val != target || current->next != NULL) {

// current = current->next;

// counter++;

// }

// if (counter == 0) {

// return pop(head);

// } else {

// temp\_node = current->next;

// retval = temp\_node->val;

// current->next = temp\_node->next;

// free(temp\_node);

// return retval;

// }

//}

**int** **main**() {

node\_t \* head = NULL;

head = **malloc**(**sizeof**(node\_t));

**if** (head == NULL) {

**return** 1;

}

head->val = 1;

head->next = **malloc**(**sizeof**(node\_t));

head->next->val = 2;

head->next->next = NULL;

print\_list(head);

pushToEnd(head, 3);

pushToEnd(head, 4);

print\_list(head);

pushToFront(&head, 0);

print\_list(head);

pushToFront(&head, -1);

pushToEnd(head, 5);

print\_list(head);

pop(&head);

removeLast(head);

print\_list(head);

removalByIndex(&head, 2);

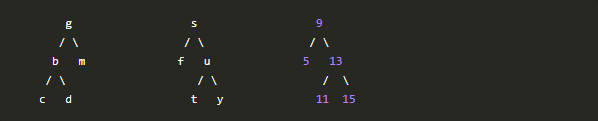
print\_list(head);

// removalByValue(&head, 3);

}

## Binary Trees

A Binary tree is a type of data structure in which each node has at most two children (left child and right child). Binary trees are used to implement binary search trees and binary heaps, and are used for efficient searching and sorting. A binary tree is a special case of a K-ary tree, where k is 2. Common operations for binary trees include insertion, deletion, and traversal. The difficult of performing the operation varies if the tree is balanced and also whether the nodes are leaf odes or branch nodes. For balanced trees, the depth of the left and right subtrees of every node differ by 1 or less. This allows for predictable depth also known as height. This is the measure of a node from root to leaf, where root is 0, and subsequent nodes are (1,2..n). This can be expressed by the integer part of log2 (n) where n is the number of nodes in the tree:



The operations performed on trees requires searching in one of two main ways:

### Depth First Search (DFS)

This is an algorithm for traversing or searching tree or graph data structures. One starts at the root and explores as far as possible along each branch before backtracking. There are three types of depth first search traversal. Pre-Order visit, left, right, In-Order Left, visit, right, Post-Order left, right, visit.

### Breadth-First Search (BFS)

This is an algorithm for traversing or searching tree or graph structure. In level-order, where we visit every node on a level before going to a lower level.

## Unions

C Unions are essentially the same as C Structures, except that instead of containing multiple variables, each with their own memory a Union allows for multiple names to the same variable. These names can treat the memory as different types (and the size of the union will be the size of the largest type, + any padding the compiler might decide to give it).

So if you wanted to be able to read a varaible’s memory in different ways, for example read an integer one byte at a time, you could have something like this:

**union** intParts {

**int** theInt;

**char** bytes[**sizeof**(**int**)];

};

Allowing you to look at each byte individually without casting a pointer and usnig pointer arithmetic:

**int** **main**() {

**union** intParts parts;

parts.theInt = 5968145; //Arbitrary number >255 (1 byte)

**printf**("The int is %i\nThe bytes are [%i, %i, %i, %i]\n", parts.theInt, parts.bytes[0], parts.bytes[1], parts.bytes[2], parts.bytes[3]);

//vs

**int** theInt = parts.theInt;

**printf**("The int is %i\nThe bytes are [%i, %i, %i, %i]\n", theInt, \*((**char**\*)&theInt+0), \*((**char**\*)&theInt+1), \*((**char**\*)&theInt+2), \*((**char**\*)&theInt+3));

// or with array syntax which can be a tiny bit nicer sometimes

**printf**("The int is %i\nThe bytes are [%i, %i, %i, %i]\n", theInt, ((**char**\*)&theInt)[0], ((**char**\*)&theInt)[1], ((**char**\*)&theInt)[2], ((**char**\*)&theInt)[3]);

}

Comining this with a structure allows you to create a “tagged” union which can e used to store multiple different types, one at a time.

For example, you might have a “umber” struct, but you don’t want to use something like this:

**struct** operator {

**int** intNum;

**float** floatNum;

**int** type;

**double** doubleNum;

};

Becayse your program has a lot of them and it takes a bit too much memory for all of the variables, so you could use this:

**struct** operatorUnion {

**int** type;

**union** {

**int** inNum;

**float** floatNum;

**double** doubleNum;

} types;

};

Like this the size of the struct is just the size of the int type + the size of the largest type in the union (the double). Not a huge gain, oly 8 or 16 bytes, but the concept can be applied to similar structs.

use:

**struct** operator op;

op.type = 0; //int, probbly better as an enum or macro constant

op.types.intNum = 352;

Also, if you don’t give the union a name then it’s memebers are accessed directly from the struct:

**struct** operator {

**int** type;

**union** {

**int** intNum;

**float** floatNum;

**double** doubleNum;

}; //no name

};

**int** **main**() {

**struct** operator op;

op.type = 0; // int

// intNum is part of the union, but since it's not named you access it directly off the struct itself

op.intNum = 352;

}

Another, perhaps more useful feature, is when you always have multiple variables of the same type, and you want to be able to use both names (for readability) and indexes (for ease of iteration), in that case you can do something like this:

**union** Coins {

**struct** {

**int** quarter;

**int** dime;

**int** nickel;

**int** penny;

}; //Anonymous struct acts the same way as an anonymous union, members are on the outer container

**int** coins[4];

};

In that example, you can see that there is a struct which contains the four (common) coins in the United States.

Since the union makes the variables share the same memory, the coins array matches with each int in the struct (in order):