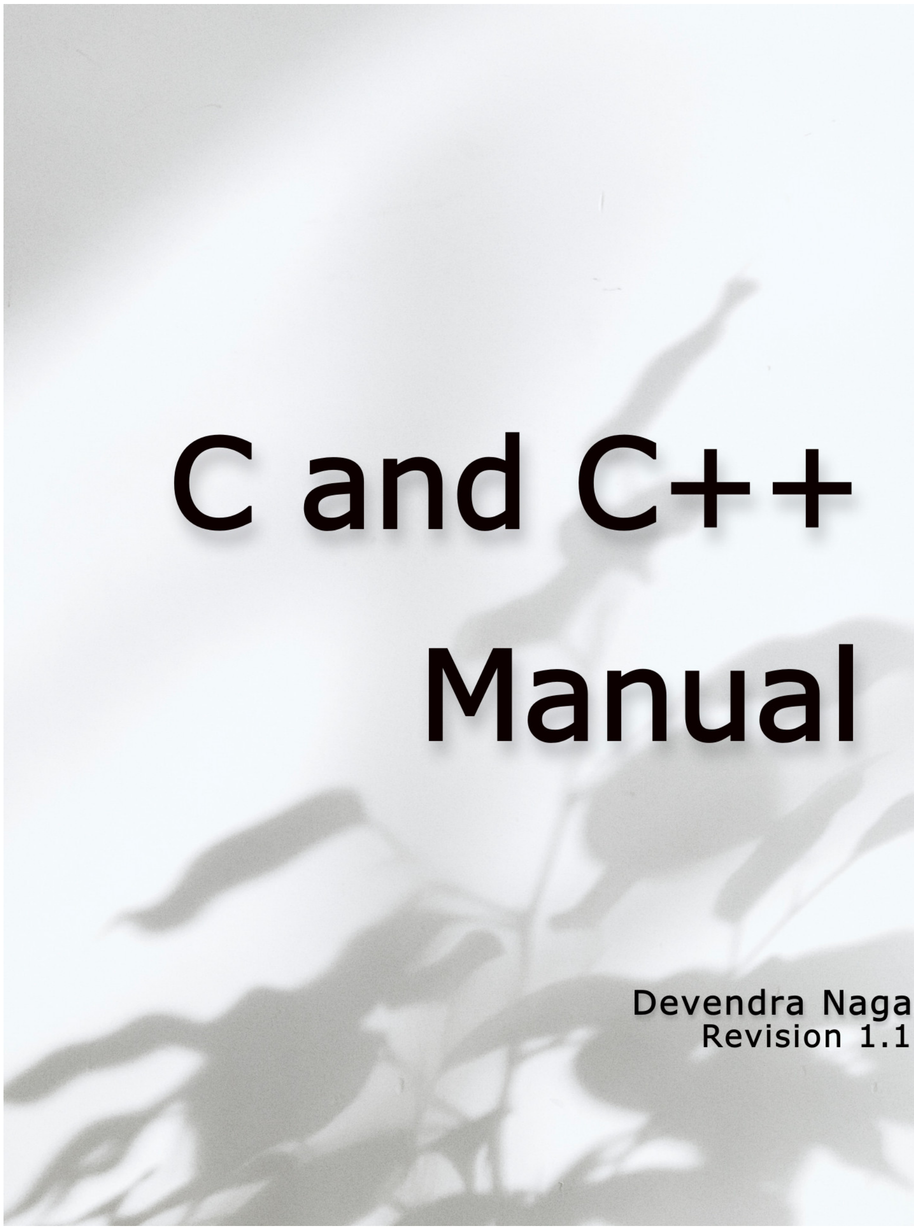

C and C++ Manual

Contents

Programming in C and C++

This is a work in progress book.

Introduction

This book tries to explain programming in C and C++. C and C++ although are the oldest programming languages they are still in use by most of the operating systems such as Linux, RTOSes such as FreeRTOS.

Both languages are still used in firmware development. Sometimes with libc and libstdc++ and sometimes not using libc and libstdc++. When writing code for Linux systems, most of the time library linkage can be used to write software. Using the library simplifies the software development as it provides most of the interfaces.

For example, the following are provided.

1. string manipulation
2. console operations (writing to and reading from console).
3. file system interaction (FILE i/o)
4. Operating system abstractions in case of C++ (such as creation and use of threads, interacting with file system)

The compiler implements the C, C++ languages. To install `gcc` and `g++` on Ubuntu:

```
sudo apt install gcc g++
```

Audience

C programming

Basic Hello World program

Below is a basic hello world program in C.

```c
#include <stdio.h>

int main()
{
    // prints Hello World on console
    printf("Hello World\n);

    return 0;
}
```

Example.1 : hello_world.c

The `main()` is a function and the program always executes from `main()`. The function call `printf`, allows the program to write to a console. The header file `stdio.h` contains the prototype of `printf`.

The `//` indicates comment line. Anything that goes after `//` for the entire line is ignored. Generally these are called C++ styled comments. These are not multi line comments.

There is another type of comment style that start with `/*` and ends with `*/`. This can be used as a multiline comment.

Here's one example,

```c
/*
 This is a multi line comment,
 showing more than one line can be written.
 */
```

The preference of using `//` or `/*` entirely depends on their coding style.

The statement `return 0` specifies that the function returns 0. In C, a function may or may not return depend on the function signature.

In order to run the program above, we need to create an executable. To create this executable, we need to compile it.

Copy and Paste the above program in a text editor and save it in a file called `hello_world.c`.

All the source code files written in C will have an extension `.c` or `.h`.

The files with `.h` extensionn are called header files. Header files contain the following.

1. Macro definitions.
2. Function prototypes.
3. Data structure definitions.

The below command is used to compile the C file.

```
gcc hello_world.c
```

Basic Data types

Below are the following data types in C. All are applicable for C++ as well.

Below are some of the data types for 32 bit systems.

S.No	type	description	ranges
1	`char`	1 byte signed integer	-127 to 127
2	`unsigned char`	1 byte unsigned integer	0 to 128
3	`short int`	2 byte signed integer	-32767 to 32767
4	`unsigned short int`	2 byte unsigned integer	0 to 65535
5	`int`	4 byte signed integer	-2147483648 to 2147483647
6	`unsigned int`	4 byte unsigned integer	0 to 4294967295
7	`float`	4 byte floating point variable	-
8	`double`	4 / 8 byte double variable	-

A variable of the above type can be declared as follows.

```
int a;
```

a is a variable of type `int`.

```
int a = 4;
```

The above statement initializes a to 4.

```
int a;

a = 4;
```

```
int a = 0x20;
```

initializes `a` with a hexadecimal number. Hexadecimals are noted with the `0x`.

The above statement assigns a to 4.

Sometimes in a program we initialize variables and we assign the variables a value sometime later.

A variable that is not initialized is called uninitialized variable. Uninitialized variables are a bigger problem when using them. The reason being that they hold some unknown value when declared.

Each declared variable is associated with a type. Read Recap section about the variables and their scope.

Declaring doubles:

```
double f = 3.14;
```

Double values are represented after the decimal `.`. Exact double values are not possible to represent in C.

sizeof operator

The `sizeof` operator is used find out the size of a data type. Below is an example.

```
#include <stdio.h>

int main()
{
    int a;

    printf("size a: %d\n", sizeof(a));

    return 0;
}
```

Example. 2 sizeof operator

The `sizeof` operator can be used on a variable or the data type itself. Such as calling `sizeof(int)` is valid.

The `%d` used to print integers. The function `printf` recognizes the integer variables given as function arguments when specified as `%d`.

The body portion start with `"` and end with `"` is called as string. More about the strings in the Strings section below.

Below are some of the format specifiers.

S.No	Format specifier	Meaning
1	`%d`	integer
2	`%c`	character
3	`%s`	string
4	`%f`	float or double
5	`%u`	unsigned integer
6	`%ld`	long integer

S.No	Format specifier	Meaning
7	`%lld`	longlong integer
8	`%lu`	long unsigned integer
9	`%llu`	long long unsigned integer
10	`%x`	hexadecimal

Below are some more functions that use the format specifiers. Functions are described in detail below.

S.No	Function Name
1	`scanf`
2	`fprintf`
3	`fscanf`
4	`vfprintf`
5	`vfscanf`

const keyword

The `const` keyword is generally applied on variables that does not change their value over the execution time.

```
const int a = 4;
```

defines a constant int of 4. Sometimes a result of a mathematical calculation can also be a constant.

```
const double radius = 3;
const double circumferance = 2 * 3.14 * 3;
```

type definition

Any type can be type defined to another type. The keyword `typedef` is used for this purpose.

```
typedef int integer_t;
```

Now, `integer_t` can be used as a new type to declare variables. The `typedef` can be applied for many other data types such as structures and function pointers.

typecasting

Variables of one type can be typecasted to other variables.

The below program gives an example of typecast from integer to double.

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

```
int main()
{
    int d;
    double v = 10.1;

    d = (int)v;

    printf("d %d v %f\n", d, v);

    return 0;
}
```

operators

C has below operators that can be used on the variables of given types.

S.No	operator	meaning
1	+	addition
2	-	subtraction
3	*	multiplication
4	/	division
5	%	modulo
6	=	equals to
7	==	comparison operator
8		
9	&&	logical AND
10		
11	&	AND
12	ˆ	XOR
13	!	NOT
14	!!	Logical NOT
15	++	increment operator
16	–	decrement operator

Below example shows an example of the operators.

```
#include <stdio.h>

int main()
{
    int a = 4;
    int b = 2;
    int sum;
    int sub;
    int mul;
```

```c
    int div;
    int mod;

    sum = a + b; // add two numbers
    sub = a - b; // subtract two numbers
    mul = a * b; // multiply two numbers
    div = a / b; // divide two numbers
    mod = a % b; // modulo two numbers

    printf("sum %d sub %d\n", sum, sub);
    printf("mul %d div %d modulo %d\n", mul, div, mod);

    return 0;
}
```

Example.3 Operators example

The `++` and `--` are increment and decrement operators. Below example shows how to use them.

The boolean operations such as `|` , `&`, `||`, `&&`, `^` and `!` never apply to the `double` or `float` variables.

```c
#include <stdio.h>

int main()
{
    int i = 0;

    printf("i %d \n", ++ i);

    return 0;
}
```

Example.4 Pre increment operator

The above program prints the value 1.

Consider the another program.

```c
#include <stdio.h>

int main()
{
    int i = 0;

    printf("i %d\n", i ++);

    return 0;
}
```

Example.5 Post increment operator

The above program prints the value 0.

This is generally called the undefined behavior. The language leaves the behavior upto the compiler. The `++ i` used, this is called prefix notation and the `i ++` is the postfix notation.

In general, it is upto the programmer to choose `++i` or `i++` appropriately. However, choosing `++i` makes it less paranoid when debugging the software.

More usecases of `++` and `--` in `while` and `for`.

The `!!` statement is used to check the value of a number is non zero or zero. Below example shows how to use it.

```c
#include <stdio.h>

int main()
{
    int a = 4;
    int b = 0;

    printf("a=%d b=%d\n", !!a, !!b);

    return 0;
}
```

Example.6 Logical NOT operator

Below example shows the use of `&` `|` and `^` operators.

```c
#include <stdio.h>

int main()
{
    int a = 0x80;
    int b = 0x81;

    printf("AND 0x%02x OR 0x%02x XOR 0x%02x\n",
                a & b, a | b, a ^ b);

    return 0;
}
```

Example.7 Logical AND OR XOR operator

The output is :

```
AND 0x80 OR 0x81 XOR 0x01
```

Scope and Lifetime of the variables

We have declared many variables over the examples. Each variable declared within the function has some certain scope and lifecycle.

In the above function, the variables `a` and `b` are called local variables. This means that the scope of these variables are within the lifetime of the function. If the function returns the variables will be destroyed or removed from the memory.

Each program has a determined memory set aside by default by the operating system. This memory is dvidied into the following partitions.

1. stack segment
2. heap segment
3. data segment
4. text segment

Below is the diagrammatic representation.

```
|---------------------|
|                     |
|                     |
|---------------------|
|                     |
|  Stack segment      |     -> Stack grows downwards in memory
|                     |
|---------------------|
|        |            |
|        V            |
|                     |
|                     |
|        ^            |
|        |            |
|---------------------|
|  Heap Segment       |
|                     |   -> Heap grows upwards in memory
|---------------------|
| Uninitialized data  |
|---------------------|
| Initialized data    |
|---------------------|
|      text           |
|---------------------|
```

1. Stack segment

In linux stacks are of 8 kbytes in size by default. But it varies on operating system, sometimes application wise. In RTOS based systems, it is configured at the linker script.

Stack grow downwards in memory. Local variables and function arguments are part of the stack.

Everytime a function is called, the function arguments kept in the stack and a call is made and grows down. After the return, the stack is freed up automatically.

In general, the stack can exhaust quickly if allocation reaches maximum value, resulting in stack overflow. In some operating systems, it will result in entire system halt (such as in RTOSes running on a Microcontroller) or a program halt while the rest of the system execute normally.

In either cases, the result is a program abort. To avoid this, we must carefully use the stack memory. Sometimes data section can be of use if there is no heap section.

Lifetime of stack variables is only local to the function. This is in general one of the right approaches to programming by confining the scope to a function or to a group.

Below example shows the use of static variables confined to a function

```c
int F()
{
    int a = 3;

    return a * 2;
}
```

Here the variable `a` is a local variable and is part of the stack.

Below is another example of scoped variable setting with stack.

```c
int F()
{
    // variable a is scoped between { and }
    {
        int a = 3;

        a *= 3;
    }

    return a; // returns error that a is undeclared
}
```

GCC provides few compiler options to check the stack usage. See `GCC compiler options`.

2. Heap segment

This is the space where the dynamic memory can be allocated. This is explicitly allocated by calls to allocation functions in programming languages. C, C++ exposes allocation routines such as the following:

1. malloc
2. calloc
3. realloc
4. free
5. new
6. delete

The last two are in C++ standard library but it supports all of them. C library supports the first 4.

Once allocated by any of these calls, only way to free is by a call to free routines. There is no automatic freeing. Without freeing any heap memory, the system will soon run out of memory space if run for long duration.

Not freeing the heap results in memory leaks. When a program stops running, in general operating system will cleanup the memory allocated to it and returns back the memory back to the free pool. However, if the program has been running for a long time and never frees memory, it results in increase of memory usage and most probably leads to memory pressure / exhaustion.

Generally operating systems responds to the memory exhaustion by triggering spontaneous kills of programs that used high memory to restore the memory usage. In linux this is called out of memory killer in the kernel space.

The Lifetime of the heap section is entire program but the scope is limited to either global, local, or file scope. But the allocated memory is always part of the heap section.

3. Data segment

Data segment is initialized and uninitialized segments.

Here are few cases the variables are part of data segment.

- Variables that are accessible by all functions globally from all the source files.
- Variables that are accessible by all the functions local to the source file. Generally we prefix them with `static` to keep the scope bound to a single file.
- Variables that are prefixed with `static` and are accessible by a function locally.

The lifetime of these variables is the entire program.

The pointers can also be global.

Though the usage of globals and static globals result in easier programming but it results in a large mess of variables and their usage. This further results in confusion in case the program need to be updated. In cases, the global or static gllobals are required, it has to be considered and an alternative approaches to the use of globals must be considered.

4. Text segment

The text segment contain the program code , instructions. It is generally below the stack and heap to avoid overwriting. It is also readonly but most of the time an active attack could overwrite the text segment and execute an unknown program (also viruses, worms).

For example,

```c
char *msg = "hello";
```

The `msg` is allocated in the text segment. Changing any of the content at this location results in a fault.

```c
#include <stdio.h>

int main()
{
    char *ptr = "hello";

    *ptr = 'i';

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

    return 0;
}
```

The above program when ran, results in segmentation fault and program stops.

In general, the language provides some mechanisms to detect the modifications at compile time. For example when `const` used as the prefix to `char *ptr`, the compilation will fail noting that the read-only location is being modified.

```c
#include <stdio.h>

int main()
{
    const char *ptr = "hello";

    *ptr = 'i';

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

    return 0;
}

```bash
c/read_only.c: In function ‘main’:
c/read_only.c:7:10: error: assignment of read-only location ‘*ptr’
```
```

```
    7 |     *ptr = 'i';
      |           ^
```

Control statements

1. if else statement

The `if` statement provides a method of controlling the execution path of a program based on the data.

The `if` statement holds the condition and is evaluated for true or false. If the condition is true, then the statements in the `if` are executed, Otherwise the statements in `else` are executed.

The `else` statement in general is followed by the `if` statement and never alone. The `else` statement does not contain any condition test like the `if`.

```c
#include <stdio.h>

int main()
{
    char p = 'd';

    if (p == 'd') {
        printf("p '%c' is %c\n", p, 'd');
    } else {
        printf("p '%c' is not 'd'\n", p);
    }

    return 0;
}
```

A conditional check `if (p)` is also a valid check , but this check only check if the value is non zero. If the value is 0, then the statements under `if (p)` never executes. Understand that in C, the conditional checks depend on the value in the variable.

Generally the `else` statement does not have to follow the `if` statement. But it is a good practise to have an `else` statement if needed.

Let us consider the below program.

```c
#include <stdio.h>

int main()
{
    char p = 'd';

    if (p = 'f') {
        printf("p '%c' is %c\n", p, 'd');
```

```
    }

    return 0;
}
```

Notice the mistake?

The `if` conditional have the assignment operator than the equality test. This results in plain assignment which executes and so the `printf` inside the `if` will execute and prints the following:

```
p 'f' is d
```

We generally tend to avoid assignment statements inside the `if` to prevent getting into such kind of issues.

2. if else-if statement

The `if else-if` statement also called `if else ladder` that is used to construct a series of `if else if` conditions.

Below is an example of the `if else-if` ladder.

```
#include <stdio.h>

int main()
{
    int number = 50;

    if (number < 50) {
        printf("number is less than 50\n");
    } else if (number > 50) {
        printf("number is greater than 50\n");
    } else if (number == 50) {
        printf("number is equal to 50\n");
    } else {
        printf("number is [%d] unknown\n", number);
    }

    return 0;
}
```

In general `if else-if` ladders are not used in many large scale applications unless there are ranges involved. That is why the above example shows the use of `if else-if` with the ranges.

In most of the cases `if else-if` never ends with an `else` case, only few programming situations `else` case might be required.

For any direct comparision (`==`) the `switch` statement is used.

For example, the `&&` and `||` can be used within the `if` conditional.

Below is an example,

```
#include <stdio.h>

int main()
{
    int a = 0x80;
    int b = 0x0;

    if (a && b) {
        printf("a and b are non zero\n");
    } else if (a || b) {
        printf("a or b are non zero\n");
    }

    return 0;
}
```

3. Switch statement

The switch statement example is as shown below.

```
#include <stdio.h>

int main()
{
    int n = 50;

    switch (n) {
        case 10:
            printf("number is 10\n");
        break;
        case 20:
            printf("number is 20\n");
        break;
        case 50:
            printf("number is 50\n");
        break;
        default:
            printf("number [%d] is unknown\n", n);
        break;
    }
    return 0;
}
```

As you can see, the switch has a series of `case` statements and a `default` statement. Each of the `case` statement ends with a `break` statement if necessary. If there is no `break` statement then the statements fall through. Below example shows the descripton.

```c
#include <stdio.h>

int main()
{
    int n = 10;

    switch (n) {
        case 10:
        case 20:
            printf("n is %d\n", n);
        break;
    }

    return 0;
}
```

The output is :

```
n is 10
```

In some cases the fallthroughs are needed to have execute a series of statements for more than one case types.

The `switch` statement can also be used with characters. However, it cannot be used with strings. Strings are discussed more below. Below program is an example usage of the `switch` statement with character.

```c
#include <stdio.h>

int main()
{
    char p = 't';

    switch (p) {
        case 't':
            printf("value is t\n");
        break;
        default:
            printf("value is %c\n", p);
        break;
    }
}
```

In general, the `default` statement can be omitted. The `switch` statement must atleast have one `case`.

4. Trigraph ?: sequence

The `?:` is called a trigraph sequence. Here's how it can be used.

```
#include <stdio.h>

int main()
{
    int a = 10;
    int b = 5;
    int res;

    res = (a > b) ? a : b;

    printf("res %d\n", res);

    return 0;
}
```

Trigraphs are similar to the if else cases the true case is right after the `?` and the false case is right after the `:`.

They are mostly useful when writing simple test on a variable instead of the general `if else` conditionals.

Loops

1. While loop

The `while` loop allows to loop over a certain condition until it fails. An example of the `while` is as follows.

```
while (condition) {
    // statements
}
```

An example use of `while` loop is as follows.

```
#include <stdio.h>

int main()
{
    int i = 0;

    while (i < 10) {
        printf("i %d\n", i);
        i ++;
    }

    return 0;
}
```

In the above program the loop repeats until `i` reaches 10. Upon reaching 10, the while condition fails breaking the loop.

The `break` statement can be used in the `while` loop as well.

```
int main()
{
    int i = 0;

    while (1) {
        if (i >= 10) {
            break;
        }
        printf("%d\n", i);
        i ++;
    }

    return 0;
}
```

Above program shows the use of `while (1)`. Generally this means that the condition in the `while` loop is never false. It is an infinite loop.

Generally infinite loops are not preferable in programming without any conditional checks in the `while` statement.

The infinite loops generally do nothing but increase in CPU load on the process the program runs and consumes the CPU cycles unnecessarily. However, some programs written for the operating systems do need to run infinitely (such as graphics, display, editors etc). To do this, operating systems employ certain event based mechanisms supported by the hardware. This ensures that the program executes only based on certain events.

2. For loop

The `for` loop is similar to the `while` loop. The syntax is as follows,

```
for (initialization; condition; increment / decrement operation)
```

Below is an example of the use of `for` loop.

```
#include <stdio.h>

int main()
{
    int i;

    for (i = 0; i < 10; i ++) {
        printf("i %d\n", i);
    }
```

```
    return 0;
}
```

the `i = 0` statement in `for` executes only once. The `i < 10` statement executes everytime the loop repeats. The `i ++` statement executes everytime the statements in the `for` loop executes.

Another way to do is the following:

```
#include <stdio.h>

int main()
{
    int i = 0;

    for (;i < 10; i ++) {
        printf("i %d\n", i);
    }

    return 0;
}
```

The initializer statement can be left aside.

The above `while (1)` can be re-written with `for` as follows.

```
#include <stdio.h>

int main()
{
    int i = 0;

    for (;;) {
        if (i >= 10) {
            break;
        }
        printf("i %d\n", i);
        i ++;
    }

    return 0;
}
```

The `for(;;)` is also an infinite for loop. As mentioned, the infinite loops must be used with caution.

3. do while loop

The `do..while` loop is similar to the `while`. Below is an example.

```c
#include <stdio.h>

int main()
{
    int i = 0;

    do {
        printf("Hello World\n");
    } while (i != 0);

    return 0;
}
```

Once run, it prints `Hello World`. This means that the statements execute and the checks happen later.

4. Goto statement

The statement `goto` is similar to a jump instruction in assembly. The above loop can be rewritten with `goto` as follows.

```c
#include <stdio.h>

int main()
{
    int i = 0;

begin:
    if (i < 10) {
        printf("i %d\n", i);
        i ++;
        goto begin;
    }

    return 0;
}
```

We do not use `goto` in most of the programs for the following reasons:

1. Readability reduces with many gotos with in a function or within a C file.
2. Incorrectly written gotos can cause loops in program.

Gotos are not bad when used correctly in a program. For example in usecases when certain conditions fail during a program initialization, the deinitialization sequence must do the opposite. In such cases a jump required on the failure case.

Here's a pseudo code example,

```c
int init_1()
```

```c
{
    ...
    return 0;
}

int init_2()
{
    ...
    return 0;
}

void deinit_1()
{
    ...
}

int init_main()
{
    int ret;

    ret = init_1();
    if (ret != 0) {
        return -1;
    }

    ret = init_2();
    if (ret != 0) {
        goto deinit;
    }

deinit:
    deinit_1();
    return -1;
}
```

More about functions in the `functions` section.

In areas such as Automotive and Aerospace software application, `goto` statement is seldom used. It is treated as a bad practise. So avoiding this is a good step when writing software for such applications.

Arrays

1. One Dimensional Arrays

One dimensional array are the base type in arrays.

An array of integers is defined as,

```c
int a[10];
```

Above statement defines an array `a` of 10 integers. Each element in the array is an element of type integer.

Array indexes start from 0. Each item in the array is indexed with regular numbers ranging from 0 to 9.

Maximum elements in the above array are 10 but the last index of the 10th element is 9, not 10. Accessing the array beyond its maximum range is also called out of bounds access. Out of bounds accesses are major security problem as the element is accessing an address beyond the allocated range.

Below program assigns the elements in the array.

```c
#include <stdio.h>

int main()
{
    int a[10];
    int i = 0;

    for (i = 0; i < sizeof(a) / sizeof(a[0]); i ++) {
        a[i] = i;
    }

    printf("array elements:\n");
    for (i = 0; i < sizeof(a) / sizeof(a[0]); i ++) {
        printf("\ta[%d] = %d\n", i, a[i]);
    }
    printf("\n");
}
```

The size of an array is calculated the same way.

```c
#include <stdio.h>

int main()
{
    int a[10];

    printf("size of array %lu\n", sizeof(a));

    return 0;
}
```

With the `sizeof`, one can also find out the number of elements in the array as follows.

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

```c
int main()
{
    int a[10];

    printf("number of elements %d\n", sizeof(a) / sizeof(a[0]));

    return 0;
}
```

Initializing array elements

The below statement generally initializes the array.

```c
int a[10] = {0};
```

However, this initializes the first element to 0. Since only one element is initialized then by default all elements are initialized to 0.

So if we have initialized it,

```c
int a[10] = {10};
```

the first element of the array is initialized to 10 and the rest of the elements are initialized as 0s. Below is one example:

```c
#include <stdio.h>

int main()
{
    int a[10] = {10};
    int i;

    for (i = 0; i < sizeof(a) / sizeof(a[0]); i ++) {
        printf("a[%d] = %d\n", i, a[i]);
    }

    return 0;
}
```

This example prints the first element as 10 and rest as 0.

General way sometimes tend to be the use of `memset` which is discussed in below sections. But the below example shows how to initialize an array.

```c
int a[10];

memset(a, 0, sizeof(a));
```

Sets all the elements of the array `a` to 0.

Another way to set array elements is as follows:

```c
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int i;

for (i = 0; i < sizeof(a) / sizeof(a[0]); i ++) {
    printf("a[%d] = %d\n", i, a[i]);
}
```

But this means that all array elements must be initialized which is impractical for a large set of arrays.

2. Two Dimensional Arrays

Two dimensional arrays are represented as follows.

```c
int a[10][10];
```

denotes a two dimensional array.

To access the elements one must iterate both indexes with two loops. Or we call them here the nested loops.

```c
#include <stdio.h>

int main()
{
    int a[10][10];
    int i;
    int j;

    for (i = 0; i < 10; i ++) {
        for (j = 0; j < 10; j ++) {
            a[i][j] = j + i;
        }
    }

    for (i = 0; i < 10; i ++) {
        for (j = 0; j < 10; j ++) {
            printf("a[%d][%d] = %d\n", i, j, a[i][j]);
        }
    }

    return 0;
```

3. Three Dimensional Arrays

A 3 dimensional array is a group of 2-D arrays. Thet are denoted as follows,

```c
int array[10][20][40];
```

The below example shows how a 3-D array is used.

```c
#include <stdio.h>

int main()
{
    int array[10][20][40];
    int i;
    int j;
    int k;

    for (i = 0; i < 10; i ++) {
        for (j = 0; j < 20; j ++) {
            for (k = 0; k < 40; k ++) {
                array[i][j][k] = 10 + i;
            }
        }
    }

    for (i = 0; i < 10; i ++) {
        for (j = 0; j < 20; j ++) {
            for (k = 0; k < 40; k ++) {
                printf("array[%d][%d][%d] = %d\n", i, j, k, a[i][j][k]);
            }
        }
    }

    printf("\n");
    return 0;
}
```

The usecases of arrays range in many places, such as grouping a set of characters together, or grouping of similar types together to represent a hardware device or group of same hardware devices and so on. Also arrays play a big role in search and sort techniques.

Macros

Macros are compile time constants and do not allocate any space in runtime. This means that the preprocessor replaces the sections where Macros are used, into their corresponding values.

In the above example we used `#include <stdio.h>`, where `#include` is a directive. This informs the compiler to replace this statement with the header file `stdio.h`.

#define macro

The preprocessor stage, replaces the macros with the actual values present in the macro definition.

The macro `#define` defines macro constants. An example is as follows.

```
#define TWO 2
```

Macro statements are used as substitutes for constants or some common operations.

For example using,

```
#define PI 3.1413

double circumferance(double radius)
{
    return 2 * PI * radius;
}
```

is more meaningful that using a constant value.

```
#define ADD(_a, _b) ((_a) + (_b))
```

is an operation. We discuss about function like macros in the below sections.

#undef macro

The `#undef` macro undefines a macro. For example,

```
#define PI 3.1413
#undef PI

double circumferance(double radius)
{
    return 2 * PI * radius;
}
```

results in compiler error as the preprocessor removes `PI`.

#ifdef macro

The `#ifdef` macro checks if a certain macro is defined and if so preprocessor enables the portion of code that is between the `#ifdef` and `#endif`. The `#ifdef` is always followed by `#endif` or `#else` statement.

For example,

```
int a = 10;
#ifdef CONFIG_MACRO
a = 5;
#else
a = 6;
#endif
```

Will set a to 5 if `CONFIG_MACRO` is defined or 6 otherwise.

Below example shows how to define the macro.

```
#include <stdio.h>

#define CONFIG_MACRO

int main()
{
    int a = 10;

#ifdef CONFIG_MACRO
    a = 5;
#else
    a = 6;
#endif

    printf("%d\n", a);
}
```

Defining a simple `#define CONFIG_MACRO` is enough to enable the statements under the `#ifdef`. In general these can be passed as command line arguments instead to the compiler. For example,

```
gcc -DCONFIG_MACRO macro.c
```

The argument `-D` to the `gcc` accepts the macros and many number of -Ds can be given as arguments.

The `#ifndef` macro is a negation of `#ifdef` where if the particular macro variable is not defined, preprocessor will enable that portion of the code.

Functions

Function in a C program is a group of instructions. Functions allow us to break a large program into pieces of understandable segments. Each segment with some defined business logic and logical implementation.

A function has none, one or more input arguments and returns or do not return anything. A prototype is as follows,

```
void function(void)
```

or more generally,

```
return-type function-name(arguments, ..).
```

For example,

```
#include <stdio.h>

void print_hello()
```

```c
{
    printf("Hello World\n");
}

int main()
{
    print_hello();
}
```

The above example calls a function called `print_hello` to print the “Hello World” on screen.

The call to the function is simply by writing its name followed by the parantheses with none, one or more arguments and followed by a semicolon.

This function does not accept any argument and does not return anything.

The statements,

```c
void print_hello()
{
    printf("Hello World\n");
}
```

comprise the body of the function. Its also referred as function definition. A program typically contains more than one function.

`main()` is also a function which is the starting point of a program.

If the `main()` is not defined, then the compilation results in undefined error. In general the linker expects `main()` to be defined as it is expected by the sequence before it calling `main()`. Who calls `main()`? For this, the short answer is the call is defined somewhere in the `libc`.

A function can only have one signature in C (A function can have many signatures in C++). `main()` prototypes are many unlike many other functions. Some of the most used prototypes are as follows.

`int main(void)`

`int main(int argc, char **argv)`

`void main(void)` -> however this is seldom used when writing software. `main()` must return. This is to let the executing shell to know the status of the program when returned. The shell can use this status to further perform certain operations. (example is that shell scripts can use the return status of a program).

The second prototype is further described in command line arguments section.

A function can return any data type or none. For example,

```c
int f(void);
```

The above function returns integer type but accepts no arguments.

A function can take one or more arguments and return none or one type. For example,

```
int f(double a, double b);
```

The above function accepts two variables of type `double` and returns an integer.

A small example shows the usecase.

```
#include <stdio.h>

int add(int a, int b)
{
    return a + b;
}

int sub(int a, int b)
{
    return a - b;
}

int mul(int a, int b)
{
    return a * b;
}

int div(int a, int b)
{
    return a / b;
}

int main()
{
    int a = 10;
    int b = 5;

    printf("add %d\n", add(a, b));
    printf("sub %d\n", sub(a, b));
    printf("mul %d\n", mul(a, b));
    printf("div %d\n", div(a, b));

    return 0;
}
```

The above program creates and calls 4 functions `add`, `sub`, `mul`, and `div` from `main()`.

Each function does exactly one job and named as per the job it does to help reader of the program understands. This is usually the discipline that is followed when writing software.

Each function returns only integers, but the program does not work correctly (rounding off errors) when given floating point numbers. Writing such generic functions are explained more in C++ templates.

Variadic functions

Variadic functions are the ones that can accept infinite arguments as input to them.

The header `stdarg.h` contains the macros and helper functions to make variadic functions possible.

There are 2 important functions / macros in `stdarg.h`.

```
va_start
va_end
```

Libc also provide few more variadic functions that takes in the arguments.

writing your own printf

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

void log_printf(const char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    vfprintf(stderr, fmt, ap);
    va_end(ap);
}

int main()
{
    int a = 10;
    log_printf("test: test message a=%d\n", a);
}
```

Function like macros

Since we read about macros above, we can rewrite the above functions as follows.

```
#define add(_a, _b) ((_a) + (_b))
#define sub(_a, _b) ((_a) - (_b))
#define mul(_a, _b) ((_a) * (_b))
```

```
#define div(_a, _b) ((_a) / (_b))

int main()
{
    int a = 10;
    int b = 5;

    printf("add %d\n", add(a, b));
    printf("sub %d\n", sub(a, b));
    printf("mul %d\n", mul(a, b));
    printf("div %d\n", div(a, b));

    return 0;
}
```

inline functions

Strings

The below statement is a base char type.

```
char d;
```

String is an array of characters ending with `\0`. For example, the below statement defines a base string type.

```
char d[20];
```

declares a string of 19 elements with the last element allocated to the `\0`.

String manipulation operations

The header file `string.h` contains the functions that help to manipulate the string data.

String manipulation functions The standard library provides below or more of the functions to manipulate the strings.

1. strlen

The library function `strlen` is used to get the length of a string.

The `strlen` prototype is as follows:

```
int strlen(const char *str);
```

Below is an example,

```
#include <stdio.h>

int main()
```

```c
{
    char str[] = "hello world";

    printf("strlen = %d\n", strlen(str));

    return 0;
}
```

The `strlen` function counts all the characters in the string excluding the `\0` marker.

It can be implemented as follows.

```c
int string_length(const char *str)
{
    int i = 0;

    for (i = 0; str[i] != '\0'; i ++);

    return i;
}
```

The above program iterates over each character in the string `str` until the character is `\0`. So the content of the for loop does nothing. So we end it with a semicolon. Sometimes open and closed braces (`{` and `}`) can be used as well.

Note that the behavior is that the last character should always be null terminated. If there are characters present beyond the null terminating character then they will be ignored.

If there is no null character in the string, then the `strlen` function will keep on reading and may even read past the allocated buffer in some cases. This sometimes causes crashes if lucky and sometimes executes other code if not lucky leading to exploits.

So caution must be taken when creating and using the strings. Always null terminate the strings.

Note that the `strlen` and `string_length` functions never check if the input is a `NULL`. The callers must take care of the pointer validity before calling `strlen`.

2. strcpy

The `strcpy` function is used to copy the source string into the destination.

Its prototype is as follows.

```c
char *strcpy(char *dst, const char *str);
```

The example of `strcpy` is as follows.

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

```c
int main()
{
    char *str1 = "Witcher";
    char str2[20];

    strcpy(str2, str1);

    printf("str1 [%s] str2 [%s]\n", str1, str2);

    return 0;
}
```

strcpy can be implemented as follows.

```c
int string_copy(char *dst, unsigned int dst_len, const char *src)
{
    uint32_t i = 0;
    uint32_t len = 0;

    for (i = 0; src[i] != '\0'; i ++) {
        if (i < dst_len) {
            dst[i] = src[i];
        } else {
            break;
        }
    }

    dst[i - 1] = '\0';

    return 0;
}
```

3. strcat

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

int main()
{
    char *str = "hello";
    char *ptr = "mangoes";
    char dst[20];

    strcpy(dst, str);
    strcat(dst, " ");
    strcat(Dst, ptr);
```

```c
    return 0;
}
```

```c
#include <stdio.h>

int string_cat(char *dst, const char *src)
{
    int i = 0;
    int j = 0;

    while (dst[i] != '\0') {
        i ++;
    }

    while (src[j] != '\0') {
        dst[i] = src[j];
        i ++;
        j ++;
    }
    dst[i] = '\0';

    return j;
}

int main()
{
    char *src = "test";
    char dst[30] = "dest";

    string_cat(dst, src);

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

    return 0;
}
```

4. strcmp

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

int main()
{
    char *str = "hello";
    char *str_1 = "hello";
    char *str_2 = "Hello";
```

```c
    int res_1, res_2;

    res_1 = strcmp(str, str_1);
    res_2 = strcmp(str, str_2);

    printf("res_1 %d res_2 %d\n", res_1, res_2);

    return 0;
}


int string_len(const char *str)
{
    int i = 0;

    while (str[i] != '\0') {
        i ++;
    }

    return i;
}

int string_cmp(const char *str1, const char *str2)
{
    int i = 0;

    if (string_len(str1) != string_len(str2)) {
        return -1;
    }

    while (str1[i] != '\0') {
        if (str1[i] != str2[i]) {
            return str1[i] - str2[i];
        }
    }

    return 0;
}
```

5. strchr

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

int main()
{
    char *str = "english movies";
```

```c
    char *pos;

    pos = strchr(str, 'm');
    if (pos) {
        printf("pos '%s'\n", pos);
    }

    return 0;
}
```

6. memcmp

The function `memcmp` compares data in the two memory locations given the size.

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>

int main()
{
    uint8_t buf1[] = {0x01, 0x02, 0x03, 0x04, 0x01, 0x04, 0x4, 0x3};
    uint8_t buf2[] = {0x01, 0x02, 0x03, 0x04, 0x01, 0x04, 0x4, 0x3};

    printf("compare %d\n", (memcmp(buf1, buf2, 8) == 0));

    return 0;
}
```

A simpler `memcmp` implementation would involve comparing byte by byte to length bytes. Below is an example:

```c
int memcmp(const void *a, const void *b, int len)
{
    int ret = 0;
    int i;

    const uint8_t *val_a = a;
    const uint8_t *val_b = b;

    if (!a || !b) {
        return -1;
    }

    for (i = 0; i < len; i ++) {
        if (val_a[i] != val_b[i]) {
            ret = -1;
            break;
        }
```

```
    }

    return 0;
}
```

Another way to implement is to compare 4 or 8 bytes at a time. This can be done based on the architecture. Below is an example:

```
int memcmp(const void *a, const void *b, int len)
{
    int ret = 0;
    int i;
    int rem = len % 8;
    int check_len = len / 8;

    const uint64_t *val_a = mem_a;
    const uint64_t *val_b = mem_b;

    const uint8_t *ptr_a = mem_a;
    const uint8_t *ptr_b = mem_b;

    for (i = 0; i < check_len; i ++) {
        if (val_a[i] != val_b[i]) {
            ret = -1;
            break;
        }
    }
    if (ret != -1) {
        if (rem != 0) {
            for (i = len - rem; i < len; i ++) {
                if (ptr_a[i] != ptr_b[i]) {
                    ret = -1;
                    break;
                }
            }
        }
    }

    return ret;
}
```

Here's the pointers taken are 8 byte and the direct comparison between 8 bytes is performed.

The input is typecasted to the 8 byte unsigned integer and the direct value comparisons are made 8 bytes at a time instead of 1 byte at a time.

Since the length may not always be aligned to 8 bytes, the remainder bytes must be compared after the first comparison.

For example if the length is 15, then the last 7 bytes must be compared byte wise.

The second example of `memcmp` generally results in higher performance on architecture where the 8 byte comparison instructions are available.

Generally these micro optimizations does not really have a greater benefit if they are not operated on a large sets of data such as comparing a buffer of over 1Megabyte or so.

7. memcpy

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>

int main()
{
    uint8_t buf1[] = {0x01, 0x02, 0x04, 0x02, 0x03, 0x05, 0x03, 0x03};
    uint8_t buf2[8];
    int i;

    memcpy(buf2, buf1, sizeof(buf1));
    for (i = 0; i < sizeof(buf1); i ++) {
        printf("%02x\n", buf2[i]);
    }

    return 0;
}
```

8. strdup

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

int main()
{
    char *ptr = "hello";
    char *ptr1;

    ptr1 = strdup(ptr);

    printf("ptr %s ptr1 %s\n", ptr, ptr1);

    free(ptr1);
```

```
    return 0;
}
```

Below is one of the implementation of `strdup`.

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

int len(const char *str)
{
    int i;

    for (i = 0; str[i] != '\0'; i ++) { }

    return i;
}

char *strdup_1(const char *str)
{
    char *ptr;
    int i;
    int length = len(str);

    // 1 for the \0
    ptr = calloc(1, length + 1);

    for (i = 0; i < length; i ++) {
        ptr[i] = str[i];
    }
    ptr[i] = '\0';

    return ptr;
}

int main()
{
    char *ptr = "hello";
    char *ptr1;

    ptr1 = strdup_1(ptr);

    printf("ptr %s ptr1 %s\n", ptr, ptr1);

    free(ptr1);

    return 0;
}
```

String to integer / double conversion

Below are few functions that convert string to other types. Below are the following.

1. `atoi`
2. `strtol`
3. `strtod`
4. `strtoul`

1. atoi

The standard library function `atoi` converts a given input string to integer. It is declared in `stdlib.h`.

Below is one example:

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

int main()
{
    char *str = "4";

    printf("%d\n", atoi(str));
}
```

Lets see another example:

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

int main()
{
    char *str = "4a";

    printf("%d\n", atoi(str));
}
```

This results in value `0`, in this case the value `0` is still legit if its taken as input. Since the input is not known. This can result in ambiguity if the result is right or wrong.

So the preference is generally not to use `atoi` when writing software.

There is another way to convert the string to integer.

```
#include <stdio.h>

int main()
{
    char *str_int = "1343";
```

```
    int intval;

    sscanf(str_int, "%d", &intval);
    printf("%d\n", intval);

    return 0;
}
```

We are using `sscanf` to read the input from the buffer into an integer.

Lets consider an invalid string input,

```
#include <stdio.h>

int main()
{
    char *str = "123a";
    int intval;
    int ret;

    ret = sscanf(str, "%d", &intval);
    if (ret != 1) {
        printf("incorrect integer\n");
    } else {
        printf("val %d\n", intval);
    }
}
```

This results in `ret` being 123. However, results in no error and the integer is still read.

2. strtol

The standard library function `strtol` converts a given input string to integer. It is delcared in `stdlib.h`.

The function prototype is as follows:

```
long strtol(const char *in, char **err, int dec_or_hex);
```

In the above function the `err` argument describes if the input is incorrect. This is checked to find out if the returned converted `long` value is legit.

The argument `dec_or_hex` is basically `10` if the input `in` is decimal or `16` if the input is hexadecimal in format `0x1`.

Below is one example:

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

int main()
```

```c
{
    char *number = "102";
    char *err = NULL;
    long num_long;

    num_long = strtol(number, &err, 10);
    if (err && (*err != '\0')) {
        printf("failed to parse number\n");
        return -1;
    }
    printf("num_long %d\n", num_long);

    return 0;
}
```

3. strtod

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

int main()
{
    char *str = "3.3";
    char *err = NULL;
    double val;

    val = strtod(str, &err);
    if (err && (err[0] != '\0')) {
        return -1;
    }

    printf("val %f\n", val);

    return 0;
}
```

4. strtoul

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

int main()
{
    char *str = "4294967295";
    uint32_t val;
    char *err = NULL;
```

```c
    val = strtoul(str, &err, 10);
    if (err && (err[0] != '\0')) {
        return -1;
    }

    printf("val %u\n", val);

    return 0;
}
```

Pointers

Strings can also be initialized with a pointer.

The below statement is a string that is allocated at compile time and the `str` is a pointer to the beginning of the string "Hello".

```c
char *str = "Hello";
```

A pointer of any type is possible.

```c
int *p;
```

declares an integer pointer.

```c
int val = 4;
int *v = &val;
```

declares an integer `val` and a pointer `v` holding the address of the variable `val`. The `&` denotes the address when placed before the variable.

Pointers can be printed with `%p` format specifier.

```c
#include <stdio.h>

int main()
{
    int val = 4;
    int *v = &val;

    printf("%d %p\n", val, v);
}
```

A size of a pointer can be evaluated as following.

```c
int *v;
int size = sizeof(v);
```

On a 64-bit machine, the size results in 8 bytes.

Pass by value and Pass by reference in functions

Consider the below example,

```
void add(int a, int b, int r)
{
    r = a+ b;
}

int main()
{
    int a = 3;
    int b = 3;
    int r = 0;

    add(a, b, r);

    printf("r %d\n", r);
}
```

Here the function `add` takes `a`, `b` and `r` as inputs. The `add` function perform the addition operation and writes the result in `r`.

Once the function executes and returns, the value of `r` is still 0. This is because the variable `r` when passed is local to the function `add`. So the result value of `r` in function `add` is not passed back.

One way to pass back the value is to return it. For example,

```
int add(int a, int b)
{
    return a + b;
}
```

And then capture the return value in the caller.

This method of passing arguments is generally called as Pass by value. In this approach, the value of the passed arguments do not change in the caller.

There is another approach to do this by using pointers. Refer to the pointers section on using pointers.

```
void add(int a, int b, int *r)
{
    *r = a + b;
}
```

This method is called as Pass by Reference. The `r` variable above is passed as a pointer. So in the caller we need to pass the address of the variable.

...

```
add(a, b, &r);
...
```

Here we passed the address of the variable `r` so the actual address that the `add` function is writing is in the original address of `r`.

This is particularly useful when functions want to change some information about the variables that they take as inputs instead of returning.

The void pointer

The void pointer is a generic pointer that can be assigned as an address to any structure, pointer or a variable. Below is one example:

```
int a[10];
void *p;

p = &a[0];
```

The void pointer cannot be dereferenced because dereferencing involve deducing the type it points, since its void the compiler wouldn't know which type it has to decode. So a typecast is required or in some cases assignment back to its type.

Pointers and Arrays A Pointer to an array can be simply assigned as follows.

```
int a[10];
int *p;

p = a;
```

or `p = &a[0]`.

The pointer `p` assigned as the pointer to the first element of the array.

```
#include <stdio.h>

int main()
{
    int a[10];
    int *p;
    int i;

    for (i = 0; i < sizeof(a) / sizeof(a[0]; i ++) {
        a[i] = i;
    }

    p = a;

    for (i = 0; i < sizeof(a) / sizeof(a[0]); i ++) {
        printf("a[%d] = %d\n", i, p[i]);
```

```
    }
    printf("\n");

    return 0;
}
```

Below is another example of accessing array elements with a pointer. The elements of array are updated with the pointer.

```
#include <stdio.h>

int main()
{
    int a[10];
    int i;
    int *p;

    for (i = 0; i < sizeof(a) / sizeof(a[0]); i ++) {
        p = &a[i];
        *p = i;
    }

    for (i = 0; i < sizeof(a) / sizeof(a[0]); i ++) {
        printf("%d\n", a[i]);
    }

    return 0;
}
```

Dynamic Memory Allocation

Dynamic memory is the runtime memory required by the program. Some programs require dynamic memory and some do not. It depends on the usecase as to why few programs require the dynamic memory to be allocated.

Dynamic memory is always allocated in the heap section of the program.

The Operating System provides a system call layer that allows the C library to make a call to the operating system to get the memory allocated for the program. Generally the memory returned is not contiguous but is sparsed. This behavior will be different for different operating systems.

Though the memory specific system calls return the memory, the C library functions keeps a note of memory asked for book keeping or for efficiency.

1. malloc

The `malloc` function is a C function that is used to allocate memory dynamically. It is declared in `stdlib.h`. The memory allocated by `malloc` may contain the old data that is used by other programs.

The prototype is as follows:

```
void *malloc(int size);
```

The example code is as follows.

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

int main()
{
    int *a;

    a = malloc(sizeof(int));
    *a = 4;

    printf("a %p *a %d\n", a, *a);

    return 0;
}
```

`malloc` makes a call to the underlying operating system call to allocate the heap memory.

Once the memory is allocated, it can only be freed with the `free`.

The call to the function `malloc` can fail and it results in a `NULL` pointer. In the example above, we are directly dereferencing `a` without checking if it can be a `NULL` pointer.

Dereferencing a `NULL` pointer results in faults and the program will be aborted.

The reason of returning `NULL` is that if the operating system is under high memory pressure (that is most of the memory is being used), in such cases, the operating system cannot allocate anymore memory available. This results in being returning a `NULL` pointer.

So, the best practises of using allocated pointers is to always check for `NULL`.

Below is another example of `malloc`:

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

int main()
{
    int *a;
    int i;

    a = malloc(sizeof(int) * 10); // allocate space for 10 integers
```

```
    if (!a) {
        return -1;
    }

    for (i = 0; i < 10; i ++) {
        a[i] = i;
    }

    for (i = 0; i < 10; i ++) {
        printf("a[%d] = %d\n", i, a[i]);
    }

    return 0;
}
```

In the above example, the pointer `a` is allocated with 10 elements and is being accessed just like an array.

2. calloc

The `calloc` function is a C function that is used to allocate memory dynamically. It is declared in `stdlib.h`. Once the memory is allocated, it is cleared and returned to the caller.

The prototype is as follows:

```
void *calloc(int n_elements, int size);
```

The example code is as follows.

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

int main()
{
    int *a;

    a = calloc(1, sizeof(int));
    *a = 4;

    printf("a %p *a %d\n", a, *a);

    return 0;
}
```

3. realloc

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

```c
int main()
{
    int *a;
    int i;

    a = calloc(4, sizeof(int));
    if (!a) {
        return -1;
    }

    for (i = 0; i < 4; i ++) {
        a[i] = i;
    }

    printf("calloced a %p\n", a);

    a = realloc(a, sizeof(int) * 10);
    if (!a) {
        return -1;
    }

    printf("realloced a %p\n", a);

    for (i = 4; i < 10; i ++) {
        a[i] = i;
    }

    for (i = 0; i < 10; i ++) {
        printf("a[%d] = %d\n", i, a[i]);
    }

    free(a);

    return 0;
}
```

4. free

Every memory allocation must be freed, otherwise the system will run out of the memory. This is called memory leak. Below is one example of the free.

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

int main()
{
    int *a;
```

```
    a = malloc(sizeof(int));
    if (a) {
        *a = 4;

        printf("a %p *a %d\n", a, *a);
        free(a);
    }

    return 0;
}
```

2. Allocating and freeing array

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

int main()
{
    int *a;
    int i;

    a = malloc(10 * sizeof(int));
    for (i = 0; i < 10; i ++) {
        a[i] = i;
    }

    for (i = 0; i < 10; i ++) {
        printf("a[%d] = %d\n", i, a[i]);
    }

    free(a);

    return 0;
}
```

Recap about variables and scope

Local variables

Global variables

1. static type

2. auto type

The type `auto` does not signify anything in C. This type is very significant however in C++.

3. volatile type

The keyword `volatile` is used in places where there is a real address being used. It is attached generally to integers.

```
volatile uint32_t *addr = 0xA0000000;
```

Lets look at an example where a register in the memory needs to be checked continuously until it reaches a certain value.

```
while ((*addr & 0x80) == 1) {
    ...
}
```

The `addr` is not set by the software but by the hardware.

Without the use of the `volatile` keyword, the compiler would simply return without executing anything in the inner loop.

Function Pointers

Structures

Data structure is a group of variables of different types. The `struct` word is used as an identifier to the compiler to make it recognize the structure.

An example of data structure looks as follows.

```
struct shelf {
    char book_name[10];
    int n_papers;
};
```

Above structure defines a shelf that contains a list of books and papers, one is a string and another is an integer.

Defining the structure variable is similar to defining the base type.

```
struct shelf s;
```

here `s` is of type structure `shelf`.

Accessing the elements in the structure is via the `.` operator.

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

struct shelf {
    char book_name[10];
    int n_papers;
};

int main()
```

```
{
    struct shelf s;

    strcpy(s.book_name, "Witcher");
    s.n_papers = 2000;

    printf("book_name: %s papers: %d\n", s.book_name, s.n_papers);

    return 0;
}
```

A structure can be inside another structure as well.

```
struct book {
    char book_name[10];
    char book_author[10];
};

struct shelf {
    struct book book;
    int n_papers;
}
```

We can apply the typedefs to structures as well. such as,

```
typedef struct book book_t;
```

Now `book_t` can be used to define structure variables. Generally `_t` prefix is used to differentiate the typedefs. But its only a choice of programmer and not defined by the C standard.

The elements are accessed as follows.

```
struct shelf s;

void set_book(struct book *b, char *book_name, char *book_author)
{
    strcpy(b->book_name, book_name);
    strcpy(b->book_author, book_author);
}

void set_shelf(char *book_name, char *book_author, int n_papers)
{
    s.n_papers = n_papers;
    set_book(&s.b, book_name, book_author);
}
```

The variable `b` of the type `struct book` is passed as a pointer to the `set_book`. The function `set_book` sets the book_name and book_author.

An array of structures is possible too.

```
struct book {
    char book_name[10];
    char book_author[10];
};

struct shelf {
    struct book books[10];
    int n_papers;
}
```

Pointers in Structures

Structures can contain pointers as well.

```
struct book {
    char *book_name;
    char *book_author;
};
```

These are allocated just the way pointers are allocated.

```
book->book_name = calloc(1, 40);
book->book_author = calloc(1, 20);
```

There can be cases when there are structures in a structure which contain pointers to another structures or variables. In such cases freeing all the structures becomes a real problem. Doing it in the same order of allocation guarantees no crashes but leaks memory. To do the right way, one must free in the reverse order of allocation.

Below example shown provides such a method.

TBD

Structure Initialization

Structures initialization can be done in many ways. One way of initialization is as follows.

```
struct A {
    int val;
    double val_d;
};

struct A a = { .val = 3, .val_d = 3.1 };
```

Is one way to initialize the structure.

Below is an example,

```c
#include <stdio.h>

struct A {
    int val;
    double val_d;
};

int main()
{
    struct A a = { .val = 3, .val_d = 3.1 };

    fprintf(stderr, "val: %d\n", a.val);
    fprintf(stderr, "val_d: %f\n", a.val_d);

    return 0;
}
```

Array of structures

Arrays of structures is possible as well.

```c
struct A {
    int val;
    double val_d;
};

struct A a[10]; // array of structures of type `A`.
```

They can be iterated just like arrays.

```c
struct A {
    int val;
    double val_d;
};

void set(struct A *a, int size)
{
    int i;

    for (i = 0; i < size; i ++) {
        a[i].val = i;
        a[i].val_d = i + 0.1 * i;
    }
}
```

Function pointers in structures

The below structure declares two function pointers `get` and `set` which are accessible from the structure variable.

```
struct S {
    int (*get)();
    void (*set)(int);
};

struct S s;

s.set(3); // set the variable
int var = s.get(); // get the variable
```

But in general the pointers `s.get` and `s.set` contain garbage pointers. So accessing them generally results in a segmentation fault or in bad situation results in abnormal program execution.

One way to assign the addresses is the following:

```
int a;
int my_get() { return a; }
void my_set(int A) { a = A; }

struct S s;

memset(&s, 0, sizeof(struct S));

s.get = my_get;
s.set = my_set;
```

Now accessing the function pointers `s.get` and `s.set` will indirectly call `my_get` and `my_set` functions.

Structure pointers have a wide range of usecases. One of such use cases is writing abstractions.

The structure can have a reference of its own as the element. This is particularly advantageous to implement data structures.

Below is one example of such structure.

```
struct S {
    int v;
    struct S *next;
};
```

The section "Data Structures" describe more on the usecases.

Typecasting from structure pointers to void pointers and assignment from structure pointer to void pointer is possible.

```
struct S *s;
void *v;
```

```c
s = calloc(1, sizeof(struct S));
v = s; //valid

s = (struct S *)v; // valid
s = v; // may produce warning that conversion from void * to struct

printf("val %d\n", v->v); // still incorrect.. dereferencing the void pointer
                          // is still incorrect
```

Bit fields

Structure padding and packing

Structure Packing

The compilers provide a mechanism to pack the elements of the structure to close any holes created due to the padding / alignment.

Gcc provides `__attribute__ ((__packed__))` specifier.

```c
struct S {
    int v;
    int p;
    char r;
} __attribute__ ((__packed__));
```

The `sizeof` on above structure returns 9. Supposedly it should have 12 without packed attribute.

Below is one example:

```c
#include <stdio.h>

struct S {
    int v;
    int p;
    char r;
} __attribute__ ((__packed__));

struct R {
    int v;
    int p;
    char r;
};

int main()
{
    printf("regular_size %ld packed_size %ld\n", sizeof(struct R), sizeof(struct S));
```

```
    return 0;
}
```

Enumeration

Enumerations in C are similar to macros. An example of an enum is as shown below.

```
enum army {
    Snipers,
    Medics,
    Seals,
};
```

The elements in the enum are accessed the following way.

```
enum army sniper_team;

sniper_team = Snipers;
```

The first element of the enum always start with 0 when uninitialized.

The elements in the `enum` can be initialized with values too. Lets see the following definition.

```
enum army {
    Snipers = 10,
    Medics,
    Seals,
};
```

The rest of elements are incremented by 1 every time. For example, Medics will become 11 and Seals will become 12.

The below declaration is valid as well.

```
enum army {
    Snipers = 10,
    Medics = 12,
    Seals,
}
```

The enumeration `Seals` now becomes 13.

The enumerations are like integers. If assigned a `double` value to an enumeration, results in compiler error.

The size of the enum can as well be calculated with `sizeof`.

```
#include <stdio.h>

enum army {
```

```
    Snipers,
    Medics,
    Seals,
};

int main()
{
    printf("size %lu\n", sizeof(enum army));

    return 0;
}
```

We can have a pointer to the enum as well.

```
enum army {
    Snipers,
    Medics,
    Seals,
};

enum army charlie_1, *charlie_2;

charlie_2 = &charlie_1;
```

is perfectly valid.

But most of the cases we never use enum pointers as they occupy only few bytes (4 or 8).

Unions

Appendix A

Significance of header files

Header files can include structures, macro definitions, and function prototypes. Sometimes they even contain inline functions.

Here are few advantages:

1. They can be used to group related functionality. Provides structure to the program when separated well.
2. They can be used to expose the prototypes, macro definitions and structures to other files. The other files can use these prototypes to avoid implicit calls.
3. Best way to distribute the software in package of header files and library `.so` or `.a` files.

Header description

The `stdio.h` contains prototypes for `printf`, `scanf`, `fprintf`, `fscanf`, `fopen`, `fclose`, `fgets` and so on. The `stdint.h` has further more data types. See `/usr/include/stdint.h` The `limits.h` contains all the ranges of the base types. See `/usr/include/limits.h`. The `stdlib.h` contains the prototypes for `atoi`, `malloc`, `calloc`, `realloc` and `free`. The `string.h` contains the prototypes for string related functions such as `strlen`, `strcpy`, `strcat`, `strcmp` and so on.

Compilation of C program

GCC compilation options

S.No	Name	Description
1	`-Wall`	Enable all warnings

Valgrind

Command line arguments (argc, argv)

The function `main` accepts two more arguments `argc` and `argv`. The signature looks as follows.

```
int main(int argc, char **argv);
```

The first argument is the number of arguments and the second argument contain the actual number of arguments.

Below is one example,

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int i;

    for (i = 0; i < argc; i ++) {
        printf("arg[%d]: %s\n", i, argv[i]);
    }

    return 0;
}
```

Compile and run the above program with the following arguments:

```
./a.out 1 2 3

arg[0]: ./a.out
arg[1]: 1
```

```
arg[2]: 2
arg[3]: 3
```

The `argv[0]` is always the program name with full path. The rest of the arguments are the ones given after `./a.out` separated by space.

`argc` provides the number of arguments in `argv`.

getopt The library function `getopt` provides a way to parse command line arguments. Its prototype is defined in `getopt.h`.

```
int getopt(int argc, const char *argv[], const char *optstring);
```

The options are passed in `optstring`. For example the user can enter options as follow as input to the command.

```
./a.out -f filename -r
```

The optstring here will be `f:r`.

- `f:` means that the option `f` will have an argument.
- `r` means that the option `r` will not have an argument, this is like a `true` or `false` statement.

There can be as many number of arguments as possible via the command line.

Below is one example of `getopt`.

1. Taking command line arguments via getopt

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

int main(int argc, char **argv)
{
    int a = 0;
    char *str = NULL;
    bool t = false;
    int ret;

    while ((ret = getopt(argc, argv, "i:s:t")) != -1) {
        switch (ret) {
            case 'i':
                a = atoi(optarg);
            break;
            case 's':
                str = optarg;
            break;
            case 't':
```

```c
                t = true;
            break;
            default:
                printf("%s <-i int value> <-s string value> -t\n", argv[0]);
                return -1;
        }
    }

    printf("a %d\n", a);
    if (str) {
        printf("str %s\n", str);
    }
    printf("t %d\n", t);

    return 0;
}
```

The function `getopt` returns a character that is parsed and the value in the `optarg` variable. The `optarg` is a string. The function `getopt` should be called in a loop. The return value of -1 from `getopt` mean that the getopt has finished parsing.

getopt_long

File I/O

File I/O operations are exposed to the user by the libc. The libc indirectly calls the underlying system calls of the Operating system to manipulate the file. The File I/O APIs are declared in the `stdio.h`.

Basic operations 1. Opening and Closing a file

`FILE` is the structure implemented by the libc that is used in many of the file operations.

```c
FILE *fp;
```

Denotes a file pointer of type `FILE`.

`fopen` is used to open the file. `fclose` is used to close the file.

The prototype of `fopen` is

```c
FILE *fopen(const char *filename, const char *mode);
```

The `fopen` function have the following modes.

S.No	Name	Description
1	`r`	read

S.No	Name	Description
2	`w`	write
3	`a`	append
4	`rb`	read binary
5	`wb`	write binary
6	`ab`	append binary

The prototype of `fclose` is

```
int fclose(FILE *fp);
```

Below example shows the use of `fopen` and `fclose` calls.

```
#include <stdio.h>

int main()
{
    const char *filename = "./test.txt";
    FILE *fp;
    int ret = -1;

    fp = fopen(filename, "r");
    if (fp != NULL) {
        printf("file opened success\n");
        fclose(fp);
        ret = 0;
    } else {
        printf("file not found\n");
        ret = -1;
    }

    return ret;
}
```

2. Reading from a File

Reading a file is possible with the `fgets` function. The `fgets` prototype is as follows.

```
void *fgets(char *str, uint32_t size, FILE *fp);
```

The `fgets` returns pointer to the string that is read, but also the argument `str` contain the data that is read from the file. The `size` argument specify the size of the string.

Below is one example use of `fgets` function.

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

```c
int main()
{
    const char *filename = "./test.txt";
    char msg[1024];
    FILE *fp;
    int ret = -1;

    fp = fopen(filename, "r");
    if (fp) {
        while (fgets(msg, sizeof(msg), fp)) {
            printf("%s", msg);
        }
        fclose(fp);
        ret = 0;
    }

    return ret;
}
```

The `fgets` function adds the `\n` new line upon every call. When calling `strlen` on it, it returns one character more than the actual length. So the last characters of the read string are `\n\0`. To strip of the last character, we can go to the end of the string and one character before and assign `\0` to it.

Below is one way to do it.

```c
char msg[1024];

fgets(msg, sizeof(msg), fp);

/* replace '\n' with '\0'. */
msg[strlen(msg) - 1] = '\0';
```

3. Writing to a file

Writing to a file can be done with the `fputs`, `fputc` or `fprintf` functions.

The prototype of `fputs` is as follows.

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

Below is one example:

```c
#include <stdio.h>

int main()
{
    const char *filename = "./test.txt";
    char msg[1024];
    FILE *fp;
```

```
    int ret = -1;

    fp = fopen(filename, "w");
    if (fp) {
        fputs("Hello World", fp);
        fclose(fp);
        ret = 0;
    }

    return ret;
}
```

4. Copying a file

Copying a file to another file in C involve reading the input file and writing the content to a new file.

Below is one example of copying a file:

```
#include <stdio.h>

int main()
{
    const char *file_in = "c/filecopy.c";
    const char *file_out = "c/filecopy.c.copy";
    FILE *f_in;
    FILE *f_wr;
    char str[1024];

    f_in = fopen(file_in, "r");
    if (!f_in) {
        return -1;
    }

    f_wr = fopen(file_out, "w");
    if (!f_wr) {
        fclose(f_in);
        return -1;
    }

    while (fgets(str, sizeof(str), f_in)) {
        fputs(str, f_wr);
    }

    fclose(f_in);
    fclose(f_wr);

    return 0;
```

```
}
```

5. Number of characters in a file

The function `fgetc` reads one character from a file. If an end of file is reached, it returns an EOF marker which the caller can then check and stop reading further.

```
int fgetc(FILE *fp);
```

In general `fgetc` does not have much usecases for high performat applications. Its use is in writing tools.

Using `fgetc` we can count the number of characters in a file.

```
#include <stdio.h>

int main()
{
    const char *filename = "c/numchar.c";
    FILE *fp;
    char a;
    int num_chars = 0;

    fp = fopen(filename, "r");
    if (!fp) {
        return -1;
    }

    while (1) {
        a = fgetc(fp);
        if (a == EOF) {
            break;
        }
        num_chars ++;
    }

    printf("number of characters %d\n", num_chars);

    return 0;
}
```

6. Finding the file size

The functions `fseek` and `ftell` can be used to seek to a certain position and finding the file offset respectively.

The prototype of `fseek` is as follows:

```
int fseek(FILE *fp, long offset, int whence);
```

The whence parameter is one of `SEEK_SET`, `SEEK_CUR` or `SEEK_END`.

The prototype of `ftell` is as follows:

```
long ftell(FILE *fp);
```

The below example uses both the functions.

```
#include <stdio.h>

int main()
{
    const char *filename = "c/filesize.c";
    long int filesize;
    FILE *fp;

    fp = fopen(filename, "r");
    if (!fp) {
        return -1;
    }

    fseek(fp, 0, SEEK_END);
    filesize = ftell(fp);

    printf("file size %ld\n", filesize);

    fclose(fp);

    return 0;
}
```

Operating with the binary files 1. Reading and Writing to a binary file

I/O operations

1. Using fscanf, fgets and fprintf

The functions `fscanf`, `fgets` and `fprintf` allow to perform more than file operations but also input and output operations on console as well.

There are 3 pointer definitions to access the input and output.

S.No	Name	Description
1	`stdin`	standard input
2	`stdout`	standard output
3	`stderr`	standard error

fscanf

The prototype of `fscanf` looks as follows.

```
int fscanf(FILE *fp, const char *fmt, ...);
```

The `fscanf` function can be used to read the input data entered on console. The `stdin` can be used as the first argument.

```
#include <stdio.h>

int main()
{
    int a;
    char str[100];
    int ret;

    ret = fscanf(stdin, "%d %s", &a, str);
    if (ret != 2) {
        printf("incorrect number of arguments\n");
        return -1;
    }

    printf("a=%d str=%s\n", a, str);

    return 0;
}
```

The `fscanf` can be specifically used to read a pattern set from a file. For example consider a database with name and age group of friends such as the following.

```
dev 33
rahul 34
nithin 38
seema 35
```

Below program reads the input file using `fscanf` and prints the contents on the screen.

```
#include <stdio.h>

int main()
{
    const char *filename = "./c/file_pattern";
    FILE *fp;
    int ret;

    fp = fopen(filename, "r");
    if (!fp) {
        return -1;
    }
```

```
    while (1) {
        char name[30];
        int age;

        ret = fscanf(fp, "%s %d", name, &age);
        if (ret != 2) {
            break;
        }
        printf("name: %s age: %d\n", name, age);
    }

    fclose(fp);

    return 0;
}
```

fgets

The prototype of `fgets` looks as follows.

```
char *fgets(char *str, int size, FILE *fp);
```

The function `fgets` returns the entire line that is read from the `fp`.

- If the `fp` is `stdin` it will return an entire line that is read.
- If the `fp` is a file it will return an entire line that is read.

Below program shows an example usage of `fgets` on `stdin`.

To exit the program press `ctrl + D`. The `ctrl + D` is an end of file marker. Soon as the function `fgets` encounters it, it returns `NULL`.

```
#include <stdio.h>

int main()
{
    char str[100];
    char *err;

    while (1) {
        err = fgets(str, sizeof(str), stdin);
        if (err == NULL) {
            printf("stopping program\n");
            break;
        }

        printf("you entered - %s", str);
    }
```

```
    return 0;
}
```

Below program shows an example of `fgets` reading from the file.

```
#include <stdio.h>

int main()
{
    char *filename = "c/fgets2.c";
    FILE *fp;
    char str[100];

    fp = fopen(filename, "r");
    if (!fp) {
        return -1;
    }

    while (fgets(str, sizeof(str), fp) != NULL) {
        fprintf(stderr, "%s", str);
    }

    fclose(fp);

    return 0;
}
```

Lets look at the below program:

```
#include <stdio.h>

int main()
{
    char *filename = "c/fgets_newline.c";
    FILE *fp;
    char str[100];

    fp = fopen(filename, "r");
    if (!fp) {
        return -1;
    }

    while (fgets(str, sizeof(str), fp) != NULL) {
        fprintf(stderr, "line '%s'\n", str);
    }
    fclose(fp);

    return 0;
```

```
}
```

It prints

```
line '#include <stdio.h>
'
line '
'
line 'int main()
'
line '{
'
line '    char *filename = "c/fgets_newline.c";
'
line '    FILE *fp;
'
line '    char str[100];
'
line '
'
```

The extra newline at each line is the `fgets` adding a newline soon as it encounters a new line.

One way to solve it is to strip off the newline at the end of the string by doing the following:

```
str[strlen(str) - 1] = '\0';
```

The above statement strips off the newline character from the buffer returned by `fgets`.

fprintf

The prototype of `fprintf` looks as follows.

```
int fprintf(FILE *fp, const char *fmt, ...);
```

Just like `fscanf` the `fprintf` prints on the console or to a file. The `stdout` pointer can be used to print the values on the console.

Below example prints the values on the console using `fprintf`.

```
#include <stdio.h>

int main()
{
    char *str = "dev";
    int a = 33;

    fprintf(stdout, "str=%s a=%d\n", str, a);
```

```
    return 0;
}
```

The `fprintf` function can take a file descriptor so it can also be a file that is opened in write mode.

Below example prints the age and name of friends in the file using `fprintf`.

```
#include <stdio.h>

struct S {
    char *name;
    int age;
} friends[] = {
    {"Dev", 33},
    {"Rahul", 34},
    {"Nithin", 38},
    {"Seema", 35},
};

int main()
{
    const char *filename = "c/file_write";
    FILE *fp;
    int i;

    fp = fopen(filename, "w");
    if (!fp) {
        return -1;
    }

    for (i = 0; i < sizeof(friends) / sizeof(friends[0]); i ++) {
        fprintf(fp, "%s %d\n", friends[i].name, friends[i].age);
    }

    fclose(fp);

    return 0;
}
```

Useful macros **1. Minimum of two numbers** i

```
#define MIN(_a, _b) (((_a) < (_b)) ? (_a) : (_b))
```

2. Maximum of two numbers

```
#define MAX(_a, _b) (((_a) > (_b)) ? (_a) : (_b))
```

3. Check if bit is set

```
#define check_bit(_val, _pos) (((_val) >> (_pos)) & 1u)
```

Useful helper functions **1. stoi**

Description:

String to integer conversion with using `strtol`.

```
int stoi(const char *str, int *val)
{
    char *err = NULL;

    *val = strtol(str, &err, 10);
    if (err && (err[0] != '\0')) {
        return -1;
    }

    return 0;
}
```

2. CSV file writer

C++ programming

Introduction

Compiling C++ programs is as simple as using `g++` instead of `gcc`. Here's how to install it in Ubuntu.

```
sudo apt install gcc-g++
```

Few conventions in C++:

1. Source File extensions end with `cxx` or `cpp` or `cc`.
2. Header file extensions end with `hxx` or `hpp` or `h`.
3. The standard header files does not have to be included with `.h` extension.
4. Structure and Enum names can be used without a `struct` prefix before them when declaring.

C++ standard is a continuously evolving standard. Right now the standard C++23 is ongoing. Currently this book tries to differentiate the features present in each standard, but the idea is that the features in each new standard must be exploitable for use in the software that is going to be written. Thus, the features will be mixed and usecases will be drawn upon these features.

cout and cin

`cout` is much similar to `printf` and `cin` is much similar to `fgets`.

But the use of `cout` and `cin` are very different.

Include header file `iostream` when using them.

For example,

```
#include <iostream>

int main()
{
    std::cout << "hello world" << std::endl;
    return 0;
}
```

Compile this with `g++` as `g++ hello_world.cc`. As usual the binary `a.out` is created after successful compilation.

The above program prints "hello world" on screen when executed.

The `std::endl` is like the newline `\n` character. We can as well add `\n` in the above string in the program.

Below is one example of the use of `cin`.

```
#include <iostream>

int main()
{
    std::string str;
    int a;

    std::cin >> a >> str;
    std::cout << "a: " << a << " " << "str: " << str << std::endl;

    return 0;
}
```

Unlike the `fscanf` or `scanf` one does not have to give format specifier for the `cin`.

The strings are taken only one at a time even for `cin` just like the `fscanf`.

New operators in C++

1. The Reference (&) operator.

2. new and delete

```
#include <iostream>

int main()
{
    int *a = new int;
```

```
    *a = 3;

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

    delete a;

    return 0;
}
```

4. Range based for

The range based for loop is a new feature introduced in C++11.

Below is one use of the range based for.

```
#include <iostream>

int main()
{
    for (int a: {1, 2, 3, 4})
        std::cout << "a: " << a << std::endl;

    return 0;
}
```

Below is another use of the range based for.

```
#include <iostream>

int main()
{
    int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    for (auto i : a) {
        printf("%d\n", i);
    }

    return 0;
}
```

The range based for loop have the following advantages:

- No missed condition checks thus removing the probability of infinite loops.
- Auto increment and initialization.

5. Enum classes

Enum classes allow one to compartmentalize the definitions of enumerations.

```
enum class fruits {
    Apples,
    Oranges,
    Banana,
};
```

The enumeration can be declared as,

```
fruits f;

f = fruits::Apple; // assignment of Apple

fruits f1 = fruits::Apple; // initialiation of enum
```

Both declarations help in their own respective.

```
#include <iostream>

enum class fruits {
    Apple,
    Oranges,
    Banana,
};

int main()
{
    fruits f = fruits::Apple;

    std::cout << (int)f << std::endl;

    return 0;
}
```

The enum comparison is still valid.

For example, below program shows the output that comparison is incorrect.

```
#include <iostream>

enum Languages {
    Telugu,
    Tamil,
    English,
    Hindi,
};

int main()
{
```

```
    Languages l1 = Languages::Telugu;
    Languages l2 = Languages::Tamil;

    if (l1 != l2) {
        std::cout << "Languages aren't same" << std::endl;
    }

    return 0;
}
```

The enumerations are not printable with `std::cout` but like the C enumeration, they can be typecasted to integer.

New keywords in C++

constexpr

explicit

auto The `auto` keyword provides an automatic type deduction when used.

An example,

```
auto i = 10;
```

says its an integer but compiler automatically understands this when the value assigned to it is an `int`.

```
auto i = 10; // an int
auto p = 10.1; // a double
auto t = "c++"; // a const string
```

We use auto at times when writing a complex type becomes very hard. We will see in the below sections on more about `auto`. Remember that not all `auto` type deductions are as we expect.

When variables of type `auto` are used, then the initialization must be followed. This generally instructs the compiler to derive the type based on the initialized value.

Typecasting

static_cast

dynamic_cast

reinterpret_cast

Classes

Classes in C++ are similar to the structures in C. The Class is enclosure for data and operations on the data.

A class would generally look like this.

```
class <name> {
    public:
        <variable_type > variable;
        <return_type> function_prototype(parameters..);

    protected:
        <variable_type > variable;
        <return_type> function_prototype(parameters..);

    private:
        <variable_type > variable;
        <return_type> function_prototype(parameters..);
};
```

The below example provides a simple class definition.

```
class S {
    public:
        S() { a = 0; }
        ~S() { }
        void set(int a) { a_ = a; }
        void get() { return a_; }

    private:
        int a_;
};
```

The functions `S()` and `~S()` are constructor and destructor respectively. The constructor gets called when the class object is instantiated. The destructor is called when the class object goes out of scope. Lifecycle of the class object is similar to that of the C variable.

The functions `set` and `get` within `S` are called public member functions. The variable `a_` is a private member variable. In general the private members are prefixed or postfixed with something that differentiates between a local variable and a class member. Without it, it gets really hard to understand the variable's lifetime.

Public members can be accessed by the users of the class while private members are not accessible.

The below declares the class object of `S`.

```
S s;
```

The member functions `set` and `get` are accessible as,

```
S s;

s.set(3);
int val = s.get();
```

Accessing `a_` directly as below results in a compiler error that the variable is part of the `private` section of the class.

```
S s;

int val = s.a_; // results in compiler error
```

The variable `a_` can only be accessible via the `get` method.

If the `public` keyword is not mentioned then the scope is by default `private`.

```
class S {
    S() { a = 0; }
    ~S() { }
    void set(int a) { a_ = a; }
    void get() { return a_; }

    int a_;
};
```

The above code shows class members are all by default, private. Class with all members private is legal and compiles until it is instantiated by creating an object of the class.

Constructors and Destructors

The constructor is called when an object of it is created. For example,

```
class S {
    public:
        S() { a = 3; }
        ~S() { }

        int get() { return a; }

    private:
        int a;
};

int main()
{
```

```
    S s;
}
```

The declaration `S s` calls the constructor `S()`. Constructors and Destructors will have the same name as the class. The destructor has `~` prefix attached to it. The destructor gets called soon after the object loses its scope.

Below is an example,

```
#include <iostream>

class P {
    public:
        P()
        {
            std::cout << "default constructor" << std::endl;
            a = 3;
        }
        ~P()
        {
            std::cout << "destructor called" << std::endl;
        }

        int get() { return a; }

    private:
        int a;
};

int main()
{
    P p;

    std::cout << "p.a: " << p.get() << std::endl;

    return 0;
}
```

Copy constructor

Copy Assignment operator

Move constructor

this pointer

The `this` pointer is nothing but self referencing the class member function.

The below program shows the use of `this` pointer:

```cpp
#include <iostream>

class S {
    public:
        S() { a_ = 0; }
        ~S() { }

        int get() { return this->a_; }
        void set(int a) { this->a_ = a; }

    private:
        int a_;
};

int main()
{
    S s;

    s.set(3);
    std::cout << "val " << s.get() << std::endl;

    return 0;
}
```

The most important use case is that when the input variable to the member function and the class variables / functions are same, then to inhibit confusion and to assist compiler, `this` can be used.

```cpp
#include <iostream>

struct S {
    public:
        S() { a = 0; }
        ~S() { }

        int get() { return a; }
        void set(int a) { this->a = a; }

    private:
        int a;
};

int main()
{
    S s;

    s.set(3);
```

```
    std::cout << "val " << s.get() << std::endl;

    return 0;
}
```

Virtual functions

namespaces

Namespace is a concept to allocate a particular name for one or more classes or functions.

The `using namespace` is used to include a particular namespace without including its name directly when calling its classes or functions defined in it.

```
#include <iostream>

using namespace std;

int main()
{
    cout << "test\n";

    return 0;
}
```

Which could've been the reference of `cout` with `std::cout` without the namespace.

We can define our own namespaces as well.

```
namespace math
{

double square(double number) { return number * number; }

}
```

Defines the function `square` in namespace `math`. Below is one way to use it.

```
#include <iostream>

namespace math
{

double square(double number) { return number * number; }

}

using namespace math;
```

```cpp
using namespace std;

int main()
{
    double n;

    n = square(3);
    cout << "number " << n << endl;

    return 0;
}
```

Another way of using it as follows:

```cpp
#include <iostream>

namespace math
{

double square(double number) { return number * number; }

}

int main()
{
    double n;

    n = math::square(3);
    std::cout << "number " << n << std::endl;

    return 0;
}
```

In order to access the function, we prefix the members with the namespace followed by the `::` operator.

Overloading

Overloading is a concept of having many signatures. In general C++ provides operator overloading and the function overloading.

Operator Overloading

Operators such as `+`, `-`, `*` and `/` and many others can be overloaded.

Overloading + operator

Below is one example of overloading the `+` operator.

```
#include <iostream>

struct A {
    int val;
};

A operator+(A a, A b)
{
    A r;

    r.val = a.val + b.val;

    return r;
}

int main()
{
    A a { .val = 3 };
    A b { .val = 3 };

    A r = a + b;

    std::cout << "r val " << r.val << std::endl;

    return 0;
}
```

Overloading the + operator in class.

Below is one example,

```
#include <iostream>

struct S {
    int a;

    S() { a = 0; }
    S(int a) { this->a = a; }
    ~S() { }

    S operator+(const S &s)
    {
        S r;

        r.a = this->a + s.a;

        return r;
```

```
    }
};

int main()
{
    S s(3);
    S r(3);
    S t;

    t = s + r;

    std::cout << "t.a: " << t.a << std::endl;

    return 0;
}
```

Overloading the ++ operator

```
#include <iostream>

struct S {
    int a;

    S() { a = 0; }
    S(int a) { this->a = a; }

    void operator ++() { a ++; }
};

int main()
{
    S s(3);

    ++s;

    std::cout << s.a << std::endl;
}
```

In the above example, two variables `a` and `b` of the structure `A` are added and another structure `r` is returned.

Overloading « operator

The `<<` operator can be overloaded as well. Below is one example of overloading the `<<` operator.

```
#include <iostream>

struct S {
```

```
    int a;
    double d;
};

std::ostream &operator<<(std::ostream &os, const S &s)
{
    os << "s.a: " << s.a << " " << "s.d: " << s.d;

    return os;
}

int main()
{
    S s { .a = 3, .d = 3.1 };

    std::cout << s << std::endl;

    return 0;
}
```

Function Overloading

Function overloading allows to have more than one signature to a function. Below is one example,

```
int F(int a);
int F(int a, int b);
```

The function `F` here is overloaded and have the two signatures. The compiler finds out which variant of `F` to be called based on the definition.

```
#include <iostream>

int F(int a) { return a; }
int F(int a, int b) { return a + b; }

int main()
{
    std::cout << "F(3): " << F(3) << std::endl;
    std::cout << "F(3, 3): " << F(3, 3) << std::endl;

    return 0;
}
```

In general the arguments to the functions are allowed to be overloaded. However, the return type of functions are not allowed to be overloaded, For example,

```
#include <iostream>
```

```cpp
int F(int a) { return a; }
void F(int a) { }

int main()
{
    F(3);
}
```

Results in a compiler error.

```
cpp/overload_f.cc:4:6: error: ambiguating new declaration of ‘void F(int)’
    4 | void F(int a) { }
      |      ^
cpp/overload_f.cc:3:5: note: old declaration ‘int F(int)’
    3 | int F(int a) { return a; }
      |     ^
```

Function overloading is really useful when a function wants to do different jobs keeping the name same but different signature.

Exception Handling

noexcept

The `noexcept` is an operator and also a specifier.

noexcept specifier

The `noexcept` specifier informs the compiler that the particular function / constructor / destructor does not produce an exception. It also governs some underlying rules when it comes to inherited classes.

A `noexcept` specifier can be attached to the end of the function as:

```cpp
int f(void) noexcept; // says function does not throw an exception
int g(void); // may throw an exception
```

A function declared with `noexcept` specifier but throws, results in the library calling `std::terminate`. This makes the exception uncatchable.

An overloaded function can have a different exception specification. For example, below example is valid.

```cpp
int f(void) noexcept;
int f(std::string arg);
```

Below program covers almost all the `noexcept` cases:

```cpp
#include <iostream>

void f() noexcept { std::cout << "f called" << std::endl; }
```

```
void f(std::string arg) { std::cout << "f called with arg: " << arg << std::endl; }

void g() noexcept { throw std::runtime_error("an exception g()"); }

void p() { throw std::runtime_error("an exception p()"); }

int main()
{
    f();
    f("test");

    try {
        p();
    } catch (std::exception &e) {
        std::cout << "caught the exception from function p(): " << e.what() << std::endl;
    }

    try {
        g();
    } catch (...) {
        std::cout << "caught the exception from function g()\n";
    }
}
```

If a base class contains any of member functions with `noexcept` specifier then the derived class must also contain the `noexcept` specification. Without it, this results in compiler error.

```
#include <iostream>

class F {
    public:
        virtual void f() noexcept = 0;
};

class G : public F {
    public:
        void f() { std::cout << "in f()" << std::endl; }
};

int main()
{
    class G g;

    g.f();
}
```

Results in compiler error,

```
cpp/noexcept_inh.cc:10:22: error: looser exception specification on overriding virtual funct
   10 |                  void f() { std::cout << "in f()" << std::endl; }
      |                       ^
cpp/noexcept_inh.cc:5:30: note: overridden function is 'virtual void F::f() noexcept'
    5 |                  virtual void f() noexcept = 0;
      |                               ^
```

However, a derived class can specify `noexcept` specifier although the base class do not have it.

For example the below program compiles.

```
#include <iostream>

class F {
    public:
        virtual void f() = 0;
};

class G : public F {
    public:
        void f() noexcept { std::cout << "in f()" << std::endl; }
};

int main()
{
    class G g;

    g.f();
}
```

Standard library

Standard library or STL in short is a group of helper function that ease up programming. Nowadays, they are more focussed towards helping programmers write OS independent software using C++.

Arrays

`std::array` defines an array type. The `std::array` contains the following methods:

The `std::array` template looks as follows,

```
template <typename T>
std::array<T, n>
```

Most usual way of declaring an array of integers is,

```
std::array<int, 10> a;
```

This is similar to declaring `int a[10]`.

1. at Returns the value at the index.

2. operator[] Used to access the element stored at the index.

3. size Returns the size of the array.

Below is an example of `std::array`.

```
#include <iostream>
#include <array>

int main()
{
    std::array<int, 10> a;
    int i;

    for (i = 0; i < a.size(); i ++) {
        a[i] = i + i;
    }

    for (i = 0; i < a.size(); i ++) {
        std::cout << "i : " << i << " "  << "a[i] : " << a.at(i) << std::endl;
    }

    return 0;
}
```

Strings

`std::string` defines a string type.

Vectors

`std::vector` defines a vector type.

Lists

`std::list` defines a list type.

Queues

`std::queue` defines a queue type. The `std::queue` can take any type. It is generally identified with templates as,

```
template <typename T> std::queue<T>
```

The `T` argument is a template type, the type is deduced when the `std::queue` has been declared.

```
std::queue<int> i; // a queue of ints

struct P {
    int p;
};
std::queue<P> p; // queue of structures (P)
```

The queue provides the following operations.

S.No	Name	Description
1	`front`	Get the first element of the queue
2	`back`	Get the last element of the queue
3	`push`	Push an element in queue
4	`pop`	Pop an element from the queue
5	`size`	Get the size of the elements
6	`empty`	Check if there are any more elements in queue

Below is one usage of queue with simply integer data type.

```
#include <iostream>
#include <queue>

int main()
{
    std::queue<int> q;
    int size;

    printf("q empty %d\n", q.empty());

    q.push(1);
    q.push(2);
    q.push(3);
    q.push(4);
    q.push(5);
    q.push(6);

    printf("number of elements %lu, queue empty %d\n", q.size(), q.empty());
    printf("front %d back %d\n", q.front(), q.back());

    while (1) {
        size = q.size();
        if (size <= 0) {
            break;
```

```
        }

        int val = q.front();
        q.pop();
        printf("val : %d\n", val);
    }
}
```

The general usecases of queues are the following:

1. Producer and Consumer data sharing with a queue. Producer adds item in queue, consumer removes item from the queue.

The above case apply to almost all real world problems involving multi threading. Multi threading is described below.

Sets

Dequeue

Maps

`std::map` defines a map type.

shared_ptr, unique_ptr The `std::shared_ptr` is a scoped allocator defined in C++11. The idea of the scoped allocation is to free the allocated memory automatically when a count of all of its references to the allocated memory become 0.

Below is one example usage of `std::shared_ptr`.

```
#include <iostream>
#include <memory>

struct S {
    int s;
};

int main()
{
    std::shared_ptr<S> s;

    s = std::shared_ptr<S>(new S);

    s->s = 1;

    std::cout << "s->s: " << s->s << std::endl;
```

```
    return 0;
}
```

The allocator `std::make_shared` is used that can return a pointer of type `std::shared_ptr`.

Below is one example,

```
#include <iostream>
#include <memory>

struct P {
    int val;
};

void print(std::shared_ptr<P> p)
{
    struct P *p1;

    p1 = p.get();
    printf("p->val %d deref->val %d\n", p->val, p1->val);
    printf("p.val %d\n", (*p).val);
    printf("use_count %ld\n", p.use_count());
}

int main()
{
    std::shared_ptr<P> ptr;

    ptr = std::make_shared<P>();
    ptr->val = 4;

    print(ptr);

    return 0;
}
```

As you can see we do not call any `free` or `delete`. The reason being that when the program goes out of scope (in this case the main function scope) it is already freed.

Running the `valgrind` shows 0 leaks.

```
==208668== HEAP SUMMARY:
==208668==     in use at exit: 0 bytes in 0 blocks
==208668==   total heap usage: 4 allocs, 4 frees, 74,780 bytes allocated
==208668==
==208668== All heap blocks were freed -- no leaks are possible
```

Writing `shared_ptr` class:

To write the `shared_ptr` we need to consider the following.

- Referencing counting.
- Allocation and freeing.

The allocation part is not available in this example and leave it upto the caller. The destructor will perform the freeing, this is exactly one of the purposes of the `shared_ptr` feature.

```
#include <iostream>

template <typename T>
class shared_ptr {
    public:
        explicit shared_ptr() : count_(0), memory_(nullptr) {
        }
        explicit shared_ptr(T *t) : count_(0) {
            memory_ = t;
            count_ ++;
        }

        shared_ptr operator=(const shared_ptr &s) {
            this->count_ ++;
            std::cout << "called" << std::endl;
            return *this;
        }

        shared_ptr(const shared_ptr &t) {
            count_ = 0;
            memory_ = nullptr;
            memory_ = t.memory_;
            count_ = t.count_ + 1;
        }

        ~shared_ptr() {
            count_ --;
            if (memory_ && (count_ == 0)) {
                delete memory_;
            }
        }
        void set(T *memory) {
            memory_ = memory;
            count_ ++;
        }

        T *get() { return memory_; }
```

```
        bool unique() const { return count_ == 1; }

        T *operator->() const { return memory_; }
        T &operator*() const { return *memory_; }

    private:
        int count_;
        T *memory_;
};

struct S {
    int s;
};

void K(shared_ptr<S> s)
{
    s->s = 6;
}

int main()
{
    shared_ptr<S> s = shared_ptr<S>(new S);
    shared_ptr<int> s1 = shared_ptr<int>(new int);

    s->s = 3;
    *s1 = 3;

    K(s);

    std::cout  << "s: " << s->s << " " << "s1: " << *s1 << std::endl;

    return 0;
}
```

Here we write a deref operator `->` to return the actual underlying pointer. The caller still assumes that the `shared_ptr` is still a wrapper which is true. We also have written the operator `*` if in case the called type is a basic type such as an integer or a floating point.

unique_ptr

The `std::unique_ptr` is a scoped allocator that is similar to `shared_ptr`. The feature is introduced in C++14.

```
#include <iostream>
#include <memory>
```

```
struct P {
    int val;
};

void print(std::unique_ptr<P> &p)
{
    printf("val %d\n", p->val);
}

int main()
{
    std::unique_ptr<P> ptr;

    ptr = std::make_unique<P>();
    ptr->val = 4;

    print(ptr);
    return 0;
}
```

File systems

Threads

C++ implements abstraction of threads based upon the pthreads. The class `std::thread` defines the thread interface. Threads defined in C++11 onwards.

Creating a thread is a simple job of declaring a thread object and passing the function that serves as a thread function.

Below is one example:

```
#include <iostream>
#include <thread>

void thread_f()
{
    std::cout << "in thread" << std::endl;
}

int main()
{
    std::thread t(thread_f);

    std::cout << "starting thread" << std::endl;
    t.join();
    std::cout << "joined thread" << std::endl;
}
```

Compile the above program with `-pthread` option. Some compilers may not require this option.

Lets see below example, that creates two threads.

```
#include <iostream>
#include <thread>

void thread_1()
{
    std::cout << "in thread_1" << std::endl;
}

void thread_2()
{
    std::cout << "in thread_2" << std::endl;
}

int main()
{
    std::thread t1(thread_1);
    std::thread t2(thread_2);

    std::cout << "waiting for threads" << std::endl;

    t1.join();
    t2.join();

    std::cout << "stop" << std::endl;
}
```

When compiling and running this program results in non-sequential outputs. For example.

```
in thread_1
waiting for threads
in thread_2
stop
```

But the expectation is that the `main` function messages will appear before the thread function calls.

In general, when threads are created by the operating system, the execution totally depends on the scheduler.

Mutexes

```
#include <iostream>
#include <thread>
```

```cpp
#include <mutex>

std::mutex lock;
static int count;

void thread_1()
{
    while (1) {
        std::cout << "in thread_1: waiting for lock" << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
        lock.lock();
        std::cout << "in thread_1: acquired" << std::endl;
        count ++;
        std::cout << "in thread_1: val " << count << std::endl;
        std::cout << "in thread_1: released" << std::endl;
        lock.unlock();
    }
}

void thread_2()
{
    while (1) {
        std::cout << "in thread_2: waiting for lock" << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
        lock.lock();
        std::cout << "in thread_2: acquired" << std::endl;
        count ++;
        std::cout << "in thread_2: val " << count << std::endl;
        std::cout << "in thread_2: released" << std::endl;
        lock.unlock();
    }
}

int main()
{
    std::thread t1(thread_1);
    std::thread t2(thread_2);

    t1.join();
    t2.join();
}
```

Conditional Variables

Derived Classes

C++ allows a class to inherit one or more other classes. This is called inheritance.

```
struct B {
};

struct D : public B {
};
```

Here the class `D` inherits the class `B`. The public member functions in `B` are inherited in D. This means they are callable in D without class object. They can also be overrriden if needed.

For example,

```
struct B {
    B() { a_ = 3; }
    ~B() { }
    int get() { return a_; }

    private:
        int a_;
};

struct D : public B {
    D() { a_ = 6; }
    ~D() { }
    int get() { return a_; }
    int get_b() { return B::get(); } // access B::get() directly

    private:
        int a_;
};
```

we access the member of `D` the following way:

```
D d;

std::cout << "d.get_b(): " << d.get_b() << std::endl;
```

```
D d;

std::cout << "d.get():: " << d.get() << std::endl;
```

This results in accessing `a_` within `D`.

Following is another way of accessing `B::get()`.

```
D d;

std::cout << "d.B::get(): " << d.B::get() << std::endl;
```

Below is an example of the inheritance with base and derived classes.

```
#include <iostream>

struct B {
    public:
        B() { a_ = 3; }
        ~B() { };
        int get() { return a_; }

    private:
        int a_;
};

struct D : public B {
    public:
        D() { a_ = 6; };
        ~D() { };
        int get() { return a_; }
        int get_b() { return B::get(); }

    private:
        int a_;
};

int main()
{
    D d;

    std::cout << "d.a: " << d.get()
              << " d.B::a: " << d.B::get()
              << " b.a: " << d.get_b() << std::endl;

    return 0;
}
```

Abstract Classes

```
class abstract_class {
    public:
        virtual function_return function_prototype(parameters..) = 0;
}
```

Below is one example,

```
class S {
    public:
        virtual int get() = 0;
};
```

The implementation inherits the abstract class. For example,

```
class R : public S {
    public:
        R() { a_ = 3; }
        ~R() { }

        int get() { return a_; }

    private:
        int a_;
};
```

Below is one simple example,

```
#include <iostream>

class S {
    public:
        virtual int get() = 0;
};

class R: public S {
    public:
        R() { a_ = 3; }
        ~R() { }

        int get() { return a_; }

    private:
        int a_;
};

int main()
{
    R r;

    std::cout << "R: " << r.get() << std::endl;

    return 0;
}
```

Though the above program instantiates `R` directly, it may not be very useful to instantiate `R`. In general abstract classes can be instantiated via other means. See Design patterns for Factory method.

Templates

Templates allow to write software generically. Below is an example of a template.

```
template <typename T>
class calculator {
};
```

Where `T` is the type. The instantiation of the class object for this would be,

```
class calculator<int> cal;
```

Defines the class `calculator` as a template.

There can be more than one template types.

For example, the following is valid.

```
template <typename T, typename R, typename P>
class calculator {
};
```

The member functions of the class can be written as follows.

```
template <typename T>
class calculator {
    public:
        T add(T a, T b);
        T sub(T a, T b);
        T mul(T a, T b);
        T div(T a, T b);
        T mod(T a, T b);
};
```

The member functions describe that the inputs to the member functions are all of type `T` and returns type `T`.

For example,

```
calculator<int> cal;
```

declaration of object means that the member functions also are of same type.

For example, passing other type instead of the same results in compiler error.

We can write the calculator program that is written in C with macros, in C++ with templates as follows:

```
#include <iostream>
```

```cpp
template <typename T>
class calculator {
    public:
        T add(T a, T b) { return a + b; }
        T sub(T a, T b) { return a - b; }
        T mul(T a, T b) { return a * b; }
        T div(T a, T b) { return a / b; }
        T mod(T a, T b) { return a % b; }
};

int main()
{
    calculator<int> cal;

    std::cout << "Add: " << cal.add(3, 3) << std::endl;
    std::cout << "Sub: " << cal.sub(3, 3) << std::endl;
    std::cout << "Mul: " << cal.mul(3, 3) << std::endl;
    std::cout << "Div: " << cal.div(3, 3) << std::endl;
    std::cout << "Mod: " << cal.mod(3, 3) << std::endl;

    return 0;
}
```

Here we used `calculator<int>` for the `cal` object. This means all the operations / member functions of the calculator will accept integers. If we used `calculator<double>` it would be the double that is being used in all the operations.

For example, there can be a chance that string could've been used such as `calculator<std::string> cal`. In this case the compilation results in failure because the arguments given are integers. Certain overloaded string operations such as `+`, `-` may work, but the other operations which does not have the overloaded types will result in compilation failure.

A normal function can be overloaded with the templates such as the following example,

```cpp
#include <iostream>

template <typename T>
void print(T val)
{
    std::cout << "template: val: " << val << std::endl;
}

void print(int v)
{
    std::cout << "int: val: " << v << std::endl;
```

```
}

int main()
{
    print(3);
    print<std::string>("hello");
    print<int>(3);

    return 0;
}
```

Here the `print(3)` is called directly which by the explicit function and variable declaration the `print(int v)` gets called. When the `print` is called with `<` and `>`, the templatized version gets invoked.

If in case `print(int v)` is not available, the first call to `print(3)` actually results as an implicit call to `print<int>(3)`. The compiler deduces the type implicitly.

Usecase 1: Implementing std::array

```
#include <iostream>

template <typename T, int n>
class array {
    public:
        explicit array() = default;
        ~array() = default;

        T &operator[](int index) { return array_[index]; }

        T at(int index) { return array_[index]; }

        int size() { return n; }

        void clear() {
            for (auto i = 0; i < n; i ++) {
                array_[i] = 0;
            }
        }
    private:
        T array_[n];
};

int main()
{
    array<int, 10> a;
```

```
    a[1] = 4;

    std::cout << "array " << a[1] << std::endl;
}
```

Usecase 2: Implementing LRU cache

LRU is called as Least Recently Used. It is a cache mechanism that allows to remove the least used data members. The below algorithm describes based on the assumption that the cache is finite.

For example consider the following cache.

```
| A | B | C | D | E |
```

Each of these elements are associated with a sequence number.

```
| A |1| B |2| C |3| D |4| | E |5|
```

This can be represented in structural format as,

```
template <typename T>
struct lru_cache_item {
    T item;
    uint32_t seq_no;
};
```

The elements are structured as,

```
A,1
B,2
C,3
D,4
E,5
```

If `F` needs to be added to the above list, the element `A` will be evicted because the sequence number following `A` is the least.

So the resulting list becomes,

```
F,6
B,2
C,3
D,4
E,5
```

If item `B` wants to be updated (or more specifically location where the element `B` is stored needs to be updated), the sequence number belong to it will also be updated. For example the resulting list becomes,

```
F,6
B,7
C,3
```

```
D,4
E,5
```

Now, if any new element needs to be added to the list the next element to be removed is `C`.

```
#include <iostream>

/**
 * Defines a cache line item
 */
template <typename T>
struct lru_cache_items {
    T val;
    bool is_avail;
    uint32_t seq_no;
};

/**
 * Template of the lru_cache.
 */
template <typename T, int n>
class lru_cache {
    public:
        explicit lru_cache()
        {
            index_ = 0;
            seq_no_ = 0;

            for (auto i = 0; i < n; i ++) {
                items_[i].is_avail = false;
                items_[i].seq_no = 0;
            }
        }
        ~lru_cache() { }

        lru_cache &push(T &val)
        {
            /* Add to the cache for the first n items */
            if (index_ < n) {
                seq_no_ ++;

                items_[index_].val = val;
                items_[index_].is_avail = true;
                items_[index_].seq_no = seq_no_;

                index_ ++;
```

```
    } else {
        /**
         * Try evicting an item if the item is old.
         *
         * if the particular cache line's sequence number is oldest, evict it and up
         */

        /* Find least recently used. */
        int i;
        int index = -1;
        uint32_t least_val = seq_no_;

        for (i = 0; i < n; i ++) {
            if (items_[i].seq_no < least_val) {
                least_val = items_[i].seq_no;
                index = i;
            }
        }

        if (index != -1) {
            seq_no_ ++;

            items_[index].val = val;
            items_[index].is_avail = true;
            items_[index].seq_no = seq_no_;
        }
    }

    return *this;
}

void update(T &val, int index)
{
    /* Update always involve updating sequence number, a way to tell that the
     * cache line is hot.
     */
    seq_no_ ++;

    items_[index].val = val;
    items_[index].is_avail = true;
    items_[index].seq_no = seq_no_;
}

int get_index(T &val)
{
    int i;
```

```
            for (i = 0; i < n; i ++) {
                if (items_[i].val == val) {
                    break;
                }
            }

            return i == n ? -1 : i;
        }

        T &get_val(int index)
        {
            return items_[index].val;
        }

    private:
        lru_cache_items<T> items_[n];
        uint32_t seq_no_;
        uint32_t index_;
};

int main()
{
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int p1 = 11;
    int p2 = 12;
    int u1 = 13;
    int u2 = 14;
    int i;
    lru_cache<int, 10> lru;

    for (i = 0; i < 10; i ++) {
        lru.push(a[i]);
    }

    lru.update(a[0], 0);
    lru.update(a[1], 1);
    lru.push(p1);
    lru.push(p2);

    for (i = 0; i < 10; i ++) {
        std::cout << "val: " << lru.get_val(i) << std::endl;
    }
}
```

Appendix B

Scoppe and Lifetime

Use cases

Safe Queue

```
#include <iostream>
#include <mutex>
#include <condition_variable>
#include <thread>
#include <queue>

template <typename T>
class safe_que {
    public:
        safe_que(const safe_que &) = delete;
        const safe_que &operator=(const safe_que &) = delete;
        safe_que(const safe_que &&) = delete;
        const safe_que &&operator=(const safe_que &&) = delete;

        /**
         * @brief - get an instance.
         */
        static safe_que *instance() {
            static safe_que q;
            return &q;
        }
        ~safe_que() { }

        /**
         * @brief - add an element to the safe queue.
         */
        void add(T &elem) {
            std::unique_lock<std::mutex> lock(lock_);
            items_.push(elem);
            cond_.notify_all();
        }

        /**
         * @brief - get the element from the queue.
         */
        void get(T &val) {
            std::unique_lock<std::mutex> lock(lock_);
            cond_.wait(lock);
            val = items_.front();
```

```
            items_.pop();
        }

    private:
        explicit safe_que () { }
        std::queue<T> items_;
        std::mutex lock_;
        std::condition_variable cond_;
};

void thread_f()
{
    safe_que<int> *q = safe_que<int>::instance();

    while (1) {
        int data;

        /* Get the element from the queue. */
        q->get(data);
        printf("data %d\n", data);
    }
}

int main()
{
    safe_que<int> *q = safe_que<int>::instance();
    std::thread t(thread_f);
    int count = 0;

    /* Produce data every 100 msec. */
    while (1) {
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        count ++;

        q->add(count);
    }
}
```

Thread Pool

The thread pool is a group of threads that work on specific jobs that are queued to them.

The threads are created early in the startup. All the threads could listen on a single queue or on a multi queue. The main thread assigns the tasks to the threads and each thread execute these tasks that are queued to them.

So idea of having separate queue for each thread generally makes sense to avoid any possible starvation when each thread pulls the task from the queue.

Thus we need to define a context for each thread. It may look something like the below.

```
struct thread_context {
    uint32_t id_; // identifier for thread
    std::shared_ptr<std::thread> t_; // actual thread pointer
    std::mutex lock_; // lock for the queue
    std::condition_variable cond_; // condition variable for the queue
    std::queue<work_fn> work_list_; // list of function callbacks
    int queue_length_; // length of the queue
    bool queued_; // signals if work is queued
    bool signalled_; // signal to quit the thread

    // constructor
    explicit thread_context(uint32_t id);

    // queue the work
    void queue(work_fn fn);

    // get thread id
    uint32_t get_id() { return id_; }

    // join the thread
    void join() { t_->join(); }

    // get the queue size
    int get_queue_size() { return queue_size_; }

    // signal the thread
    void signal();
    // destructor
    ~thread_context();
    void worker_thread(); // the thread function
};
```

The worker function callback would look as follows,

```
typedef std::function<void(void)> work_fn;
```

The `id_` is a thread identifier.

We took `std::thread` as `shared_ptr` so we can instantiate it later during the allocation.

The `lock_` and `cond_` variables are used to sequentialize access to the queue `work_list_`.

The queue holds the list of function callbacks that are to be executed.

We use `queue_length_` to determine the fairness and optimize the time it takes for a work callback to execute on a thread.

The variable `queued_` is used for synchronization between the main thread and the worker threads.

The variable `signalled_` is used to inform the thread when to quit.

The `worker_thread` is the worker thread function that executes the queued work functions.

The constructor `thread_context` would create the thread as follows.

```
thread_context::thread_context(uint32_t id)
{
    t_ = std::make_shared<std::thread>(&thread_context::worker_thread, this);
}
```

Now, the main thread or the caller library functions would have to store each thread context.

```
class thread_pool {
    public:
        // initialize thread pool with n_threads
        explicit thread_pool(int n_threads);
        ~thread_pool();

        // queue work to the thread pool
        void queue(work_fn fn);

        // run the thread pool wait for them to finish execution
        void run();

        // signal the threads to stop
        void signal();

    private:
        // store number of threads
        int n_threads_;

        // vector of threads
        std::vector<std::shared_ptr<thread_context>> tc_list_;
};
```

The users of the thread pool can then simply do,

```
thread_pool t(4);

t.queue(&work_1);
```

```
t.queue(&work_2);

..

t.run();
```

The method `thread_pool::run` is simply used to wait for all threads to finish. This call will return if the threads stop executing. So at some point in time such as program stop, we call `thread_pool::signal` to signal the threads to stop.

The job queueing is takes a very generic function that accepts and returns no parameter.

This is of type `std::function` so one can use straight functions or use `std::bind` to create a callback.

For example, to make a private member function of the below class `S` to be called, one can make a callback and pass it to the `thread_pool::queue`.

```
struct S {
    public:
        explicit S() { }
        ~S() { }

        void register_work();

    private:
        void work()
        {
            std::cout << "work function" << std::endl;
        }
};

void S::register_work()
{
    auto callback = std::bind(&S::work, this);
    thread_pool tp(4);

    tp.queue(callback);
}
```

Below is the thread pool implementation.

```
/**
 * @brief - Thread pool implementation within 200 lines.
 *
 * @author - Devendra Naga (github.com/devendranaga/)
 *
 * @copyright - 2023-present.
```

```
 * @license - GPLv2
 */
#include <iostream>
#include <queue>
#include <vector>
#include <functional>
#include <thread>
#include <mutex>
#include <condition_variable>

typedef std::function<void(void)> work_fn;

class TD {
    public:
        explicit TD(uint32_t id) :
                        id_(id),
                        queue_size_(0),
                        queued_(false),
                        signalled_(false)
        {
            t_ = std::make_shared<std::thread>(&TD::thread_fn, this);
        }
        ~TD () { }

        void queue(work_fn fn)
        {
            {
                std::unique_lock<std::mutex> l(lock_);
                queued_ = true;
                queue_size_ ++;
                work_list_.push(fn);
                cond_.notify_one();
            }
        }

        uint32_t get_id() { return id_; }

        void join() { t_->join(); }

        int get_queue_size() { return queue_size_; }

        void signal()
        {
            std::unique_lock<std::mutex> l(lock_);
            signalled_ = true;
            cond_.notify_one();
```

```
        }

    private:
        uint32_t id_;
        int queue_size_;
        bool queued_;
        bool signalled_;
        std::queue<work_fn> work_list_;
        std::shared_ptr<std::thread> t_;
        std::mutex lock_;
        std::condition_variable cond_;

        void thread_fn()
        {
            int queue_size = 0;
            work_fn fn = nullptr;

            while (1) {
                {
                    fn = nullptr;
                    std::unique_lock<std::mutex> l(lock_);
                    if (queue_size == 0) {
                        cond_.wait(l, [this] { return (queued_ == true) ||
                                                      (signalled_ == true); });
                        if (signalled_) {
                            break;
                        }
                        queued_ = false;
                    }
                    queue_size = work_list_.size();
                    if (queue_size > 0) {
                        fn = work_list_.front();
                        work_list_.pop();

                        printf("remaining items in thread %d %d\n",
                                                id_, queue_size_);

                    }
                }
                if (fn) {
                    fn();
                    queue_size_ --;
                }
            }
        }
};
```

```cpp
class TP {
    public:
        explicit TP(int n_threads) : n_threads_(n_threads)
        {
            int i;

            for (i = 0; i < n_threads; i ++) {
                std::shared_ptr<TD> td;

                td = std::make_shared<TD>(i);
                td_list_.push_back(td);
            }
        }

        void queue(work_fn fn)
        {
            int lowest = td_list_.begin()->get()->get_queue_size();
            std::vector<std::shared_ptr<TD>>::iterator it;
            std::vector<std::shared_ptr<TD>>::iterator lowest_it =
                                                        td_list_.end();

            for (it = td_list_.begin(); it != td_list_.end(); it ++) {
                int q_size = it->get()->get_queue_size();
                if (q_size <= lowest) {
                    lowest = q_size;
                    lowest_it = it;
                }
            }

            printf("chose lowest id [%d] queue [%d]\n",
                            lowest_it->get()->get_id(),
                            lowest_it->get()->get_queue_size());
            if (lowest != -1) {
                lowest_it->get()->queue(fn);
            }
        }

        void run()
        {
            for (auto it : td_list_) {
                it.get()->join();
            }
        }

        void signal()
```

```
        {
            for (auto it : td_list_) {
                it.get()->signal();
            }
        }

    private:
        int n_threads_;
        std::vector<std::shared_ptr<TD>> td_list_;
};

static int count;
std::mutex lock;

void work_1()
{
    fprintf(stderr, "executing infinite loop\n");
    while (1) {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        {
            std::unique_lock<std::mutex> l(lock);
            fprintf(stderr, "work_1: counter: %d\n", count);
            if (count > 1) {
                break;
            }
        }
    }
}

void work_2()
{
    std::unique_lock<std::mutex> l(lock);
    fprintf(stderr, "work_2: counter: %d\n", count);
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    count ++;
}

void work_3()
{
    std::unique_lock<std::mutex> l(lock);
    fprintf(stderr, "work_3: counter: %d\n", count);
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    count ++;
}

int main()
```

```
{
    TP t(4);
    int i;

    t.queue(&work_1);

    for (i = 0; i < 10; i ++) {
        t.queue(&work_3);
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
        t.queue(&work_2);
    }

    t.signal();

    t.run();
}
```

Event Driven System

Design Patterns

Factory Design pattern

Singleton Design pattern

The singleton pattern is used when an object is being used by many other classes. One way to do is to instantiate it statically and return that instance.

Since its been used by many other classes, the instantiation happens statically within the class itself. For this one generally defines `instance` member function that returns the statically declared class object. The constructor is hidden to prevent any more instantiations by the class declarations.

An example singleton class looks as follows.

```
class singleton {
    public:
        static singleton *instance() {
            static singleton s;
            return &s;
        }
        ~singleton() = default;
        singleton(const singleton &) = delete;
        const singleton &operator=(const singleton &) = delete;
        singleton(const singleton &&) = delete;
        const singleton &&operator=(const singleton &&) = delete;

        int member(...);
```

```
    private:
        explicit singleton();
}
```

We delete the copy and move constructors so that only one instance that is created during the call to the static member function `instance` is the only instance that is available.

usecase.1: Logging utility

Singleton can be used when writing a logging utility that logs the message / debug message to something like console or to a file, but does not require instantiation everytime when we want to use the object.

An example of it looks as follows:

```
class log {
    public:
        static log *instance() {
            static log l;
            return &l;
        }
        ~log() { }
        log(const log &) = delete;
        const log &operator=(const log &) = delete;
        log(const log &&) = delete;
        const log &&operator=(const log &&) = delete;

        int info(const char *msg, ...);
        int verbose(const char *msg, ...);
        int debug(const char *msg, ...);
        int warn(const char *msg, ...);
        int error(const char *msg, ...);
        int fatal(const char *msg, ...);
    private:
        explicit log() { }
};
```

The above class is a singleton that has many member functions for logging such as,

1. info
2. verbose
3. debug
4. warning
5. error
6. fatal

The member functions of this singleton can be accessed from anywhere as long as they include the header file that this class belongs.

The call can be simply made as :

```
log *l = log::instance();

l->info("info message\n");
```

or

```
log::instance()->info("info message\n");
```

usecase.2: Datastore

Data store is another use of singleton class. Lets see the below class:

```
class key_val_datastore {
    public:
        static key_val_datastore *instance() {
            static key_val_datastore ds;
            return &ds;
        }
        ~key_val_datastore() { }
        key_val_datastore(const key_val_datastore &) = delete;
        const key_val_datastore &operator=(const key_val_datastore &) = delete;
        key_val_datastore(const key_val_datastore &&) = delete;
        const key_val_datastore &&operator=(const key_val_datastore &&) = delete;

        int write(uint32_t val);
        int write(std::string val);
        int read(uint32_t &val);
        int read(std::string &val);
    private:
        explicit key_val_datastore() { }
};
```

Just as in the usecase 1, the data store can be read and written with the member functions.

Ofcourse there will be parallel accesses, which can be sequentialized with the use of mutexes.

Builder Design pattern

Appendix C

Code organization for software development

Building large software

creating libraries

When writing software for a large scale project, sometimes some of the utility functions or common routines need to be called often. Sometimes, many application tend to replicate these and roll out their own implementation of these functions. This is generally has the following problems:

1. Code bloat with repeat of many functions doing same thing.
2. If there is a problem in one function, the other callers that are not using this function will not get a benefit when the function's problem is solved.
3. Increased program size.
4. Increased development times.

These are some of the reasons why libraries concept is introduced.

The Compiler provides a way to generate libraries out of a group of C or C++ source files. They are group of object files and the functions they offer, contain prototypes in the respective .h files.

These libraries can then be linked with the other object files during the linkage time to create the final binary / library.

There are static and dynamic libraries.

Static libraries are the ones that when linked, copies the function directly to the target binary. This increases the size of the binary considerably.

Dynamic libraries on the other hand, keeps a reference of the function in the target binary. This may not increase the size of the binary. However, during the runtime, the loader sees the reference of the function and loads the function when it gets called. This adds additional runtime over head.

Check `cmake` section about creating static and dynamic libraries.

creating binaries

cmake

cmake is a scripting language that can be used to create what are known as CMake Files. Each of these can be used to compile the group of source files to generate target libraries or binaries.

1. Creating a basic CMakeLists.txt

```
cmake_minimum_required(VERSION 3.9)
project(Example)

set(SRC
        ./file_1.c
        ./file_2.c)

add_executable(file_ops ${SRC})
```

The above cmake file is created to make an executable file called `file_ops`.

The source files `file_1.c` and `file_2.c` are part of the `SRC` definition.

The call `add_executable` instructs to create the binary `file_ops` with the given files identified by `SRC`.

The items `project` and `cmake_minimum_required` are not mandatory but cmake warns when we do not keep them. They generally help you guide about what project you are building to and the type of cmake features you are using.

Below is the command to generate the target binary,

```
mkdir build/
cd build/
cmake ..
make
```

`cmake` build generates a lot of intermediate artifacts and dirties the directory. So, better create a directory called `build` and run `cmake` from there.

If you do not have `cmake` installed, on Ubuntu run the following command:

```
sudo apt install make cmake
```

2. Creating library

```
cmake_minimum_required(VERSION 3.9)
project(Example_Lib)

set(LIB_SRC
        ./file_1.c)

add_library(file ${LIB_SRC})
```

The `add_library` instructs the cmake to create a static library `libfile.a`.

Below example creates a shared library.

```
cmake_minimum_required(VERSION 3.9)
project(Example_Lib)

set(LIB_SRC
        ./file_1.c)
```

```
add_library(file SHARED ${LIB_SRC})
```

The `SHARED` specifier creates a `.so` file called `libfile.so`. Without mentioning `SHARED`, the `add_library` generates a `.a` file by default.

3. Linking with libraries

The `target_link_libraries` is used to link the library to the target binary.

Below is one example:

```
cmake_minimum_required(VERSION 3.9)
project(Example_Bin)

set(SRC
        ./file.c)

set(LIB_SRC
        ./file_1.c)

add_library(file ${LIB_SRC})
add_executable(file_ops ${SRC})
target_link_libraries(file_ops file)
```

4. Adding CFLAGS and CPPFLAGS

```
set(CMAKE_C_FLAGS "-Wall -Werror")
set(CMAKE_CXX_FLAGS "-Wall -Werror")
```

The flag `CMAKE_C_FLAGS` and `CMAKE_CXX_FLAGS` takes all the compiler option that can be passed to `gcc/g++` or `clang`.

5. adding C++ standard

```
set(CMAKE_CXX_STANADARD "11")
```

The macro `CMAKE_CXX_STANDARD` sets the C++ standard when compiling.

Data Structures

Linked Lists

Linked list is a chain of elements terminated with a `NULL` pointer.

Each item in chain is called the node. Each node contains data and a pointer to the next item in the list. The last node pointer will be `NULL`.

Linked list structure looks as follows:

```
|--------|    |--------|    |--------|
| item 1 |--->| item 2 |--->| item 3 |--> .... -> NULL
```

```
|--------|     |--------|     |--------|
```

The list always ends with a NULL pointer.

The linked list structure looks as follows.

```
struct linked_list {
    void *elem;
    struct linked_list *next;
}
```

the `data` pointer holds the data and the `next` pointer links to the next element in the list.

Below are some of the general operations on the linked list.

S.No	Name	Description
1	`add`	Add an element to the list at the end
2	`add_head`	Add an element to the list at the head
3	`delete`	delete an element from the list
4	`find`	find an element in the list
5	`count`	count the number of elements in the list
6	`for_each`	iterate through each eement in the list
7	`print`	print all the elements of the list
8	`clean`	clean all the linked list and free up the memory allocated

Lets define two global variables `head` and `tail`.

```
static struct linked_list *head;
static struct linked_list *tail;
```

We use two pointers `head` and `tail`. The `head` is used to iterate over each element from the beginning and `tail` is used to add element at the end.

1. add

Adding an element can be as simple as adding an element at the end.

If there are no elements in the list, add the element at the `head`. If there are elements in the list, add the element after `tail`. When ever an element is added point the `tail` to the last element.

```
int add(void *data)
{
    struct linked_list *node;

    node = calloc(1, sizeof(struct linked_list));
    if (!node) {
```

```
        return -1;
    }

    /* Assign the element */
    node->elem = data;

    if (!head) {
        head = node;
        tail = node;
    } else {
        tail->next = node;
        tail = node;
    }

    return 0;
}
```

Iterating over the elements is simple as using a `while` or `do..while` or `for` loop.

2. add_head

Adding an element at the head is done as follows:

1. Set `node->next` to `head`.
2. Make `head` point to `node`.

```
int add_head(void *data)
{
    struct linked_list *node;

    node = calloc(1, sizeof(struct linked_list));
    if (!node) {
        return -1;
    }

    node->elem = data;

    if (!head) {
        head = node;
        tail = node;
    } else {
        node->next = head;
        head = node;
    }

    return 0;
}
```

3. delete

Deleting an element from list need be considered two possibilities.

1. If the given `elem` is at the `head`.
    1. Save the `head` pointer at `node`.
    2. Move the `head` to the next element.
    3. Free up the saved `node`.
2. If the given `elem` is somewhere in the middle.
    1. Set `prev` and `node` to `head`.
    2. Iterate over each element in the link.
    3. If the `elem` pointers are matched skip the `node`.
    4. Point the `prev->next` to `node->next`, moving the link.
    5. Free up the `node`.

There is a special case here if `elem` to delete is in the middle, the `node` can potentially be the `tail` pointer. Correct the `tail` pointer to old element `prev`.

```
int delete(void *elem)
{
    struct linked_list *node;
    struct linked_list *prev;

    node = head;
    prev = node;

    if (head->elem == elem) {
        head = head->next;
        free(node);
        return 0;
    } else {
        while (node != NULL) {
            if (node->elem == elem) {
                prev->next = node->next;
                if (node == tail) {
                    tail = prev;
                }
                free(node);
                return 0;
            }
            prev = node;
            node = node->next;
        }
    }

    return -1;
}
```

4. find

The find iterates over each element and compares the elem pointer with the given pointer and returns true if both are same.

```
bool find(void *elem)
{
    struct linked_list *node;

    for (node = head; node != NULL; node = node->next) {
        if (node->elem == elem) {
            return true;
        }
    }

    return false;
}
```

There is another way to perform find. This is to call the custom callback that returns true if matched and false if not. Below is an example:

```
bool find(bool (*callback)(void *elem))
{
    struct linked_list *node;
    bool found = false;

    for (node = head; node != NULL; node = node->next) {
        found = callback(node->elem);
        if (found == true) {
            break;
        }
    }

    return found;
}
```

5. count

The count iterates over each link and increments the counter.

```
int count()
{
    struct linked_list *node;
    int n = 0;

    for (node = head; node != NULL; node = node->next) {
        n ++;
    }

    return n;
}
```

The `count` can be a local variable and incremented during `add` and `delete` operations. It can then be returned directly in the `count` function.

6. for_each

The `for_each` iterates over each link and calls the callback. The caller must pass the callback and the caller will get the element as the data.

```
void for_each(void (*callback)(void *elem))
{
    struct linked_list *node;

    for (node = head; node != NULL; node = node->next) {
        if (callback) {
            callback(node->elem);
        }
    }
}
```

7. print

Print the contents of the list by iterating over each element in the list.

```
void print()
{
    struct linked_list *node;

    printf("elements:\n");
    for (node = head; node != NULL; node = node->next) {
        int *data = node->elem;

        printf("%d\n", *data);
    }
}
```

8. clean

Cleaning up of linked list is required to free up the memory used by the linked list.

Here's one way to cleanup a linked list.

1. Take two pointers: `node` and `prev`.
2. `node` points to head and `prev` points to `node`.
3. Move `node` forward. free the `prev` pointer.
4. Set `prev` pointer back to `node`.
5. Repeat until `node` reaches end.

```
void clean()
{
    struct linked_list *node;
```

```
    struct linked_list *prev;

    node = head;
    prev = head;

    while (node) {
        node = node->next;
        free(prev);
        prev = node;
    }
}
```

A `while` loop can be used as well for iteration.

```
struct linked_list *node = head;

while (node) {
    node = node->next;
}
```

However, the `for` loop looks more complete as the assignment, condition and increment are all in one place.

Below is one full example:

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

struct linked_list {
    void *elem;
    struct linked_list *next;
};

static struct linked_list *head;
static struct linked_list *tail;

int add(void *data)
{
    struct linked_list *node;

    node = calloc(1, sizeof(struct linked_list));
    if (!node) {
        return -1;
    }

    node->elem = data;

    if (!head) {
```

```c
        head = node;
        tail = node;
    } else {
        tail->next = node;
        tail = node;
    }

    return 0;
}

int add_head(void *data)
{
    struct linked_list *node;

    node = calloc(1, sizeof(struct linked_list));
    if (!node) {
        return -1;
    }

    node->elem = data;

    if (!head) {
        head = node;
        tail = node;
    } else {
        node->next = head;
        head = node;
    }

    return 0;
}

int delete(void *elem)
{
    struct linked_list *node;
    struct linked_list *prev;

    node = head;
    prev = node;

    if (head->elem == elem) {
        head = head->next;
        free(node);
        return 0;
    } else {
        while (node != NULL) {
```

```
            printf("%p %p\n", node->elem, elem);
            if (node->elem == elem) {
                prev->next = node->next;
                if (node == tail) {
                    tail = prev;
                }
                free(node);
                return 0;
            }
            prev = node;
            node = node->next;
        }
    }

    return -1;
}

bool find(void *elem)
{
    struct linked_list *node;

    for (node = head; node != NULL; node = node->next) {
        if (node->elem == elem) {
            return true;
        }
    }

    return false;
}

int count()
{
    struct linked_list *node;
    int n = 0;

    for (node = head; node != NULL; node = node->next) {
        n ++;
    }

    return n;
}

void for_each(void (*callback)(void *elem))
{
    struct linked_list *node;
```

```
    for (node = head; node != NULL; node = node->next) {
        if (callback) {
            callback(node->elem);
        }
    }
}

void print()
{
    struct linked_list *node;

    printf("elements:\n");
    for (node = head; node != NULL; node = node->next) {
        int *data = node->elem;

        printf("%d\n", *data);
    }
}

void clean()
{
    struct linked_list *node;
    struct linked_list *prev;

    node = head;
    prev = head;

    while (node) {
        node = node->next;
        free(prev);
        prev = node;
    }
}

int main()
{
    int a = 10;
    int b = 20;
    int c = 30;
    int d = 40;
    int e = 50;
    int f = 60;

    add(&a);
    add(&b);
    add(&c);
```

```
    add(&d);
    add(&e);
    add(&f);

    print();

    delete(&a);
    delete(&f);

    add(&f);

    print();

    clean();
}
```

Doubly Linked Lists

Doubly linked list is a chain of elements terminated with a `NULL` pointer.

Each item in chain is called the node. Each node contains data and two pointers. One pointer points to the next elements and another points backwards.

```
        |--------|     |--------|
NULL<---| item 1 |---->| item 2 |--->.... ---> NULL
        |--------|<----|--------|<---.....
```

The doubly linked list structure looks as follows.

```
struct DL {
    void *data;
    struct DL *prev;
    struct DL *next;
};
```

The `data` pointer holds the data and the `prev` pointer points to the previous node in the list and the `next` pointer points to the `next` element in the list.

S.No	Name	Description
1	`dl_add_head`	Add an element to the list at the head
2	`dl_delete_item`	Delete an element from the list
3	`dl_find_fwd`	Find an element in the list with forward iteration
4	`dl_find_rv`	Find an element in the list with reverse iteration
5	`dl_for_each_fwd`	Iterate over all the element with forward iteration

S.No	Name	Description
6	`dl_for_each_rv`	Iterate over all the elements with reverse iteration
7	`dl_count`	Count the number of elements in the list
8	`dl_free_fwd`	Remove all the elements with forward iteration
9	`dl_free_rv`	Remove all the elements with reverse iteration

Similar to the linked list, we create two elements, `head` and `tail`.

```
struct DL *head;
struct DL *tail;
```

1. dl_add_head

```
int dl_add_head(void *data)
{
    struct DL *node;

    node = calloc(1, sizeof(struct DL));
    if (!node) {
        return -1;
    }

    node->data = data;

    if (!head) {
        head = node;
        tail = node;
    } else {
        tail->next = node;
        node->prev = tail;
        tail = node;
    }

    return 0;
}
```

2. dl_delete_item

```
bool dl_delete_item(void *data, void (*callback)(void *data))
{
    struct DL *node;
    struct DL *prev;
```

```
    if (head->data == data) {
        node = head;
        head = head->next;
        head->prev = NULL;
        if (callback) {
            callback(node->data);
        }
        free(node);

        return true;
    } else if (tail->data == data) {
        node = tail;
        tail = tail->prev;
        tail->next = NULL;
        free(node);

        return true;
    } else {
        node = head;
        while (node) {
            if (node->data == data) {
                node->prev->next = node->next;
                node->next->prev = node->prev;
                if (callback) {
                    callback(node->data);
                }
                free(node);

                return true;
            }
            node = node->next;
        }
    }

    return false;
}
```

3. dl_find_fwd

```
bool dl_find_fwd(bool (*callback)(void *data))
{
    struct DL *node;

    for (node = head; node != NULL; node = node->next) {
        if (callback(node->data)) {
            return true;
        }
```

```
    }

    return false;
}
```

4. dl_find_rv

```
bool dl_find_rv(bool (*callback)(void *data))
{
    struct DL *node;

    for (node = tail; node != NULL; node = node->prev) {
        if (callback(node->data)) {
            return true;
        }
    }

    return false;
}
```

5. dl_for_each_fwd

```
void dl_for_each_fwd(void (*callback)(void *data))
{
    struct DL *node;

    for (node = head; node != NULL; node = node->next) {
        if (callback) {
            callback(node->data);
        }
    }
}
```

6. dl_for_each_rv

```
void dl_for_each_rv(void (*callback)(void *data))
{
    struct DL *node;

    for (node = tail; node != NULL; node = node->prev) {
        if (callback) {
            callback(node->data);
        }
    }
}
```

7. dl_count

```
int dl_count()
{
```

```c
    struct DL *node;
    int count = 0;

    for (node = head; node != NULL; node = node->next) {
        count ++;
    }

    return count;
}
```

8. dl_free_fwd

```c
void dl_free_fwd(void (*callback)(void *data))
{
    struct DL *node = head;
    struct DL *prev;

    while (node) {
        prev = node;
        if (callback) {
            callback(prev->data);
        }
        node = node->next;
        free(prev);
    }
}
```

9. dl_free_rv

```c
void dl_free_rv(void (*callback)(void *data))
{
    struct DL *node = tail;
    struct DL *prev;

    while (node) {
        prev = node;
        if (callback) {
            callback(prev->data);
        }
        node = node->prev;
        free(prev);
    }
}
```

Below is a full example of doubly linked list.

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

```c
struct DL {
    void *data;
    struct DL *prev;
    struct DL *next;
};

struct DL *head = NULL;
struct DL *tail = NULL;

int dl_add_head(void *data)
{
    struct DL *node;

    node = calloc(1, sizeof(struct DL));
    if (!node) {
        return -1;
    }

    node->data = data;

    if (!head) {
        head = node;
        tail = node;
    } else {
        tail->next = node;
        node->prev = tail;
        tail = node;
    }

    return 0;
}

bool dl_find_fwd(bool (*callback)(void *data))
{
    struct DL *node;

    for (node = head; node != NULL; node = node->next) {
        if (callback(node->data)) {
            return true;
        }
    }

    return false;
}
```

```
bool dl_find_rv(bool (*callback)(void *data))
{
    struct DL *node;

    for (node = tail; node != NULL; node = node->prev) {
        if (callback(node->data)) {
            return true;
        }
    }

    return false;
}

int dl_count()
{
    struct DL *node;
    int count = 0;

    for (node = head; node != NULL; node = node->next) {
        count ++;
    }

    return count;
}

bool dl_delete_item(void *data, void (*callback)(void *data))
{
    struct DL *node;
    struct DL *prev;

    if (head->data == data) {
        node = head;
        head = head->next;
        head->prev = NULL;
        if (callback) {
            callback(node->data);
        }
        free(node);

        return true;
    } else if (tail->data == data) {
        node = tail;
        tail = tail->prev;
        tail->next = NULL;
        free(node);
```

```
        return true;
    } else {
        node = head;
        while (node) {
            if (node->data == data) {
                node->prev->next = node->next;
                node->next->prev = node->prev;
                if (callback) {
                    callback(node->data);
                }
                free(node);

                return true;
            }
            node = node->next;
        }
    }

    return false;
}

void dl_for_each_fwd(void (*callback)(void *data))
{
    struct DL *node;

    for (node = head; node != NULL; node = node->next) {
        if (callback) {
            callback(node->data);
        }
    }
}

void dl_for_each_rv(void (*callback)(void *data))
{
    struct DL *node;

    for (node = tail; node != NULL; node = node->prev) {
        if (callback) {
            callback(node->data);
        }
    }
}

void dl_free_fwd(void (*callback)(void *data))
{
    struct DL *node = head;
```

```c
    struct DL *prev;

    while (node) {
        prev = node;
        if (callback) {
            callback(prev->data);
        }
        node = node->next;
        free(prev);
    }
}

void dl_free_rv(void (*callback)(void *data))
{
    struct DL *node = tail;
    struct DL *prev;

    while (node) {
        prev = node;
        if (callback) {
            callback(prev->data);
        }
        node = node->prev;
        free(prev);
    }
}

void print(void *data)
{
    printf("val %d\n", *(int *)data);
}

int main()
{
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int i;

    for (i = 0; i < 10; i ++) {
        dl_add_head(&a[i]);
    }

    printf("forward[%d]: \n", dl_count());
    dl_for_each_fwd(print);

    dl_delete_item(&a[0], NULL);
    dl_delete_item(&a[2], NULL);
```

```c
    dl_delete_item(&a[9], NULL);

    printf("reverse[%d]: \n", dl_count());
    dl_for_each_rv(print);

    // dl_free_fwd(NULL);
    dl_free_rv(NULL);

    return 0;
}
```

Circular Linked Lists

Circular lists is similar to the linked list and the difference is that the last node points the first node.

```
    |--------|      |--------|          |--------|
    | item 1 |----->| item 2 |----> .... | item n |
    |--------|      |--------|          |--------|
       ^                                     |
       |--------------------------------<<----|
```

Circular Doubly Linked Lists

Stack

Stacks are another data structure where the data entered first is data retrieved last or data entered in last is the data retrieved first.

```
|----------------|
|     item n     |  <-- Add and Retrieve
|----------------|
|     ...        |
|----------------|
|     item 2     |
|----------------|
|     item 1     |
|----------------|
```

Below are some operations that can be done with the stack data structure.

S.No	Name	Description
1	`top`	Get the top element of thes stack
2	`push`	Push an element into the stack
3	`pop`	Pop an element out of the stack
4	`empty`	Clear elements in the stack

S.No	Name	Description
5	`size`	Get the total size of elements in the stack

Stacks can be implemented with Linked lists as well.

```
struct stack {
    void *elem;
    struct stack *next;
};

static struct stack *head;
static int count;
```

Elements are added at `head` and removed at `head`.

1. top

```
void *top()
{
    void *elem = NULL;

    if (head) {
        elem = head->elem;
    }
    return elem;
}
```

2. push

```
int push(void *elem)
{
    struct stack *node;

    node = calloc(1, sizeof(struct stack));
    if (!node) {
        return -1;
    }

    node->elem = elem;

    if (!head) {
        head = node;
    } else {
        node->next = head;
        head = node;
    }
```

```
    count ++;

    return 0;
}
```

3. pop

```
void *pop()
{
    struct stack *node;
    void *elem = NULL;

    if (head) {
        elem = head->elem;
        node = head;
        head = head->next;
        free(node);
        count --;
    }

    return elem;
}
```

4. empty

```
void empty(void (*callback)(void *elem))
{
    struct stack *node;
    struct stack *prev;

    node = head;
    prev = head;

    while (node) {
        prev = node;
        node = node->next;
        if (callback) {
            callback(prev->elem);
        }
        free(prev);
    }
}
```

5. size

```
int size()
{
    return count;
}
```

Below is one full implementation of stack example:

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

struct stack {
    void *elem;
    struct stack *next;
};

static struct stack *head;
static int count;

void *top()
{
    return head->elem;
}

int push(void *elem)
{
    struct stack *node;

    node = calloc(1, sizeof(struct stack));
    if (!node) {
        return -1;
    }

    node->elem = elem;

    if (!head) {
        head = node;
    } else {
        node->next = head;
        head = node;
    }

    count ++;

    return 0;
}

void *pop()
{
    struct stack *node;
    void *elem = NULL;
```

```
    if (head) {
        elem = head->elem;
        node = head;
        head = head->next;
        free(node);
        count --;
    }

    return elem;
}

int size()
{
    return count;
}

void empty(void (*callback)(void *elem))
{
    struct stack *node;
    struct stack *prev;

    node = head;
    prev = head;

    while (node) {
        prev = node;
        node = node->next;
        if (callback) {
            callback(prev->elem);
        }
        free(prev);
    }
}

int main()
{
    int a = 1;
    int b = 2;
    int c = 3;
    int d = 4;
    int e = 5;
    int f = 6;

    push(&a);
    push(&b);
    push(&c);
```

```
    push(&d);
    push(&e);
    push(&f);

    printf("size : %d\n", size());

    while (1) {
        int *elem = pop();
        if (elem == NULL) {
            break;
        }
        printf("%d\n", *elem);
    }

    empty(NULL);

    return 0;
}
```

Queue

The queue adds elements at the last and retrieves them at the first. For this we use two pointers `head` and `tail`.

```
|-------------|
|   item 1    |  <-- First (Remove elements)
|-------------|
|   item 2    |
|-------------|
|    ...      |
|-------------|
|   item n    |  <-- Last (Add elments)
|-------------|
```

The below structure definition is as follows:

```
struct queue {
    void *elem;
    struct queue *next;
};

static struct queue *head;
static struct queue *tail;
static int count;
```

The following are operations of queue:

S.No	Name	Description
1	`front`	Get the front element in the queue
2	`back`	Get the back element in the queue
3	`empty`	Empty the queue
4	`size`	Get the number of elements in the queue
5	`push`	Push an element in the queue
6	`pop`	Pop an element from the queue

1. front

```
void *front()
{
    void *elem;

    if (head) {
        elem = head->elem;
    }

    return elem;
}
```

2. back

```
void *back()
{
    void *elem;

    if (tail) {
        elem = tail->elem;
    }

    return elem;
}
```

3. empty

```
void empty(void (*callback)(void *elem))
{
    struct queue *node;
    struct queue *prev;

    node = head;
    prev = head;

    while (node) {
        prev = node;
        node = node->next;
```

```
        if (callback) {
            callback(prev->elem);
        }
        free(prev);
    }
    count = 0;
}
```

4. size

```
int size()
{
    return count;
}
```

5. push

```
int push(void *elem)
{
    struct queue *node;

    node = calloc(1, sizeof(struct queue));
    if (!node) {
        return -1;
    }

    node->elem = elem;

    if (!head) {
        head = node;
        tail = node;
    } else {
        tail->next = node;
        tail = node;
    }

    return 0;
}
```

6. pop

```
void *pop()
{
    struct queue *node;
    void *elem = NULL;

    if (head != NULL) {
        node = head;
        elem = node->elem;
```

```
        head = head->next;
        free(node);
    }

    return elem;
}
```

Below is an example:

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

struct queue {
    void *elem;
    struct queue *next;
};

static struct queue *head;
static struct queue *tail;
static int count;

void *front()
{
    void *elem;

    if (head) {
        elem = head->elem;
    }

    return elem;
}

void *back()
{
    void *elem;

    if (tail) {
        elem = tail->elem;
    }

    return elem;
}

void empty(void (*callback)(void *elem))
{
    struct queue *node;
    struct queue *prev;
```

```
    node = head;
    prev = head;

    while (node) {
        prev = node;
        node = node->next;
        if (callback) {
            callback(prev->elem);
        }
        free(prev);
    }
    count = 0;
}

int size()
{
    return count;
}

int push(void *elem)
{
    struct queue *node;

    node = calloc(1, sizeof(struct queue));
    if (!node) {
        return -1;
    }

    node->elem = elem;

    if (!head) {
        head = node;
        tail = node;
    } else {
        tail->next = node;
        tail = node;
    }

    return 0;
}

void *pop()
{
    struct queue *node;
    void *elem = NULL;
```

```c
    if (head != NULL) {
        node = head;
        elem = node->elem;
        head = head->next;
        free(node);
    }

    return elem;
}

int main()
{
    int a = 10;
    int b = 20;
    int c = 30;
    int d = 40;
    int e = 50;
    int f = 60;

    push(&a);
    push(&b);
    push(&c);
    push(&d);
    push(&e);
    push(&f);

    printf("size : %d\n", size());
    printf("Front: %d\n", *(int *)front());
    printf("Back:  %d\n", *(int *)back());

    while (1) {
        int *elem = pop();
        if (!elem) {
            break;
        }
        printf("%d\n", *elem);
    }

    empty(NULL);

    return 0;
}
```

Ring Buffer

Tree

Merkel Trees

Hash Tables

Search and Sorting