Contents

Programming in C and C++

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 no use of 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,

```
/*
 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	-

S.No	type	description	ranges
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
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.

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	^	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;
    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.

```
#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.

```
#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.

```
#include <stdio.h>

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

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

    return 0;
}
```

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

```
#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;
}
```

The output is :

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

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

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

3. Heap segment

This is the space where the dynamic memory can be allocated.

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.

```c
#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.

```
#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.

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

```c
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.

```c
#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.

```
#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.

```
#include <stdio.h>

int main()
{
    int i = 0;

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

```
        i ++;
        goto begin;
    }

    return 0;
}s
```

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,

```
int init_1()
{
    ...
    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,

```
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.

```
#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.

```
#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.

```
#include <stdio.h>

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.

```
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,

```
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:

```
#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.

```
int a[10];

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

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

2. Two Dimensional Arrays

Two dimensional arrays are represented as follows.

```
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.

```
#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,

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

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

```
#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()
{
    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,

```
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,

```
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()
{
    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.

```
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.

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

The example of `strcpy` is as follows.

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

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.

```
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

```
#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);

    return 0;
}
```

```
#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

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

```c
#include <string.h>

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

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

    return 0;
}
```

6. memcmp

```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;
}
```

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

Library Functions to convert a string to other types 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()
{
    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

```
#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

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

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

    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".

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

A pointer of any type is possible.

```
int *p;
```

declares an integer pointer.

```
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.

```
#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.

```
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

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;
}
```

Dynamic Memory Allocation

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`.

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

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.

```
#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;
}
```

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;
}
```

Bit fields

Structure packing

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

Command line arguments (argc, argv)

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.

```
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

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

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>

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.

```
char msg[1024];

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

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

3. Writing to a file

```
#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

5. Number of characters in a file

6. Finding the file size

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

I/O operations

1. Using fscanf, fgets and fprintf

C++ programming

New operators in C++

1. The Reference (&) operator.

New keywords in C++

constexpr

explicit

auto

Allocating and Freeing

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..);
};
```

Constructors and Destructors

Copy constructor

Move constructor

this pointer

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;
using namespace std;

int main()
{
    double n;

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

    return 0;
}
```

Another way of using it as follows:

```
#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 funbction, we prefix the members with the namespace followed by the :: operator.

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++.

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:

```
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.

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

Below program covers almost all the `noexcept` cases:

```
#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";
```

```
    }
}
```

Arrays

`std::array` defines an array type.

Strings

Vectors

Lists

Queues

Maps

shared_ptr, unique_ptr

File systems

Threads

C++ implements abstraction of threads based upon the pthreads. The class `std::thread` defines the thread interface.

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;
}
```

Lets see below example, that creates two threads.

```cpp
#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

Conditional Variables

Polymorphism

Inheritance

Simple Inheritance

Multiple inheritance

Abstract Classes

```
class abstract_class {
    public:
        virtual function_return function_prototype(parameters..) = 0;
}
```

Templates

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;
}
```

Appendix B

Scoppe and Lifetime

Use cases

Safe Queue

Design Patterns

Factory design pattern

Singleton 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.

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`.

3. Linking with libraries

4. Adding CFLAGS and CPPFLAGS

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;
}
```

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;
    }
}
```

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) {
        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 doubly_linked_list {
    void *data;
    struct doubly_linked_list *prev;
    struct doubly_linked_list *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	`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	`insert`	insert an element at a particular position in the list
5	`find`	find an element in the list
6	`count`	count the number of elements in the list
7	`for_each`	iterate through each eement in the list

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 |
    |--------|      |--------|           |--------|
       ^                                     |
       |-------------------------------<<----|
```

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     |
|----------------|
|     ...        |
```

```
|----------------|
|     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
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`.

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;
```

```
    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

Search and Sorting