
Programming
with C and C++
By Devendra Naga

Contents

Loops

Loop statements are another programming construct that allow executing something for a certain number of times.

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
}
```

Example.22 While condition

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;
}
```

Example.23 While loop example

Sometimes it doesn't have to be a condition, a variable should be enough.

The following statements are valid in this regard.

```
bool is_set = true;
while (is_set) { } // valid

int i = 1;
while (i) { } // valid
```

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;
}
```

Example.24 while loop and break statement

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.

Sometimes, infinite loops are required but with OS supported waiting mechanism that does not cause much of a CPU load and provides better resource utilization between other processes.

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;
}
```

Example.25 For loop

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;
}
```

Example.26 For iteration

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;
}
```

Example.27 Infinite for loop

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

To stop an infinite loop program, press `Ctrl + C` key combination on the terminal within Linux.

3. do while loop

The `do..while` loop is similar to the `while`. The statement within the `do..while` executes first and the condition is checked for validity. Below is an example.

```
#include <stdio.h>

int main()
{
    int i = 0;

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

    return 0;
}
```

Example.28 do..while loop

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

The `do..while` statement is generally used in cases where the code must be executed at least once no matter the condition satisfies for the first time.

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;
}
```

Example.29 goto loop

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;
}
```

Example.30 goto usecase

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 (since indexing starts at 0), 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.

```
int a[];
```

is invalid because array without number of elements defined is invalid syntax. However,

```
int a[] = {1, 2, 3, 4};
```

is still valid and compiler assumes that the array contains 4 elements.

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");

    return 0;
}
```

Example.31 array iteration

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;
}
```

Example.32 sizeof array

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;
}
```

Example.33 number of elements in an array

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;
}
```

Example.34 Initializing array

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.

Another way to set array elements is as follows:

```
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]);
}
```

Example.35 iterating array

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

copying array elements

Copying one array to another is simple as iterating over each element and copying one element to another.

Below is one example,

```
#include <stdio.h>

int main()
{
    int a1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int a2[10];
    int i;

    for (i = 0; i < sizeof(a1) / sizeof(a1[0]); i ++) {
```

Below is an example. Download it here.

```
#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 in below code snippet.

Download it here.

```
#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 the format `0x`.

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

Converts string to double.

The function prototype is as follows:

```
double strtod(const char *ptr, char **end_ptr);
```

Below is the example:

```
#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

Converts string to unsigned long.

Below is the prototype.

```
unsigned long strtoul(const char *nptr, char **err_ptr, int base);
```

Below is one example:

```
#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

Pointer is a type that associates with an address.

A variable that is stored in the memory is associated with an address. Pointer stores the address of a variable or even a pointer.

Pointers can be associated with integer data types, floats, doubles, strings, arrays, structures and even the functions.

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.

A pointer can be assigned `NULL` stating that it points to no address. This `NULL` is different from the null terminating character `\0`.

The null terminating character can be applied only to the strings. While the `NULL` pointer is applied to the pointers.

In Standard library, the definition of the `NULL` is a macro. Something like the following:

```
#define NULL (void *)0
```

This is more generically, a value 0 type casted to a void pointer.

The `NULL` pointer is used to inform that the pointer points to nothing. Also deferencing the `NULL` pointer results in a abrupt program stop or a segmentation fault.

The below statement,

```
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

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.

To typecast back to the type generally, the typecast need to be used.

```
void *p;
int *a;

a = (int *)p;
printf("val %d\n", *a);
```

The above code can be rewritten as,

```
void *p;
int a = 10;

p = &a;
printf("val %d\n", *(int *)p);
```

Here the void pointer `p` is typecasted to integer pointer using the `int *` and then dereferenced using the `*` operator.

Typecasting can be used for the structure pointer as well.

```
struct S {
    int p;
};

struct S s = {
    .p = 3,
};

void *p = &s;

struct S *r = (struct S *)p; // typecast back to struct S
```

The above code snippet can be rewritten as follows,

```
#include <stdio.h>

struct S {
    int p;
};

struct S s = {
    .p = 3,
};

int main()
{
    void *p = &s;

    printf("%d\n", ((struct S *)p)->p);

    return 0;
}
```

The below implementation of program results in compiler error because of the de-reference of void pointer.

```
#include <stdio.h>

int main()
{
    int p = 3;
    void *a;

    a = &p;

    printf("%d\n", *a); // compiler error.. de-referencing void *

    return 0;
```

```
}
```

Another way to do is the following.

```
#include <stdio.h>

int main()
{
    int p = 3;
    void *a;

    a = &p;

    printf("%d\n", *(int *)a); // typecasting implictly and then de-referencing the pointer

    return 0;
}
```

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. In C, the void pointer can point to anything or any pointer type can be assigned to the void pointer. No explicit typecast is required.

Typecast is required in few places such as typecasting between different pointer types. For example consider the following program,

“c #include <stdio.h>

struct S { int p; };

struct S s = { .p = 3, };

int main() { void *p = &s; int* p1 = &s;

```
printf("%d\n", ((struct S *)p)->p);

return 0;
```

}

The above program has the following statement,

```c
int *p1 = &s;
```

Which results in the following warning:

```
deref_structs.c:14:15: warning: initialization of ‘int *’ from incompatible pointer type ‘st
   14 |     int *p1 = &s;
      |               ^
```

This needs to be explicitly typecasted if the objective is to really typecast from one type to another. In general this is simply not the case. The right approach to this is as follows,

```c
int *p1 = (int *)&s;
```

Below is one example use of pointers with arrays,

```c
#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.

```c
#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;
}
```

When passing an array to a function, the caller takes it as a pointer input. This is called as array decaying into a pointer. When such thing happens, calling `sizeof` on the pointer gives you `4` or `8` bytes that is the size of the pointer on the architecture. In general it is wise to pass the number of elements of the array as argument to the function call.

For example,

```
int f(int *a, int a_len)
{
}

int main()
{
    int a[10];

    f(a, 10);
}
```

Pointer Arithmetic

Arithmetic operations are allowed on pointers, but they generally are dangerous if not done correct. The danger is that a non-allocated / non-reserved address being accessed during an arithmetic operation results in either unknown code execution (resulting to using this portion of code for viruses or exploits) or if lucky leads to a program crash.

```
#include <stdio.h>

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

    while (*p != '\0') {
        printf("'%c'", *p);
        p ++;
    }
```

```
    printf("\n");

    return 0;
}
```

The above code checks for the `\0` character and iterates through each character in the string `p`. The operator `++` allows us to move to the next character.

When a `++` is performed on a character pointer, since the type the pointer is pointing to, is `char` the next address is the next byte.

The programs written in the **strings** can be rewritten with pointers. Below is an example of `strlen` function rewritten using pointer arithmetic.

```
#include <stdio.h>

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

    if (!str) {
        return -1;
    }

    while (*str != '\0') {
        len ++;
        str ++;
    }

    return len;
}

int main()
{
    char *str = "this is string";

    printf("len %d\n", string_len(str));

    return 0;
}
```

When the same pointer points to an integer, the next address will be the next 4 bytes. Below is an example,

```
#include <stdio.h>

int main()
{
    int a[10] = {1, 2, 3, 4, 5, 6};
```

Recap about variables and scope

auto type

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

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

Pointers to functions are similar to pointer to the variables. Functions also have addresses.

```
int (*fptr)(void);
```

Defines a function pointer, that accepts no arguments and returns an integer.

```
int (*fptr)(char *fmt, ...);
```

Defines a variable argument function.

Function pointers in software are useful to write abstractions.

Consider the following case,

1. An Ethernet phy product A provides capability for transmit, receive functionality for internet working.
2. Another Ethernet phy product B provides same capabilities for internet working.

So to the user it does not matter which hardware is being used downside. All he care about is the internet.

An Operating system programmer must have to implement something that must abstract the two interfaces to provide a uniform interface to the user. This generally is where function pointers come in.

Lets convert that above feature into software.

```
struct ethernet_driver {
    int (*transmit)(uint8_t *packet, int packet_size);
    int (*receive)(uint8_t *packet);
};
```

The structure `ethernet_driver` provides an interface to the upside user and the downward phy device.

```
enum eth_driver_list {
    A,
    B,
};

struct ethernet_driver *e;
enum eth_driver_list list[2] = { A, B };

e = probe_for(&list, 2);
```

Here, the function `probe_for` looks for hardware registers which hardware is inserted, whether its A or B. If its A, it would return the `ethernet_driver` pointer of the driver A, otherwise it returns B.

in A_eth_Driver.c:

```
int a_transmit(uint8_t *packet, int packet_size);
int a_receive(uint8_t *packet);

static const struct ethernet_driver A_driver = {
    .transmit = a_transmit,
    .receive = a_receive,
};

struct ethernet_driver *get_A_functionality()
{
    return &A_driver;
}
```

in B_eth_Driver.c:

```
int b_transmit(uint8_t *packet, int packet_size);
int b_receive(uint8_t *packet);

static const struct ethernet_driver B_driver = {
    .transmit = b_transmit,
    .receive = b_receive,
};

struct ethernet_driver *get_B_functionality()
{
    return &B_driver;
}
```

in probe.c:

```
#define A_REG 0x000A0000
#define B_REG 0x000B0000

struct ethernet_driver *probe_for(enum eth_driver_list *list, int len)
{
    int i;

    for (i = 0; i < len; i ++) {
        if (list[i] == A) {
            if (match_reg(A_REG)) {
                return get_A_functionality();
            }
        } else if (list[i] == B) {
```

```
            if (match_reg(B_REG)) {
                return get_B_functionality();
            }
        }
    }

    return NULL;
}
```

The functionality of the caller for `probe_for` remain the same even if a new driver gets added, it will have a new driver file implementing the new driver and the `probe_for` function update to call the driver.

The most important observation here is that the caller can directly perform a call to `eth->transmit` and `eth->receive` without knowing the underlying driver functionality.

This is called abstraction. This feature is very well defined within the language in C++ as Abstract classes. We will describe this in more details.

Array of function pointers

The array of function pointers have the below syntax.

```
int (*fptr[4])(void); // defines array of 4 function pointers
```

To be much simpler, one can `typedef` the function pointer and define the arrays.

```
typedef int (*fptr)(void); // defines a function pointer

fptr f[4]; // defines 4 function pointers
```

Below is an example of array of function pointers,

```
#include <stdio.h>

int f()
{
    static int count = 0;

    printf("F called %d\n", count);

    return ++ count;
}

typedef int (*fptr_f)(void);


int main()
{
    fptr_f fptr[4];
    int i;

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

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

    return 0;
}
```

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; // define book typedef

typedef struct book book_t; // define typedef in a new line

typedef struct { // define typedef without naming
} 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.

One can also use macros for better sounding names.

```
#define Book_Info struct book
```

But doing this generally avoided although its possible.

Macros though can be used this way, their whole purpose is for naming constants or writing small function like macros.

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.

```
struct S1 {
    int v;
};

struct S2 {
    struct S1 s1;
};

...
struct S2 s2;

s2.s1.v = 3;

printf("v: %d\n", s2.s1.v);
```

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,

```
#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;
}
```

To initialize an array of structures, one can use the following approach.

```
struct A {
    int val;
    double val_d;
} a[] = {
    {
        .val = 3,
        .val_d = 3.1,
    },
    {
        .val = 4,
        .val_d = 3.2,
    },
    {
        .val = 5,
        .val_d = 3.3,
    }
};
```

When initializing an array statically the array subscript is not required.

In general, if the array never changes during the program lifetime, then the

variable can be set `const`.

```
const struct A a[] = { ... };
```

Array of structures

Arrays of structures is possible as well.

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

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

They can be iterated just like arrays.

```
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;
    }
}
```

Allocating structures

Structures are allocated in the same way as any variable.

```
struct S {
    int a;
    int b;
};

struct S *s;

s = calloc(1, sizeof(struct S));
```

the access to the member variables can then be done using the `->` operator.

```
s->a = 3;
s->b = s->a;
```

The freeing is same as well. A call to `free(s)` would free up the allocated memory.

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.

While we can have function pointers in structures, but we cannot have functions in structure. This is not allowed in C. However, C++ allow this grouping of data and operations.

The below structure is incorrect and results in error.

```
#include <stdio.h>

struct S {
    int f(void);
};

int main()
{
}
```

6. nullptr

C++11 onwards the standard defines `nullptr` that can be assigned to any type of pointer. When assigned, the compiler implicitly takes care of assigning the null pointer. The `nullptr` does not point to 0 or `(void *)0`.

For example the following statements are still valid in C++.

```
int *p = 0; // old use of 0 for null
int *p = NULL; // NULL pointer use in C
```

In general it is extremely difficult to really find the type of the variable when its using `auto` type deduction.

```
auto res = get_data();

if (res == 0) { // assuming 0 here is a null pointer

}
```

However, the type understood could be a `NULL` if the `get_data` is actually returning a `NULL` pointer.

```
if (res == nullptr) { // always guaranteed the res is a null pointer
}
```

Data struture pointers can be assigned to `nullptr` as well.

```
struct S {
    int a;
};

S *s1 = NULL; // still valid
S *s2 = nullptr; // valid
```

In some of the standard types such as `shared_ptr`, `unique_ptr` assigning 0 could result in ambiguous result. The `shared_ptr` and `unique_ptr` are part of the standard template library. They will be discussed further in their corresponding sections.

```
std::shared_ptr<S> s1 = 0; // may work
std::shared_ptr<S> s2 = nullptr;
```

7. Functions with default arguments

C++ allows functions with default arguments. For example,

```
int f(int a, int b = 10)
{
    return a + b;
}
```

In the above function `f`, the caller does not have to pass the second argument so the call `f(10)` is valid.

Or if explicitly wanting to pass the value, then the second argument can be passed. For example `f(3, 3)` is valid and the argument `b` takes value `3`.

```
#include <iostream>

int f(int a, int b = 10)
{
    return a + b;
}

int main()
{
    int r = f(10);

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

    r = f(3, 3);

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

    return 0;
}
```

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.

The `sizeof` on classes would give the size of the variables (excluding member functions).

```
#include <iostream>
```

```
class S {
    public:
        int get() { return a; }

    private:
        int a;
        int p;
        double r;
};

int main()
{
    std::cout << "size: " << sizeof(S) << std::endl;

    return 0;
}
```

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()
{
```

```
    switch (l) {
        case Languages::Telugu:
            return l = Languages::Tamil;
        break;
        case Languages::Tamil:
            return l = Languages::English;
        break;
        case Languages::Hindi:
            return l = Languages::Telugu;
        break;
        default:
            return l = Languages::Telugu;
        break;
    }
}

int main()
{
    Languages l1 = Languages::Telugu;
    Languages l2 = Languages::Tamil;

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

    if (l1 < l2) {
        std::cout << "l1 is less than l2" << std::endl;
    }

    if (Languages::Hindi < 100) {
        std::cout << "hindi less than 100" << std::endl;
    }

    std::cout << l1 << std::endl;

    ++l1;

    std::cout << l1 << std::endl;

    std::cout << std::is_enum_v<Languages> << 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>

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

```
#include <iostream>

struct S {
    S(int r) {
        if (r == 0) {
            throw std::runtime_error("r is 0");
        }
    }

    int a;
};

int main()
{
    try {
        S r1(0);
    } catch (std::runtime_error &r) {
        std::cout << "exception: " << r.what() << std::endl;
    }

    S r2(3);

    return 0;
}
```

```
#include <iostream>
#include <exception>

class S : public std::out_of_range {
    public:
        S(const char *err) : std::out_of_range(err) { }
        ~S() { }

        int a;

        void set(int v) {
            if (v > 10) {
                throw *this;
            }
            a = v;
        }

        virtual const char *what() const noexcept {
            return "a value out of range";
```

```
        }
};

int main()
{
    S r1("");

    try {
        r1.set(11);
    } catch (S &r) {
        std::cout << r.what() << std::endl;
    }

    return 0;
}
```

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";
```

```
    }
}
```

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++.

std::pair

`std::pair` defines a pair of values. The usecases are when returning more than one variable from a function or passing to a function as key-value pair.

`std::pair` can be used to construct a pair or `std::make_pair` can be used as well.

```
auto p = std::pair<std::string, int>("test", 1); // constructs std::pair type in p
auto p = std::make_pair<std::string, int>("test", 1); // makes a pair of type std::pair
```

Below is one example of the usecase for `std::pair`:

```
#include <iostream>

void f(std::pair<const std::string, int> p)
{
    std::cout << "first: " << p.first << " second: " << p.second << std::endl;
}

int main()
{
    f(std::make_pair("test", 1));
    f(std::make_pair<std::string, int>("test", 2));
    f(std::pair("test", 1));
}
```

`std::pair` also exposes operators `==`, `!=`, `<` and `>`. One can use certain operations for comparison. Here's one example: Download it here

```
#include <iostream>

int main()
{
    auto r = std::pair<std::string, int>("test", 1);
    auto r1 = std::pair<std::string, int>("test", 1);

    std::cout << "r == r1: " << (r == r1) << std::endl;

    return 0;
}
```

`std::pair` can be implemented with templates. See usecases section for the implementation of `std::pair`.

std::initializer_list

std::bitset

```
    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.

Threads can be detached with `.detach` method. However, at some point the main thread has to wait. Without it the program does not execute or the threads will not run.

```
#include <iostream>
#include <thread>

void t1()
{
    printf("in t1\n");
}

int main()
{
    std::thread t(t1);

    t.detach();

    printf("in main thread\n");
    while (1) {
        std::this_thread::wait_for(std::chrono::milliseconds(100));
    }
}
```

In general, when threads are created by the operating system, the execution totally depends on the scheduler.

Mutexes

Mutex is a synchronization primitive which sequences out the shared data accesses.

For example if more than one user (reader / writer) of the shared data is accessed the read or write results may not be as expected. A write may have happened in the middle and did not completely finish due to the reader accessing it. The same could happen while reader accessing the shared data the writer may have overwritten it, resulting in the incorrect read result.

Mutex provides basic lock and unlock primitives. A `lock` primitive locks the section of the code that performs operation (read/write) on the data. An `unlock` primitive unlocks the section of the code that was locked before using `lock` primitive. Below is one example,

```
std::mutex m;

m.lock();
// perform data modification / access data
m.unlock();
```

Mutexes are defined in C++11 standard library. They are similar to the `pthread_mutex` types in the POSIX.

`std::mutex` class defines the mutex. It has the following member functions.

S.No	Member function name	Description
1	`lock`	locks the mutex
2	`unlock`	unlocks the mutex
3	`try_lock`	tries to lock the mutex, returns if the mutex is not available
4	`native_handle`	returns the underlying implementation defined handle

A `lock` call would wait indefinitely until the other thread performs a `unlock` on the thread object. A second `lock` call would result in thread waiting on `lock` itself again. This is called deadlock.

Below is one example,

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex m;
```

```
void thread_function()
{
    printf("in thread\n");

    printf("aquiring lock\n");
    m.lock();
    printf("try acquiring lock again\n");
    m.lock();
    printf("acquired double lock\n");
}

int main()
{
    std::thread t(thread_function);

    t.join();
    return 0;
}
```

The above program acquires the lock at the first `m.lock()` statement and tries to acquire the second lock and waits on it indefinitely until someone unlocks.

A `try_lock` approach to the locking would generally resolve the problem. But in general a double lock acquisition is not preferred in any situations.

A `try_lock` would fail if a lock has been already acquired. So the code can safely return instead of getting into deadlock.

Below is one example,

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex m;

void thread_function()
{
    printf("try acquiring the lock\n");

    m.lock();
    printf("took the lock.. trying again to acquire the lock\n");

    if (m.try_lock()) {
        printf("acquired lock again..\n");
        m.unlock();
    } else {
```

```
        printf("failed to acquire the second lock\n");
    }
}

int main()
{
    std::thread t(thread_function);

    t.join();

    return 0;
}
```

Below is another example of two threads contending to use the `count` variable with `mutex` used to serialize the access.

```
#include <iostream>
#include <thread>
#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

In C++ conditional variable is similar to the `pthread_cond` types.

The `std::condition_variable` is defined in C++11 standard. Below is an example of using condition variable.

```
#include <iostream>
#include <queue>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex mut;
std::condition_variable cond;
std::queue<int> ints;

void t1()
{
    while (1) {
        std::unique_lock l(mut);
        cond.wait(l);
        if (ints.size() != 0) {
            printf("deque %d\n", ints.front());
            ints.pop();
        }
    }
}

int main()
{
    std::thread t(t1);
    int i = 1;

    t.detach();
    while (1) {
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        std::unique_lock l(mut);
        ints.push(i);
        cond.notify_one();
        i ++;
    }
}
```

In most cases a single producer and single consumer does not require a condition variable. In general calling `mutex.lock()` would keep the calling thread in sleep.

However if more than one thread is waiting on the data (more consumers), a

condition variable is required.

Calling `cond.wait()` on a mutex effectively lets the thread wait on the condition. Calling `cond.notify_one()` in the producer thread will effectively signal the thread waiting on `cond.wait()`, thus waking it up effectively.

In a simple world, the producer generates the data, takes the mutex lock, inserts the data into the queue. It then notifies the thread using `cond.notify_one()`. The consumer waits on the `cond.wait()` on the mutex. The `cond.wait()` returns soon as `cond.notify_one()` or `cond.notify_all()` is called on the same condition variable.

The thread checks if the number of entries in the queue are non zero and uses the print statement to print the first entry in the queue. The thread then calls `queue.pop()` to pop the item from the front of the queue.

`cond.wait()` accepts the mutex of type `std::unique_lock`.

std::unique_lock

The `std::unique_lock` provides a way to always perform the unlock no matter what. The `unique_lock` is scope driven. If the lock goes out of scope (it can be within a braces `{` and `}` or within the function. The destructor of `unique_lock` calls `.unlock` method. Thus it guarantees that the mutex is always unlocked.

Here is one example of it:

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex count_lock;
int count = 0;
void thread_function()
{
    while (1) {
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        std::unique_lock l(count_lock);
        count ++;
        printf("count in thread %d\n", count);
    }
}

int main()
{
    std::thread t1(thread_function);

    while (1) {
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
```

```
        std::unique_lock l(count_lock);
        count ++;
        printf("count in main thread %d\n", count);
    }
}
```

std::timed_lock

`std::timed_lock` is used in cases where the thread must not wait for the lock to be acquired and need to do other job periodically or so. In such cases `std::timed_lock` helps.

The member functions of `std::timed_lock` have same members as the `std::mutex` with addition of `try_lock_for` member function which accepts timeout argument of type `std::chrono`.

Below is one example,

The code waits every one second and tries again to acquires a lock for 3 times.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>

std::timed_mutex t_m;
int val = 0;
void thread_function()
{
    using namespace std::literals::chrono_literals;
    int retry = 0;

    while (retry <= 3) {
        if (t_m.try_lock_for(1s)) {
            printf("val %d\n", val);
            std::this_thread::sleep_for(1s);
            t_m.unlock();
            break;
        } else {
            printf("failed to acquire lock\n");
            retry ++;
        }
    }
}

int main()
{
```

```
    using namespace std::literals::chrono_literals;

    t_m.lock();
    std::thread t1(thread_function);
    std::this_thread::sleep_for(2s);
    val ++;
    t_m.unlock();
    t1.join();
}
```

Creating a Vector of Threads

A vector of thread objects can be created in a loop then added to the `std::vector` object.

Pushing the thread object into the vector does not really add the object as the thread's destructor is called upon. Instead of creating a thread object a shared pointer of the thread object is created. Below is one example,

```
#include <iostream>
#include <vector>
#include <thread>
#include <memory>

void thread_func(int a)
{
    printf("in thread %d\n", a);
}

int main()
{
    std::vector<std::shared_ptr<std::thread>> threads;
    int i;

    for (i = 0; i < 10; i ++) {
        std::shared_ptr<std::thread> thread;

        thread = std::make_shared<std::thread>(thread_func, i);
        threads.push_back(thread);
    }

    for (auto thread : threads) {
        thread->join();
    }

    return 0;
}
```

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()`.

```cpp
#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;
    }
}
```

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_ --;
                }
            }
        }
};

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()
```