C Programming (Low Level Programming)

Language Features: in C++ not in C

- Classes, objects, inheritance, polymorphism, etc.
- Structs having static data members and functions
  - Structs in C are strictly collections of data members (plain old data).
- Namespace control
  - C only has a single top-level namespace.
- Overloading
- Exception handling.
  - C has a system with `<setjump.h>` but it has all sort of trouble.
- Builtin Memory allocation (`new` and `delete` operations)
  - Instead, you use the *library functions* `malloc()` and `free()`, but they aren't built into the language itself.
- `cin`/`cout`.
  - In C, you use IO with the library functions under the `<stdio.h>` header.

Compilation Process (before execution)

1. Preprocessor: incorporate headers and `#define`s, expand macros, strip comments.
2. Compiler: convert code to assembly language.
3. Assembler: assemble code to object files/machine code (code is now in true binary).
4. Linker: produce the final executable; if other libraries are used via `#include` directives, link them first.

Fundamental Compilation Process be like:

```
$ gcc -E foo.c > foo.i # (generate preprocessed code)     (preprocessing)
$ gcc -S foo.i        # (generate assembly code)         (compiling)
$ gcc -c foo.s        # (generate object code)           (assembling)
$ gcc foo.o -o foo    # (generate executable file)       (linking)
```

- `gcc -E foo.c > foo.i` generates `foo.i`

  It just runs the macro part, basically the same as source code but replace macron parts.

  FROM (in `foo.c` (a **C file**))

  ```
  #define X 2
  f(X)
  ```

  TO (in `foo.i`)

  ```
  f(2)
  ```

- `gcc -S foo.i` generates `foo.s`

  like

  ```
  movq $27, %ordi
  call f
  movl %eax, %ebx
  ```

  which is not really machine code but a text file

- `gcc -c foo.s` generates `foo.o`

  which is a `foo.s` but in binary format `foo.o` (hexidecimal)

  ```
  FC|D1|1B
  F1|?
  FC|03|AB
  ```

  but we cannot run it still due to the `?` because we don't know where it is ddefined

- `gcc foo.o -o foo` generates the executable file

  which fills the question mark part in `foo.s` and using library files

  the CPU can only execute structures in the RAM but not in the file (secondary-storage)

p.s. you can directly compiling and linking a C program like:

```
$ gcc -c x.c
$ gcc -o foo x.o
```

Combining several of the detailed steps into fewer commands without explicitly generating intermediate files like preprocessed code or assembly code is efficient for everyday development and build processes.

NOTE: Machine code and assembly code are different representations of the same thing. Machine code is raw binary while assembly code is a human-readable form of that binary, as if it goes through the binary and assigns each opcode sequence a meaningful name. You can view the assembly code of an object file by **disassembling** it:

```
objdump -d binary_name
objdump -D binary_name
```

Linking Process (before execution)

Separating the compilation from the linking:

```
# Compile without linking using -c flag
$ gcc main.c def.h -c -o main.o
$ gcc def.c def.h -c -o def.o
# Link the two objects into the target executable with -o flag
$ gcc def.o main.o -o main
```

p.s. `-o` flag by itself doesn't imply the creation of an executable file; it simply specifies the output file name.

While you could do this on one line like:

```
gcc main.c def.c -o main
```

The former has the benefit that when defined in a `Makefile`, only the files that are updated are recompiled. This makes the process more efficient as we are not going through the entire compilation sequence for every file on every change.

Namespace Visibility

C has a very primitive version of namespace visibility compared to C++, and it does so with the keywords `static` and `extern`.

In C, **static allocation** is completely different from the `static` keyword. `static` defines a variable with **static lifetime**. This means the variable will have **internal linkage** (the identifier is visible only in the source file it is in).

Variables defined without `static` like `int x;` have **external linkage**, meaning another file is allowed to declare it. It would do so with the `extern` keyword:

```
// s.c
int x;
static int a[1000];
```

```
// t.c
extern int x;
```

The two `x`'s are one and the same. On the other hand, there is no way for `t.c` to access the identifier the array named `a` (by *identifier*; if the array is passed by reference to a function, the memory can still be accessed).

> **TL;DR:** `static` is like "private". If you want your code to be *modular*, you can use `static` to define symbols as part of a private API that other parts of the code need not worry about. If you prefer the approach where all parts of the program know about every other part, you can avoid using `static` altogether.

IMPORTANT: Every file that `#include`s a file gets their own private copy of every `static` symbol. This is usually the cause of "defined but never used" warnings.

C gets pretty complex when it comes to linking. Suppose you try to define a variable with the same name in yet another source file:

```
// v.c
int x;
```

Depending on the flags used, you would either get:

- An error.
- A situation where the `x`'s refer to the same variable.

Memory Allocation

There are 3 main allocation strategies:

Strategy	Memory Location	Lifetime	Notes
Static	Data	Before the program starts and will always exist until the program terminates.	Completely unrelated to the `static` keyword.
Auto	Stack	As part of its enclosing function's **call frame**.	LIFO policy makes it cheap and simple to push and pop values as needed. Completely unrelated to the `auto` keyword in C++.
Dynamic	Heap	Requested at runtime instead of at compile-time and must be manually freed.	Usually using a memory manager implementation like `malloc()` and `free()`.

Static Allocation

```
// Statically allocate an integer
int x;

// Statically allocate an array of 1000 integers
int a[1000];

// Statically allocate an array of 1000 integers that also
// has static VISIBILITY (i.e. private to this file)
static int b[1000];
```

➖ There's a performance/lifetime argument against static allocation. Typically, a program runs faster if you use stack allocation instead of static allocation (where a part of memory is always allocated for something for the entire duration of the program).

Stack Allocation

➕ Very cheap. The behavior of the stack allocation can be implemented at the *machine level*:

```
int f(int n) {
  char buf[512]; // allocate 512B on the stack
  buf[0] = n;
}
```

```
subq $512,%rsp
; ...
addq $512,%rsp
```

Heap Allocation

In C, you typically use the `malloc` library function defined under that `<stdlib.h>` header to dynamically allocate data:

```
// p is a pointer to a block of contiguous memory
// that is the size of 5 ints
int *p = (int *)malloc(5 * sizeof(int));

/* Code that uses that memory */

// De-allocate that memory when finished
free(p);
```

p.s. `malloc` stands for "memory allocation" and is a standard library function in C. It dynamically allocates a block of memory on the heap and returns a pointer to the beginning of the block.

For example:

```
#include <stdlib.h>

int main() {
    int *arr = (int*)malloc(10 * sizeof(int)); // Allocate memory for an
array of 10 integers

    if (arr == NULL) {
```

```
        // Memory allocation failed
        return 1;
    }

    // Use the allocated memory
    for(int i = 0; i < 10; i++) {
        arr[i] = i;
    }

    // Free the allocated memory when done
    free(arr);

    return 0;
}
```

`10 * sizeof(int)` Calculates the total number of bytes needed to store 10 integers.

`(int*)` is a cast to an `int*` type. `malloc` returns a pointer of type `void*`, which is a generic pointer type in C. It explicitly converts the `void*` pointer to an `int*` pointer, indicating that this memory will be used to store integers. `

ASIDE: Void Pointers

The special type `void *` refers to a **generic pointer**. You are allowed to cast any pointer type to a void pointer. Void pointers give you a lot of freedom, at the cost of checking. For example, attempting to return something dereferenced by the pointer is an error because C does not know what type it is:

```
int bad_stuff(int *p) {
   void *q = (void *)p;
   return *q; // bad!
   return *((int *)q); // you gotta give it a type
}
```

You can think of a void pointer as the most free type of pointer. Its value is just the value of the memory address and nothing more. Attempting to use increment/decrement operators would modify it by a unit's worth of memory (typically 1 byte).

I guess this could be useful when you know something meaningful is at a memory address, but do not know the type associated with it. An example of this is the `malloc` function, which returns a pointer to a chunk of memory you allocated, but it doesn't care what type you treat that memory as, so it returns a generic `void *`.

Common Problems in C

Pointer Dangers

Of course, the fact that you must `free()` your memory after using it means that using heap allocation is potentially dangerous. It's much easier to get **memory leaks** and/or **segmentation faults** with improper use of pointers and memory management.

Once you have a pointer to something that doesn't exist anymore (**dangling pointer**), you cannot look at the pointer. That leads to **undefined behavior**. If you're lucky, the program will crash. If you're unlucky, the error passes silently and possibly messes with memory that isn't related to the operation.

Dangling pointers: The pattern looks something like:

```
p = malloc(5); // allocates 5 bytes of memory
free(p); // free
*p; // using freed memory!
```

Memory Leaks

The pattern looks something like:

```
p = malloc(5); // allocates 5 bytes of memory
*p; // use
// but never freed!
```

There's no call to `free(p);`, which means the allocated memory won't be returned to the system until the program terminates. This behavior results in a memory leak.

Memory leaks are notoriously hard to trace. To catch them more reliably than with plain eyes, use tools or compiler options.

> **Why Deal with `malloc` and Pointers?**
>
> Suppose we just *pre-allocate* all objects statically and hand them out as the program runs:
>
> ```
> obj_t table[10000];
> ```
>
> - Firstly, how do you know how many objects to allocate? What if you run out?
> - Secondly, what if you never need that much memory? Your code would be bloated(臃肿), consuming more resources than it needs to. Program startup would also be slower.
>
> Thus, we use the heap to *dynamically* allocate resources *as needed*.

Refactoring (重构)

Exactly, refactoring is about restructuring existing code to make it more efficient, readable, maintainable, or conform to best practices, without altering the external behavior or functionality of the software.

Benefits of refactoring:

- ➕ Readability and modularity
- ➕ Easier to debug
- ➕ Reduces compilation time (parallel compilation of multiple files)

System Calls

A way for programs to interact with the operating system.

The concept of an **operating system** was invented to facilitate interaction between programs and the hardware. Without it, it would be much easier for programs to maliciously attack hardware or cause I/O conflicts with read/write operations.

Programs can make **system calls** *to* the operating system, and the operating system will then interact with the hardware in a well-defined way and report back with any output.

Categories of System Calls

1. Process control `fork`
2. File management `open`/`close` a file, `read`/`write`
3. Device management
4. information maintenance
5. Communication
6. Protection

File Operations

1. `open()`

Library

Part of the **POSIX** standard, which is a family of standards specified by the IEEE for maintaining compatibility between operating systems.
`open()` is included in the `<fcntl.h>` header file on UNIX-like systems (Linux, macOS, etc.).

Usage

```
int open(const char *pathname, int flags, mode_t mode);
```

Flags determine the access mode

- `O_RDONLY` for read-only
- `O_WRONLY` for write-only
- `O_RDWR` for read/write
- `O_CREAT` to create a file if it doesn't exist

Mode sets the permissions of the file if it is created. This parameter is required if O_CREAT is used in flags.

Return Value

The return value is an integer, wchich represents **a file descriptor**

- `-1` stands for error
- `any non-negative integer` stands for success

p.s. File descriptors represent a low-level way to interact with files and are obtained using system calls like open().

2. `read()`

This is a low-level input function used to **read bytes from a file descriptor into a buffer**.
Also a part of the POSIX API and is typically used with file descriptors that represent files, pipes, or sockets.
`read()` is included in the `<unistd.h>` header file.

Usage

```
ssize_t read(int fd, void *buf, size_t count);
```

Parameters

- **fd** The file descriptor of the file from which to read.
- **buf** A pointer to the buffer where the read bytes will be stored.
- **count** The number of bytes to read.

Return Value

- On success
  returns `the number of bytes read` and placed in buf. (this number could be less than count if fewer bytes are actually available)
- On error
  returns `-1` and sets errno to indicate the error.
- EOF
  returns `0` indicates end-of-file (EOF).

> Example

```
int fd = open("example.txt", O_RDONLY);

if (fd == -1) {
  perror("open");
  return 1;
}

char buffer[128];
ssize_t bytes_read = read(fd, buffer, sizeof(buffer) - 1);

if (bytes_read == -1) {
  perror("read");
  close(fd);
  return 1;
```

```
    }

    buffer[bytes_read] = '\0'; // Null-terminate the string
    printf("Read %zd bytes: %s\n", bytes_read, buffer);
    close(fd);
```

3. `write()`

like `read()`, is a low-level system call used for writing data to a file descriptor. It is also a part of the POSIX API.
`write()` also declared in the `<unistd.h>` header file on Unix-like operating systems.

Usage

```
ssize_t write(int fd, const void *buf, size_t count);
```

Return Value

- On success
  returns `the number of bytes written`. (it might write fewer bytes than requested, but this is not an error scenario; it's typically observed with non-blocking I/O or when writing to a pipe or socket that is full)
- On error
  returns `-1` and sets errno to indicate the error.

Buffers and File Descriptors

`Buffers`

In C programming, a buffer is simply a block of memory where data can be stored temporarily.
When you're using `read()` or `write()`, you provide a buffer (i.e., an array or a pointer to some memory location) where data should be read into or written from.
Buffers themselves do not have any metadata like a "current position" or a "pointer to a file" associated with them; they are just memory locations.

`File Descriptors`

File descriptors are integers that uniquely identify an open file (or socket, pipe, etc.) within a process.
They are used by the operating system to keep track of all the open files and their states, including the current position in the file for reading or writing.