C Programming

Compilation Process

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.

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
```

Compiling/Linking with GCC

Separating the compilation from the linking:

```
# First 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
```

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.

The C Language: Missing C++ Features

- 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 has a single top-level namespace.
- Overloading. There is a feature called `_Generic` that attempts a way around this, but this is esoteric.

- Exception handling. C has a system with `<setjump.h>` but it has all sort of trouble.
- Memory allocation. There is no `new` operator in C. Instead, you use the *library functions* `malloc()` and `free()`; they aren't built into the language itself.
- No `cin`/`cout`. You use IO with the library functions under the `<stdio.h>` header.

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 fine is allowed to declare it. They 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 hairy 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);
```

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); // allocate
free(p); // free
*p; // using freed memory!
```

Memory Leaks

The pattern looks something like:

```
p = malloc(5); // allocate
*p; // use
// but never freed!
```

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, consuming more resources than it needs to. Program startup would also be slower.

Thus, we use the heap to *dynamically* allocate resources *as needed*.

HW5: Refactoring

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

The `write()` System Call

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

- `fd` stands for file descriptor, which could be stdout or stderr.
- `*buf` stands for buffer. This contains any data in it.
- `count` is the number of bytes to be written to a file descriptor from the buffer.