Machine code is binary code that can be executed directly by the CPU.

Object code is a portion of machine code not yet linked into a complete program. It's a machine code for one particular library or module that will make up the completed product. It may also contain placeholders or offsets not found in the machine code of a completed program. A linker will use these placeholders and offsets to connect everything together.

Assembly code is plain text file with a mostly direct one-to-one relationship to machine instructions. This is accomplished using mnemonics for the actual instructions, registers, or other resources. Unlike machine code, the CPU does not understand assembly code. we convert assembly code with assembler or compiler.

C Programming language to executable

Source code file (hello.c)

↓ C Preprocessor

Preprocessed code file (hello.i)

↓ C Compiler

Assembly code file (hello.s)

↓ Assembler

Object code file (hello.o)

Relocation object code

Other object files or modules

Library files

↓ Linker / link editor

Executable code (hello or hello.exe)

⇣

Stored in secondary storage (e.g hard disk) as an executable image

⇣ ——————————→ when running the program

Process address space (Primary memory, e.g RAM)

(Runtime objects / modules / libraries (deferred linking))

Structure declarations

There are 2 formats:

- typedef struct {
    int x;
    int y;
  } point;

  => usage: point (p) → variable

- struct point_1 {
    int x;
    int y;
  };

  => usage: struct point_1 (p) → variable

  assignment:
  struct point p = {.y=1, .x=2};

In general, a pointer, or a struct that has elements that can reasonably be directly accessed should never be a typedef.

Name space

- Label: All identifiers declared as labels (the `goto` type).

- Tag: All identifiers declared as names of structures, unions, and enumerations.

- Member: All identifiers declared as members of any one structure or union. Every structure and union introduces its own name space of this kind.

- All other identifiers (a.k.a ordinary identifiers): These identifiers include function names, object names, typedef names, enumeration constants.

Declaration or Initialization of array

This works:

```
char input[4] = "abc"; // Initialization
```

This does not work:

```
Char my-array[4];
my_array = "abC";
```

Array is not directly assignable in C.
The name `my_array` actually resolve to the address of its first element (&my_array[0]), which is not an l-value, and as such can not be the target of assignment.

lvalues and rvalues

lvalues refers to an object that persists beyond a single expression. lvalues is something that exist in C source code.

rvalues is a temporary value that does not persist beyond the expression that uses it. rvalues is something that exist during program execution. Think of rvalues as temporary object.

C preprocessor

Any line that start with #, like #include <stdio.h>, is a preprocessor command.

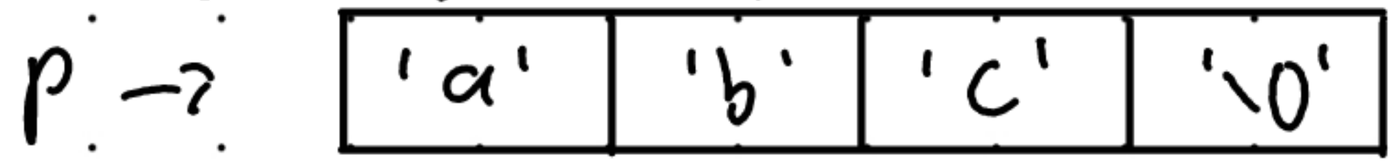
String

String in C is an array of characters.
For example:

```
const char *p = "abC";
```

The variable p points to some memory. The expression *p means dereference the pointer to give us a single character, namely the one that p is actually pointing to, which is p[0].

We can think of variable p like this:



The variable p does not really point to a "abC", p only points to some single location in memory, namely the first character in the string "abC".

It is the function using p that treat that memory as a sequence of characters.

Why *p is the same as p[0]?

For any pointer or array p and valid index i, the expressions p[i] is equal to *(p + i). To get the first character, we have index 0 and then we have p[0] which is equal to *(p+0). Adding zero (0) to anything is a no-op, so *(p+0) is the same as *(p) which is the same as *p. Therefore p[0] is equal to *p.

Char type

Char type is to store characters and strings, however, C stores char type as integer values instead of literal character values. And then, the computer has to map those integer values with a corresponding character using a numerical code.

The most common numerical code is ASCII (American Standard Code for Information Interchange).

For example, integer values 65 represents a character 'A' (upper case) in ASCII.

Difference Char *str1 and Char str2[]

Let's say we have:

```
char *str1 = "Hello!";
char str2[] = "Hello!";
```

The first statement (char *str1) telling the compiler that we want a pointer to a string literal. The String literal is saved in (read-only memory) and cannot be changed. str1 is just a pointer to the first element of the string.

The second statement (char str2[]) is a character array and generally allocated on (the stack), not in read-only memory. When we use the second statement, we are telling the compiler we want a character array that can exactly fit the string "Hello!" plus zero terminator (\0).

Add Array Size in Char Str[]

When we add size into an array of char, we need to provide the zero-terminator like this:

```
Char str[3] = "ab\0";
```

If we don't do that, the string literal will overflow and contains another strings literal. For example:

```
char a1[3] = "abc";
Char b1[] = "def";
Char *c1 = "ghi";
```

If we print a1, the result would be "abcdef" because the zero terminator is found on b1, not a1. So printing a1 would have more character than it should be.

The Reason Can Change Value const char *str

When we do this:

```
const char *str = "string";
str = "changed";
```

we are changing the pointer or virtual memory address of str, which is not constant. But the characters that the pointer point to is constant.

Constant in String Literals

There are 3 ways we can declare constant with String literals:

- `Char *const str1;` // str1 cannot be modified and the characters pointed to theoretically can but is not valid because string literals is saved in read-only memory.
- `const char *str2;` // str2 can be modified but the characters pointed to cannot.
- `const char *const str3;` // Neither str3 nor the characters pointed to can be modified.

Backward Reading Declaration

This is a tricle to read the declaration backeward, for example;

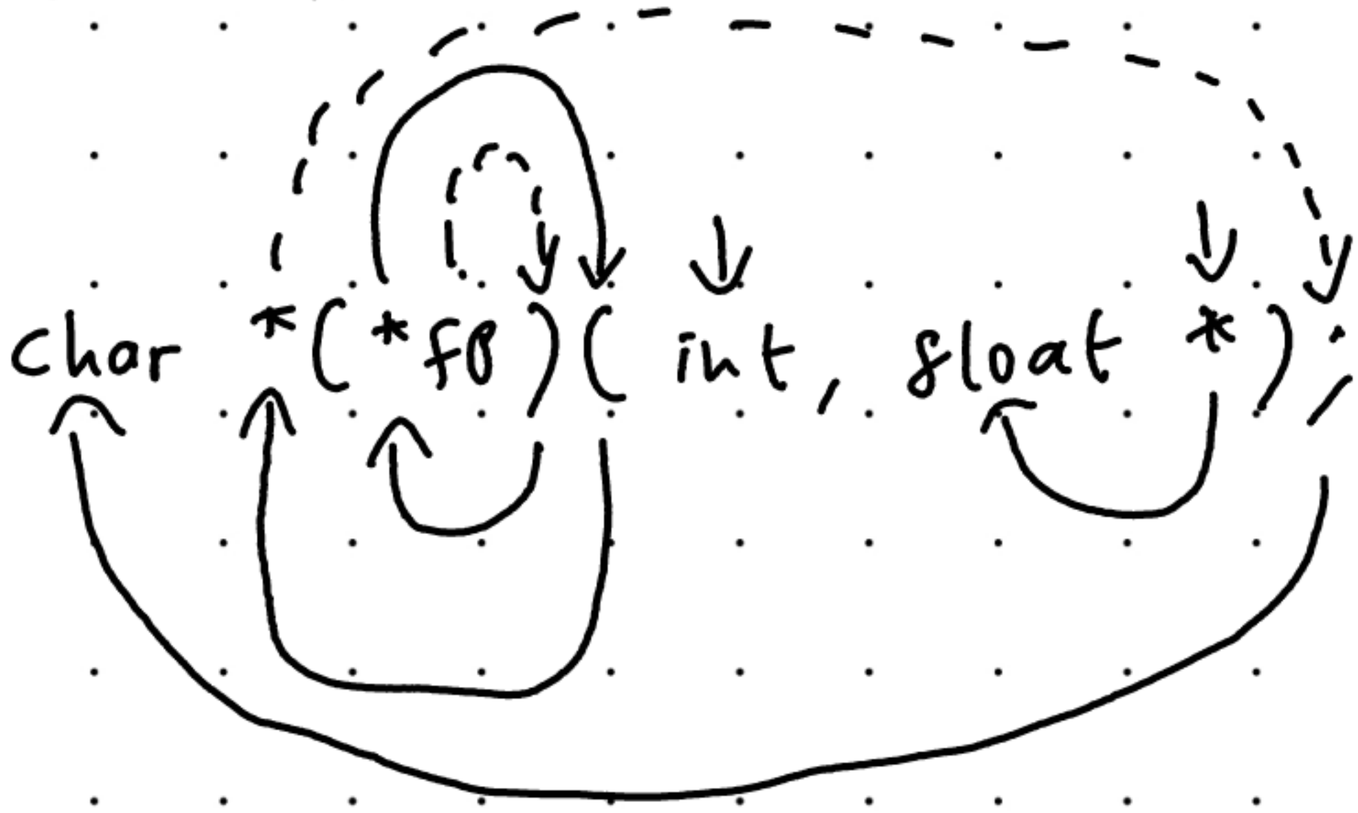
```
Const char *s; // read as "s is a pointer to a char
                  that is constant.

Char *Const e; // read as "e is a constant pointer
                  to a char.

Const Char *const x; // read as "x is a constant
                        pointer to a char that is
                        constant.
```

Clockwise / Spiral Rule

1. Starting with the unknown element, move in a clockwise / spiral direction, when encountering the following elements, replace them with the corresponding english statements:
   - [X] or [] => Array X size of... or Array undefined size of...
   - (type1, type2) => Function passing type1 and type2 returning...
   - * => pointer(s) to...
2. Keep doing the first step in a clockwise / spiral direction until all tokens have been covered.
3. Always resolve anything in parenthesis first!

Example of Clockwise / Spiral Rule

what is fp ?



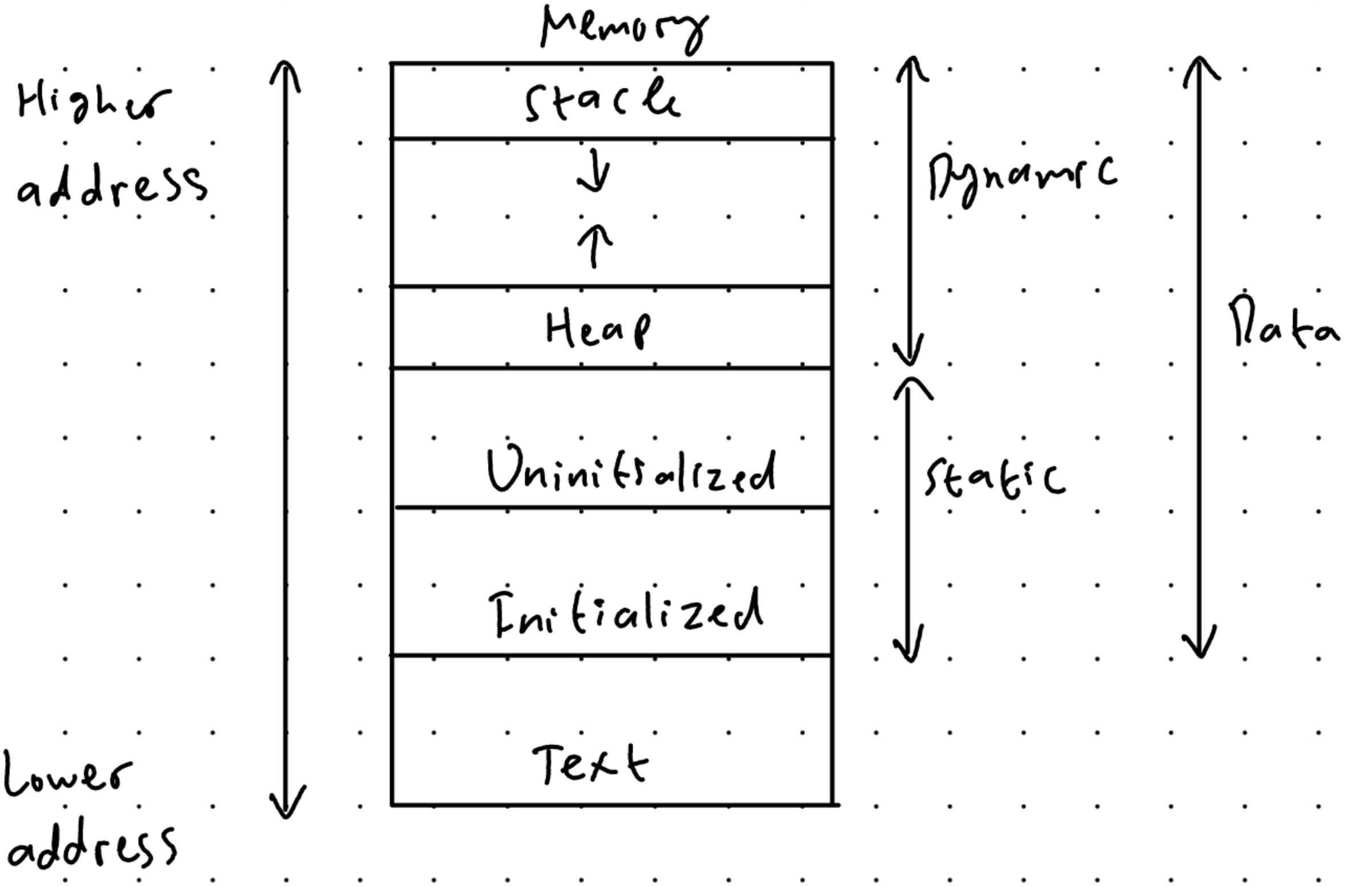
note:

--- : skipped

—— : not skipped

fp is a pointer to function passing int and pointer float returning a pointer to char.

Memory Segments



Memory Segment Explanation

- Text/code segment Contains intructions for executing the program. This segment is read-only, contains executable code, and fixed size.

- Initialized/data segment hold global and static variable that have values in the code. This segment is also read directly from executable, fixed size, and read/write permission.

- Uninitialized/BSS (Block started by Symbol) segment hold global and static variable that does not have values or initialized to zero. The assembly directive BLK (block) initialized a block of memory to zero. this segment is fixed size.

- Heap segment is dynamic segment that we can grow and shrink the size. This segment is useful if we don't know the size before compiling the program.

Memory Segment Explanation (Continue)

- Stack segment hold function calls, local variables in a function, and function parameters. This Segment operates in "last in, first out".

.bss Segment in C

The .bss segment is a part of the program's memory that is used to store global and static variable that are uninitialized or initialized to zero. The name ".bss" derived from the older assembler keyword "Block Started by Symbol".

Why Do We Need .bss Segment?

The primary reason is for memory optimization. When a program is stored on disk, the .data segment contains the values of initialized global or static variables, while the .bss segment only records how much memory to allocate for uninitialized variables and does not store any actual data.

If all variables, initialzed or not, were placed in .data segment, the executable file will be significantly larger because .data segment would contain a lot of unnecessary zeros for uninitalized variables.

At run-time, the operating system allocates the necessary memory for the .bss segment and initialize it to zero.

Segmentation Fault

On older systems or modern embedded system without virtual memory, all memory addresses were put directly onto the memory bus. If we have a pointer to address 0xdeadbeef, then that's exactly the address in physical memory that will be accessed if we dereference that pointer.

On a modern system with virtual memory, each process gets its own address space and the addresses in this virtual address space are mapped to physical addresses by CPU. If we access pointer to address 0xdeadbeef from a system with virtual memory, the CPU will consult a table of memory pages, find which page this address belongs to, and replace it with the address of physical memory from the table.

what happens if we try to access a virtual memory address which there is no mapping to physical memory or not have access (write to read-only memory)? The CPU will inform the OS, and then the OS will kill the process.

That is segmentation fault.

Memory Paging

The major Problem with Programs using the physical address of the memory directly is that we need to manually manage memory of our program so that our program's memory (did not collide) with another Program's memory.

With Virtual memory, each Process has it's own dedicated virtual address so we don't have to worry about Processes colliding with each other.

Even thought each Process has its dedicated virtual address, the Process is still being stored inside of Physical memory.

The record of where each virtual address in Physical address is usually stored by operating System in a Per-Process data structure known as a (Page table).

Translation Lookaside Buffer (TLB)

Translation Lookaside Buffer (TLB) is a memory cache that stores the most frequently accessed Page table entry.

The system first check the TLB for the translation memory address to physical address entry, if the desired entry exist in the TLB, the translation can be done quickly.

How Does TLB Decide which Entry to Store?

The TLB uses a combination of Spatial locality and temporal locality to predict what data is likely to be used.

Spatial locality refers to the tendency of a program to access data locations that are close to each other in a short period of time.

Temporal locality refers to the tendency of a program to access the same memory address repeatedly within a short time frame.

Declaration, Assignment, Initialization Varsable

- Declaration variable :

```
int X;
```

- Declaration and assigment Varcable :

```
int x;
x = 69;
```

- Initialization :

```
int x = 69;
```

Array and Pointer is not the Same

Array and pointer (arithmetic) is defined such that a pointer can be conveniently used to access an array or simulate an array.

Like Wayne Throop said, "pointer arithmetic and array indexing that are equivalent in C, pointers and array are different".

The cornerstone of the equivalence is that:

A reference to an object of type array-of-T which appear in (an expression) converted into a pointer to its first element; the type of the resultant pointer is pointer-of-T (T in here is representation of any type).

So, whenever and array appear in an expression, the compiler implicitly generates a pointer to the array's first element, (as if) the programmer had written &a[0].

The exception are when the array is
- The operand of `sizeof`,
- Using & (unary) operator,
- A string literal (initializer) for a character array,
- The _Alignof operator.

Array as Function parameter

When we are passing array as function parameters, like this:

```
int something(int array[])
{
    printf("sizeof array: %d\n", sizeof(array));
}
```

we are passing array as the pointer. Therefore `sizeof` will return the size of pointer, not the size of array.

With that in mind, if we need the size of array, we should pass the array size as another function parameter, like this:

```
int something(int array[], int size)
{
    printf("sizeof array: %d\n", size);
}
```

Two-dimensional Array to Function parameter

The rule which array converted into pointers is not applied recursively. An array of arrays in C converted into a pointer to an array, not a pointer to a pointer. Pointer to arrays can be confusing and must be treated carefully.

For example:

```
int array[2][3];

void something(int a[][3]) // or int *a[7]
{
    ...
}
```

In the example above, we can omit the number of rows because the function does not allocate space for the array. But we still need to provide the columns.

Macro Definition

We can define a macro using #define directive like this:

```
#define NICE 69
```

And for every word NICE (case-sensitive and exact word, no additional character) will be replaced with value 69 in the preprocessing process.

To check the result of preprocessing, we can use command `CPP` (more info on `man cpp`).

Difference Between Macro and Global Variable

One of the difference between global variable and macro is that we can easily change global variable value like changing the local variable, but with macro, we need to undefine the macro using #undef directive and redefine the macro again.

This is one of the reason a lot of people using macro instead of global variable, because keeping track the value of global variable can be hard as we can easily change the value.

Difference Between Structure and Union

A structure stores each member in separate memory locations, whereas a union stores all its members in the same memory location.

The size of the union is the size of the largest member.

When we write a value to a union member, the value of other union member is overwritten.

Signed and Unsigned Integer

A signed integer is an integer that (can be negatif).

An Unsigned integer is an integer that (can not be negatif),

In C, (default) integer type is a signed integer.

Difference Between itt and ++i

itt means that we (use) the value of variable i first and increment value i (later on).

++i means that we (increment) the value first and (use the result of increment) directly.

Keep in mind that either itt or ++i is prone to (integer overflow).

For Socket Programming, try check out:

man 3 getaddrinfo

Include Another C File

We can include other C files with this directive:

```
#include "file.C"
```

The problem with this approach is that when we include another C file, we are including everything, even the static function that should be local to another C file. So we lose control access to data and functions, and also we can end up with the translation unit too large for the compiler to handle.

Most of the time separate compilation and linking is a better choice. We can do that with gcc like this:

```
gcc -o main file.C main.c
```

Implement Library

The convention in C when implementing a library is to divide it into header file (file.h) and source file (file.c).

We typically include only the bare minimum interface in the header file, like struct definitions, function prototype declarations, macro definitions, external declarations, etc. Then the implementation of the function in the corresponding source file.

When making the library available to the public, we would compile the source file into binary (like file.so or file.dll, depending on operating system) and deliver the header file with the binary file.

As a user of the library, we need to do:
- Include the library header file (#include ...).
- When compiling our code, tell the linker where to find the implementation binary for the header (gcc -l library file.c).

Call Stack

An area of program memory that is used to manage function calls.

Stack Frame

The portion of memory in the call stack associated with a **single running function**.

Function that has many parameters and local variables will have larger stack frames than those with few parameters and local variables.

The compiler is able to determine during compilation the stack frame size for a function.

There are usually as many stack frames on the call stack as there are functions that have yet to return.

Virtual Stack

On linux 64-bit, the virtual stack size is 8388608 bytes or 8 MB. Our thread only use the amount it need.

Here's a sniffet of virtual stack allocation:

- The operating system allocates 8 MB of virtual memory for our "call stack" by setting up the Memory Management Unit (MMU) page tables for our thread. This requires very little RAM to hold the page table entries only.

- When our thread runs and tries to access a virtual memory address on the stack that does not have a physical memory assigned to it yet, a hardware exception called a "page fault" is triggered by the MMU.

- The CPU responds to the page fault exception by calling the page fault exception handler function inside the kernel. The kernel allocates a page of physical RAM to that virtual memory page and returns back to the user space thread.

Memory Address Space

Memory address space define a range of possible memory address whether it is real memory or virtual memory.

Let's say we have 16 GB of real memory, the memory address space is between 0 bytes until 16 GB. And when we need to use it, we grab the unused real memory address.

For virtual memory, it depends on the system being used and not related to the real memory size.
Here's the theoretical limit based on the system:

- 16 bit = 65.536 bytes (64 kB)
- 32 bit = 4.294.967.296 bytes (4 GB)
- 64 bit = 18.446.744.073.709.551.616 bytes (16 Exabytes)

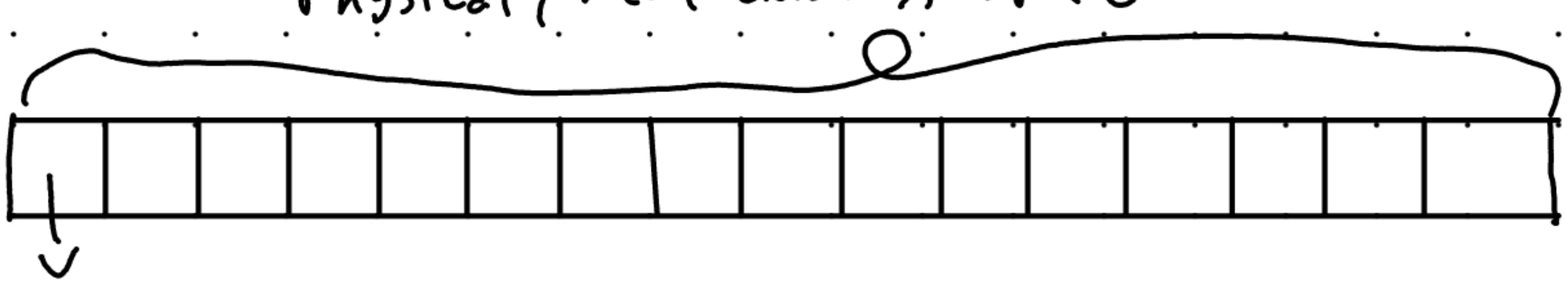
Just because a machine use 64-bit architecture, does not mean we can use all of those 64 bits address space. Current X86-64 CPUs only use 48-bit address space, which theoritically 256 Terrabytes.

An operating system may have internal limitations to how much physical or real memory that can be efficiently supported. In fact, this is actually to prevent needing overly large data structure to keep track of usage for memory that is not there.

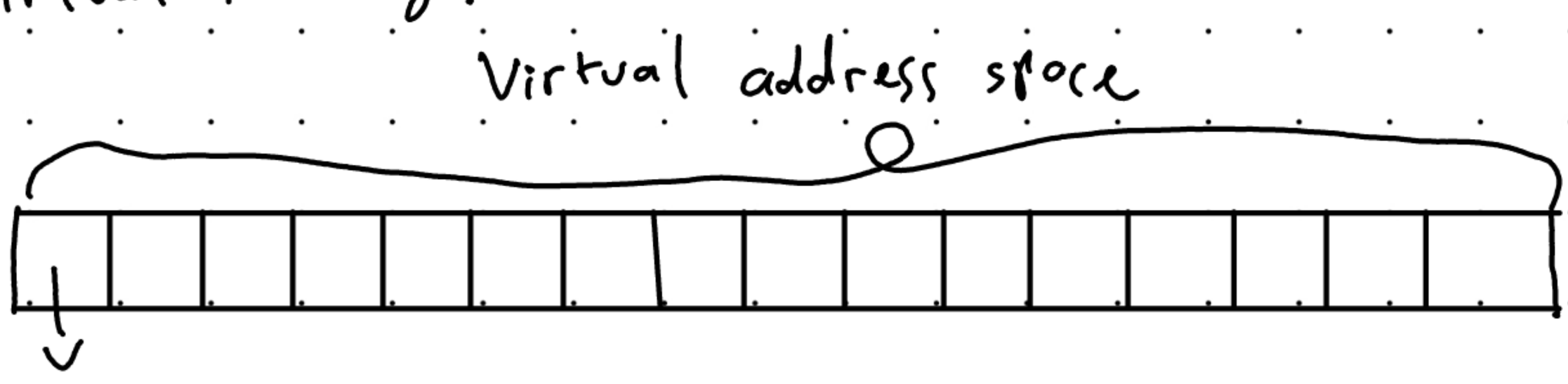
Once we spawn a process, we are creating a virtual address space that can be used by the process. Virtual address space created depending on the system.

Representation of Memory Page

Physical / real memory:



Virtual memory:



Process Memory Concept

There are a lot of different ways system organize memory, but in a typical one is

each process has one linear virtual address space.

Memory need not be contiguous.

The virtual memory is divided into pages (typically 4 kilobytes for each page). Backing each page of virtual memory is a page of real memory (called frame) or some secondary space, usually disk space. The disk space might be swap space or just some ordinary disk file.

A page of all zeroes sometimes has nothing at all backing it, there's just a flag saying it is all zeroes.

The same frame of real memory or backing store can connect to multiple virtual pages belonging to multiple processes.

The smallest amount of memory that the operating system or hardware deals with is a page, so each 4 kB within a single page is dealt with as one.

In order for a program to access any part of a virtual page, the page must, at that moment, be connected to a frame page. But because there is usually a lot more virtual memory than real memory, the pages must move back and forth between real memory and backing store regularly, coming into real memory when a process needs to access them and then retreating to backing store when not needed anymore. This movement is called paging.

When a program attempts to access a virtual page which is, at the moment, not connected to frame page is known as page fault.

When a page fault occurs, the kernel suspends the process and places the virtual page into real frame (this is called paging in or faulting in), and then resumes the process. From the process point of view, the virtual page was in frame page all along.

Within each virtual address space, a process has to keep track of what is at which address, and that process is called memory allocation. Memory allocation within a process is mainly just a matter of making sure that the same byte of memory is not used to store different things.

Processes allocate memory in 2 major ways:

- Exec
- Programmatically

Exec is the operation of creating a virtual address space for a process, loading its basic program into it, and executing the program. The operation takes a program file (an executable), it allocate space to load all the data in the executable, and then load the data and transfer controll to the executable. That data is most notably the instructions of the program (the text), literals, constants, and some variables like C static storage class.

Memory Segments

A process' virtual address space is divided into segments.
A segment is a contiguous range of virtual addresses.

Three important segments are:

- Text segment: Contains a program's instructions, literals, and static constants.
  It's allocated by exec and stay the same size for the life of the virtual address space.

- Data segment: The working storage for program.
  It can be pre-allocated and pre-loaded by exec and the process can extend or shrink it by calling functions like malloc(). Its lower end is fixed (?).

- Stack segment: Contains a program stack.
  It grows as the stack grows, but does not shrink when the stack shrink (?).

Stack Frame

The call stack is divided up into contiguous pieces called stack frames; each frame is the data associated with one function call. The frame contains the arguments given to function, the function's local variables, and the address at which the function is executing.

When our program started, the stack has only one frame, that of the function main. This is called the initial frame or the outermost frame.

Each time a function is called, a new frame is made. Each time a function returns, the frame for that function invocation is eliminated.

If a function is recursive, there can be many frames for the same function.

The frame for the function in which execution is actually occurring is called the innermost frame. The innermost frame is the most recently created of all the stack frames that still exist.

Inside our program, stack frames are identified by their addresses. A stack frame consists of many bytes, each of which has its own address; each computer has a convention for choosing one byte whose address serves as the address of the frame. Usually this address is kept in register called the frame pointer register while execution is going on in that frame.

GDB assigns number to all existing stack frames, starting with zero for the innermost frame, and then number one for the frame that called the innermost frame, and so on. These number do not really exist in our program; they are assigned by GDB to give us a way to check the stack frame in GDB.

Null address

If we are assigning a pointer with NULL, like this:

```
int *i = NULL;  // or int *i = 0;
```

In some implementation, it will point to memory address 0x0 which means that the pointer does not point to a valid memory address.

Function Pointer

Function Pointer is a type of pointer that stores the memory address of a function.