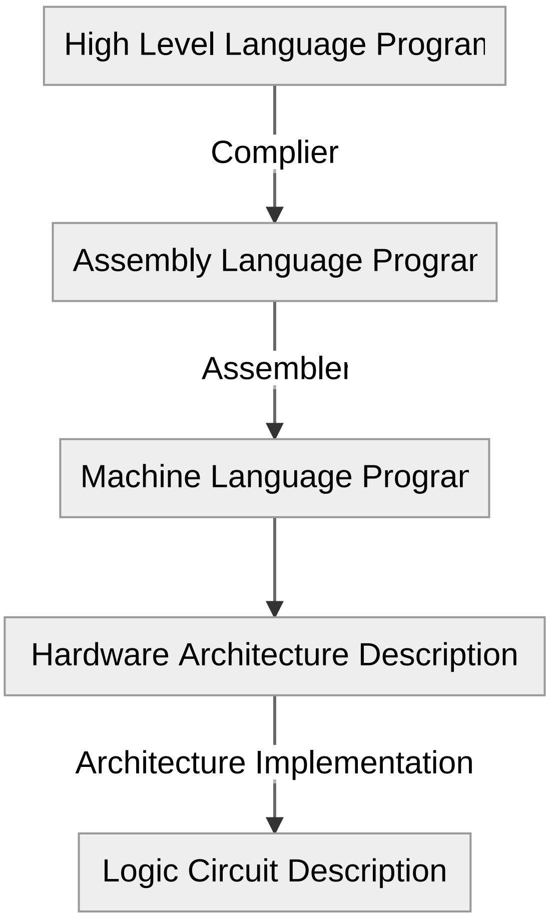
Introduction to C

History

- 1946: ENIAC (University of Pennsylvania): the first electronic general purpose computer
  - Could multiply in 10 digit numbers at 2.8ms!
  - Needed 2-3 days to setup a new program, which would be done via patch cords and switches
- 1949: EDSAC (Cambridge University): the first general *stored-program* computer
  - Programs held as numbers (32-bit binary two's complement) in memory

Computer Organization

Abstraction

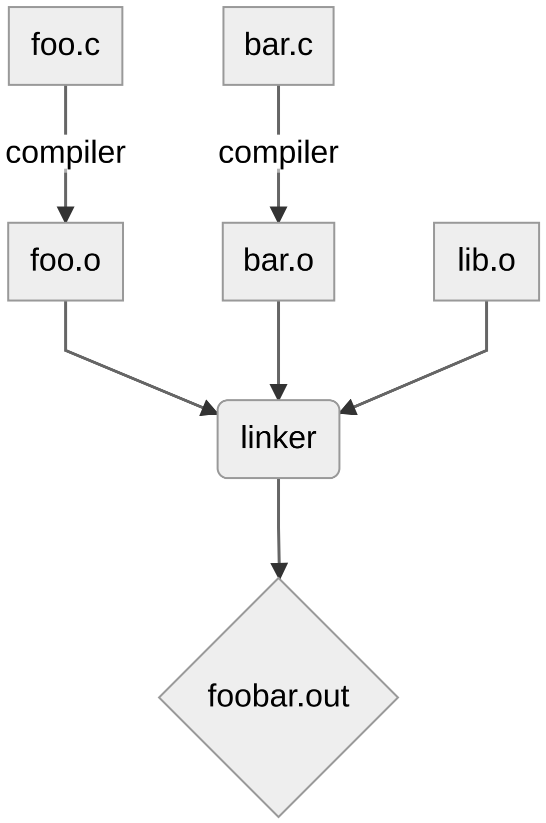

High Level Language Progran
Complier
Assembly Language Prograr
Assembler
Machine Language Prograr
Hardware Architecture Description
Architecture Implementation
Logic Circuit Description

C lets us write programs that can exploit underlying features of the architecture.

Compile vs Interpret

C **compilers** map C programs directly into *architecture-specific* machine code. In C, compilation takes two steps:

1. Compiling `.c` files to `.o` files
2. Linking `.o` files to executables. a. **Assembling** is also automatically done, but it is hidden.



> Compilation provides reasonable complication time performance and excellent run-time performance (generally significantly faster than Scheme or Java). However, compiled files, including the executable, are architecture-specific and depend on the processor type and the operating system. Thus, executables must be rebuilt on each new system. Moreover, while compilation can be done in parallel, the linker must be performed sequentially.

C Pre-Processor (CPP)

The CPP allows C source files to first pass through a macro processor before the complier sees the code, replacing comments with a single space. CPP commands (beginning in `#`) are also replaced appropriatly.

```
#include "file.h" /* inserts file.h into output */
#include <stdio.h> /* looks for file in standard location */
#define PI (3.14159) /* defines a constant */
#if{text}/#endif /* conditionally include text */
```

CPP works via string replacement, and if not used properly, the C Pre-Processor can create interesting errors.

C Language

C vs Java

Category	C	Java
Type of Language	Function Oriented	Object Oriented
Programming Unit	Function	Class (abstract data type)
Compliation	`gcc` creates machine language code	`javac` creates Java virtual machine bytecode
Execution	`a.out` loads and executes program	`java Hello` interprets bytecode
Preprocessor?	yes	no

The operators between C and Java are nearly identical.

C Syntax

True vs False

False: `0` (int), `NULL` (pointer), `FALSE` (boolean) True: everything else

Typed Variabels

Type	Description	Example
`int`	integers (including negative)	`0, 78, -217, 0x7337`
`unsigned int`	unsigned integers	`0, 6`
`float`	floating point decimal	`0.0, 3.14159`
`double`	higher precision floating point	`0.0, 3.14159`
`char`	single character	`c, \n`
`long`	longer `int`	123976128

In C, functions are typed by their return values. You must declare the type of data you plan to return from a function, which can be any C variable type. A `void` return type signals `NULL`.

Structs

```
typedef struct {
    int length_in_seconds;
    int year_recorded;
} SONG;

SONG song1;
song1.length_in_seconds = 213; /* dot notation */
song1.year_recorded = 1994;
```

Control Flow

Control flow in C is remarkably similar to Java.

- `if-else`: `if (expression) {statement}`
- `while`: `while (expression) {statement}`
- `for`: `for (initialize; check; update) {statement}`
- `switch`:

  ```
  switch (expression) {
    case case1: statements
    case case2: statements
    default: statements
  }
  ```

Bugs and Pointers

C variable declaration is similar to java with some minor but important distinctions:

1. All variable declarations must appear before they are used.
2. All declarations must be at the beginning of the block.
3. A variable may be initialized in its declaration. a. If it is not, **it holds garbage** (the contents are undefined).

Heisenbugs are bugs that have *undefined behavior*: they behave unpredictably.

Address vs. Value

Memory in much like a single huge array: each cell has an associated address and stores some value. **Pointers** are an address that refer to a particular memory location; in other words, they *point* to a memory address.
Pointer syntax:

```
int *p; /* variable p is address of an int */
p = &y; /* assign address of y to p */
z = *p; /* assign value at address in p to z, or dereference p */
```

Pointer Dangers

Consider the following code:

```
void f() {
    int *ptr;
    *ptr = 5;
}
```

After execution, we have initialized a pointer that points to address `5`, which could do anything. However, despite the countless bugs they cause, pointers allow us to pass large structs or arrays in a space (and time) efficient manner.

Using Pointers

Pointers can point to *any* data type and normally point to only one type. The execption are **NULL pointers** which is a pointer of `0`s. Writing to or reading from a null pointer crashes a program.
A C pointer is just an abstracted memory address, and the type declaration tells a complier how many bytes to fetch on each access. Thus, there are multiple solutions to pointing to different sized objects.

Arrays

An **array** is just a block of memory. The code `int ar[2];` declares an integer array of 2 elements.
Arrays are (almost) identical to pointers; `char *string` and `char string[]` are nearly identical declarations with only subtle differences. Subsequently, *an array variable can be thought of as a pointer to the first element* of the array. Consequently, the following identities hold:

```
ar; /* an array variable looks like a pointer */
ar[0] == *ar;
ar[2] == *(ar+2);
```

Array Symantics

An array of size **n** can access from **0** to **n-1** via a counter. It is good practice to create a variable holding the arrays size to pinpoint a *single source of truth*.
An array in C does not know its own length and bounds are not checked. Content just outside of an array can be accidently accessed and manipulated which can lead to **segmentation faults** or **bus errors**.

Memory

Dynamic Memory Allocation

C has the `sizeof()` operator which returns the size of a type or variabel in **bytes**. `sizeof()` even knows the size of arrays.
The allocate room for something new to point to, `malloc()` can be used in tandem with `sizeof()`.

```
ptr = (int *) mallox(sizeof(int)); /* now pts points to somewhere in memory of size (sizeof(int)) bytes */
```

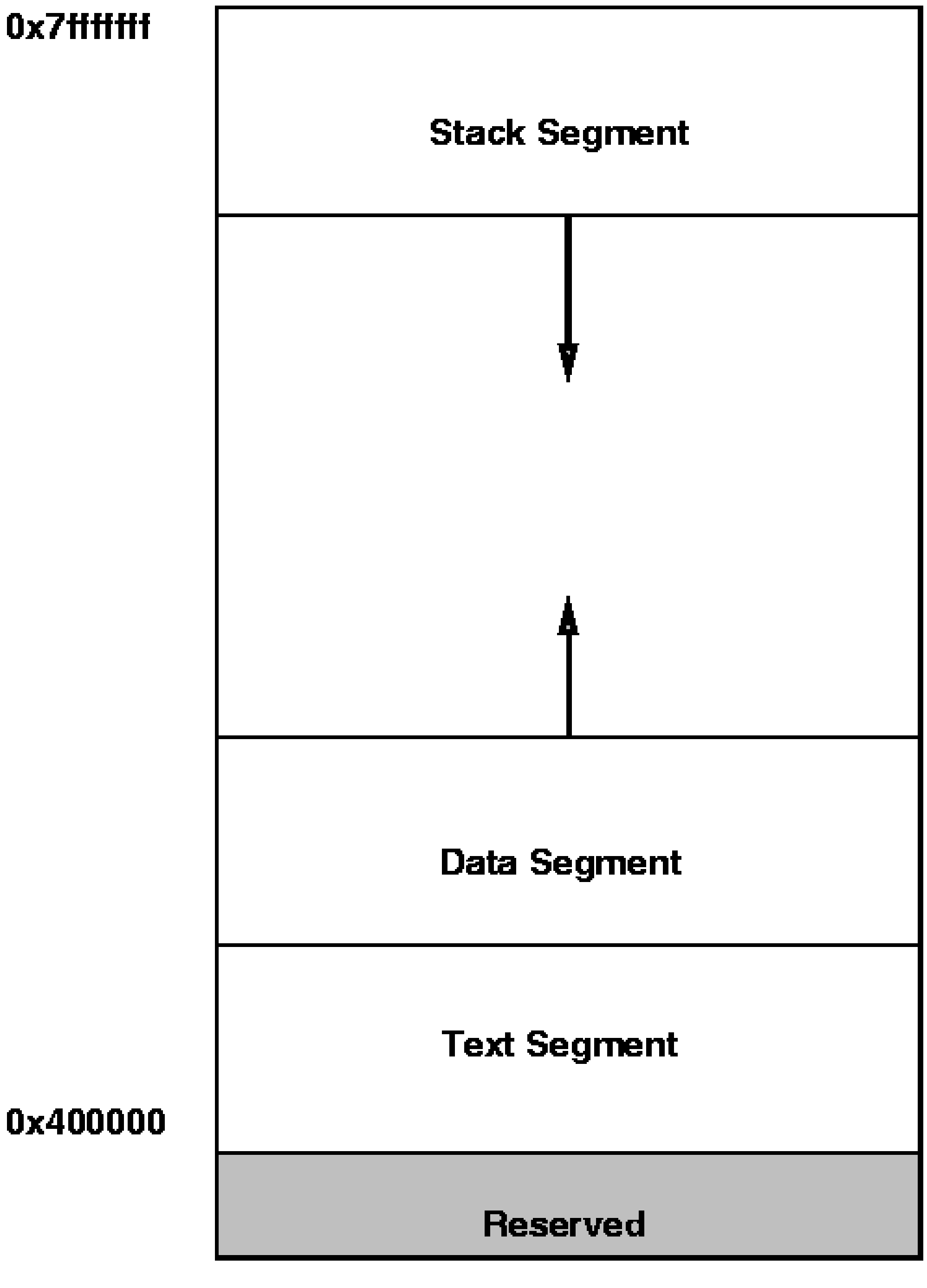
Once `malloc()` is called, **the memory location contains garbage**, so we must set a value for it. After we are done with using the space, we can free it using `free()`. `free()`ing the same piece of memory twice or calling `free()` on something we did not recieve from `malloc()` can cause crashes or strange behavior as the runtime does not check for these mistakes.

Heap Management

`realloc(ptr, size)` is used to resize previously allocated blocks. If `ptr` is `NULL` `realloc()` has the same behavior as `malloc()`, and if `size` is `0`, the `realloc()` has the same behavior as `free()`.

Memory Locations

The **stack** contains local variables, parameters, and return addresses. It growns downward.
- Stack frames are contiguous blocks of memory following a LIFO structure, where the **stack pointer&** points to the top stack frame. - When a procedure ends, the stack frame is deleted and the memory is freed. The **static storage** stores global variables. It is basically permanent for the entire run of the program.
The **heap** stores data synamically until it is deallocated by the programmer.
- The heap is a large pool of memory *not contiguously allocated*. Back-to-back requests for heap memory can result in distant blocks. The **code** is loaded when the program starts and does not change.



Memory Management

Code and static storage never grow or shrink, and similarly stack space management is easy: frames are created and destroyed in LIFO order. Managing the heap is tricky: memory can be allocated and deallocated at any time. We want `malloc()` and `free()` to run quickly with minimal overhead while simultanouesly avoiding **fragmentation** (seperation of free memory into several small chunks).

K&R Implementation

Each block of memory is **circular linked-list** containing the size of the block and a pointer to the next block. `malloc()` searches the free list for a block while `free()` checks if the adjacent blocks are also free. If so, adjacent free blocks are merged into a single, large block.
There are many implementations to choose a block in `malloc()`:

1. Best-fit chooses the smalled block that is big enough for the request. This requires traversal of the entire linked-list.
2. First-fit chooses the first block that is large enough. This is incredibly fast but greatly increases fragmentation.
3. Next-fit is basically first-fit, with the added feature of remembering (and resuming search from) the last accessed block.