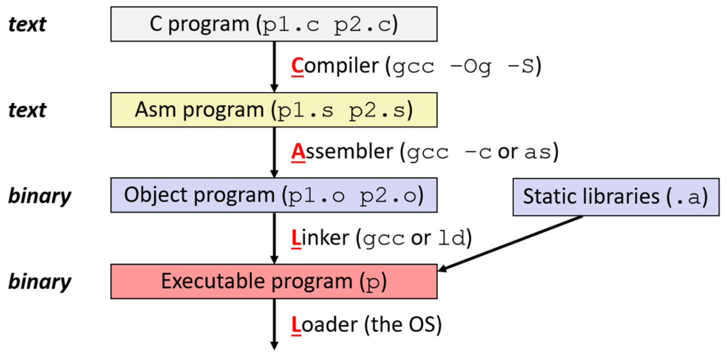
CSE 351 Lecture 13 – Executables & Arrays

Compiling, Assembling, Linking, and Loading (CALL)

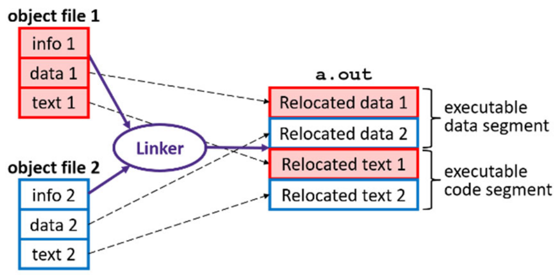
The process of building an executable from C source files is called compilation, which actually has three distinct phases: *compiling*, *assembling*, and *linking*. A standard compiler (*e.g.* `gcc`) command on our source code does all three phases consecutively, but we can ask `gcc` to stop at an intermediate phase if needed or desired. This process is designed to allow us to build a single executable from multiple source files (and libraries). Starting a process from an executable is known as *loading*.



The ***compiler*** will translate a text file (*i.e.* the bytes are meant to be interpreted as characters) of source code into a text file of assembly. In C, the first part of this is a *preprocessor* step that handles preprocessor directives (commands that start with '#' like `#include` and `#define`) – take CSE 333 to learn more. The rest is a complicated process in interpreting the semantic meaning of the source code and writing assembly code that achieves the desired behavior – take CSE 401 to learn more. You can optionally supply *compiler optimization flags* to `gcc` to specify certain attributes that you would like in your assembly code. Common `gcc` optimization flags optimize for code performance (`-O1`, `-O2`, `-O3`), compilation time (`-O0`), code size (`-Os`), or debugging (`-Og`). The output will vary based on compiler version, settings, and flags.

The ***assembler*** will convert a text file of assembly code into a binary ***object file***. There are different formats for object files, but they will contain *object code*, which is incomplete machine code, and information tables. In particular, the object code is "incomplete" because we lack the addresses associated with our labels of the finalized executable; the object file will contain the bytes for the instructions, static data, and literals found in your original source file, but they need to be patched up later once all addresses become known. In order to do the patching, we create two information tables in our object file: the ***symbol table***, which hold the list of globally-accessible labels (*e.g.* function names, global variables) and the ***relocation table***, which holds the list of addresses to be patched.

The ***linker*** will stitch together all the object and static library files needed to produce the final executable. Each object file contains its own symbol table, relocation table, data segment, and instructions, so they are combined into a larger, complete data and code segments for the executable. References are resolved by running through each relocation table (*i.e.* what addresses it *needs*) and finding the corresponding entry in a symbol table (*i.e.* what address it *has*) from any of the object files being linked.



The ***loader*** will take an executable and set up its memory sections and initialize the register values. This is primarily handled by the operating system and will be covered more later during Processes.

Arrays in Assembly

Declaring an array **T** ar[N]; is guaranteed to allocate enough *contiguous* space to hold the specified N elements of size sizeof(**T**). Separate array declarations are not guaranteed to be adjacent. The name of an array (ar) is actually just a *label* in assembly that represents the address of the array. Array subscript notation is just syntactic sugar for address dereferencing: ar[i] ↔ *(ar+i) ↔ Mem[ar + i*sizeof(T)], which we know can be conveniently specified in an x86-64 memory operand:

1) **(Rb, Ri, S)** – with Reg[Rb] = an address, Reg[Ri] = the index, and S = sizeof(**T**)
2) **D(,Ri,S)** – with D = an array name/label, Reg[Ri] = the index, and S = sizeof(**T**)

Note that since an array name is just an assembly label (constant), we can use it as the displacement.

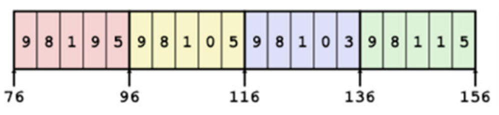
Multidimensional and Multilevel Arrays

A ***multidimensional array*** is still a *contiguous* chunk of memory big enough to hold all of the necessary elements. C utilizes ***row-major ordering***, placing consecutive elements in each *row* next to each other. An N-dimensional array requires N subscripts to access an individual element; fewer subscripts will return the *address* of a larger conceptual element (*e.g.* in a 2D array ar, ar[r][c] returns an element, ar[r] returns the address of row r and ar returns the address of the matrix). Accessing an element is done by an address calculation followed by a *single* memory access.

A ***multilevel array*** is created by adding extra levels of arrays of pointers to arrays. Each individual array is guaranteed to use contiguous memory, but allocating the extra levels takes up more space in total and adds an extra memory access per level.

Multidimensional array

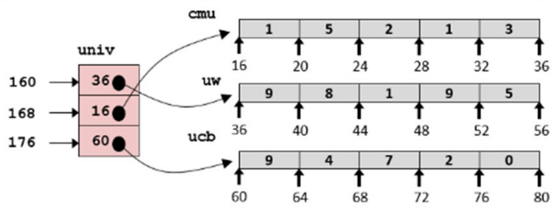
```
int get_sea_digit
  (int index, int digit)
{
  return sea[index][digit];
}
```



Mem[sea+20*index+4*digit]

Multilevel array

```
int get_univ_digit
  (int index, int digit)
{
  return univ[index][digit];
}
```



Mem[Mem[univ+8*index]+4*digit]