CSSE2010

CSSE2010/CSSE7201
Lecture 21

Compilation & Linking

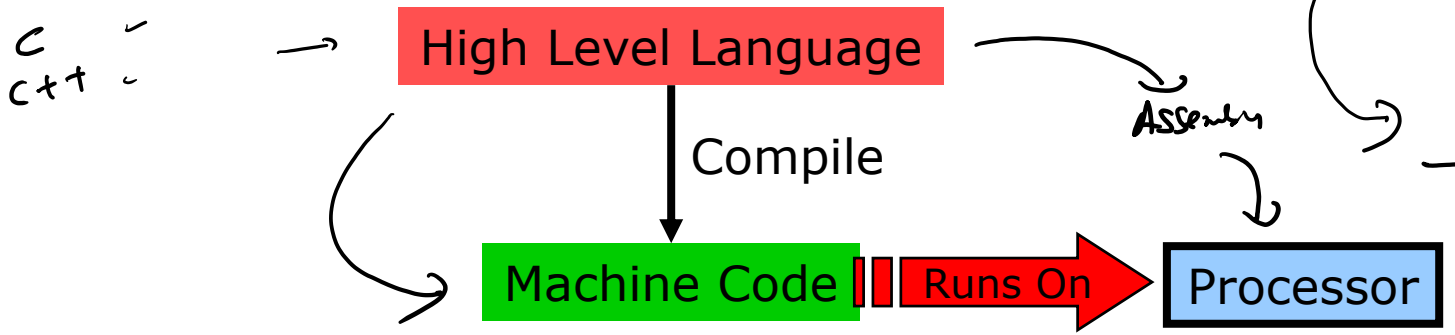
School of Information Technology and Electrical Engineering
The University of Queensland

Admin

- Assignment 2 has been released
  - Get started now
  - Consider the correct version – IN vs EX
  - Refer to getting started videos (IN and EX) and get the base code up and running first
  - Read the specification a few times and the provided code and comments
  - Additional tutor support will be available during weeks 12-13
  - **Keep versions and take backups**
  - Submissions due on Monday 1st Nov 2021 4:00PM AEST.
  - CSSE7201 students will have an additional theoretical questions as part of assignment 2 – released separately and submission separately by the same due date Monday 1st Nov 2021 4:00PM AEST.

Compilation

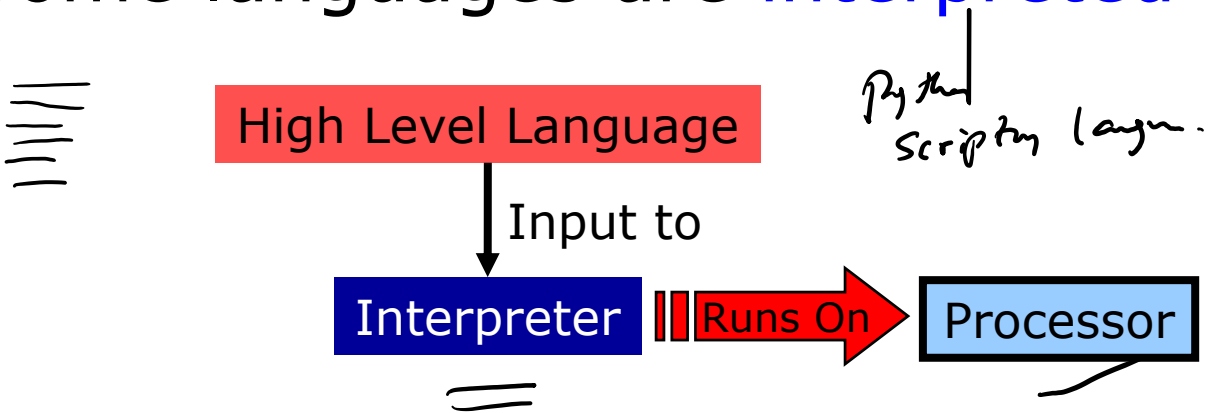
- Programs must be converted to machine code to run
  - Process is called compilation
    - often goes via assembly language



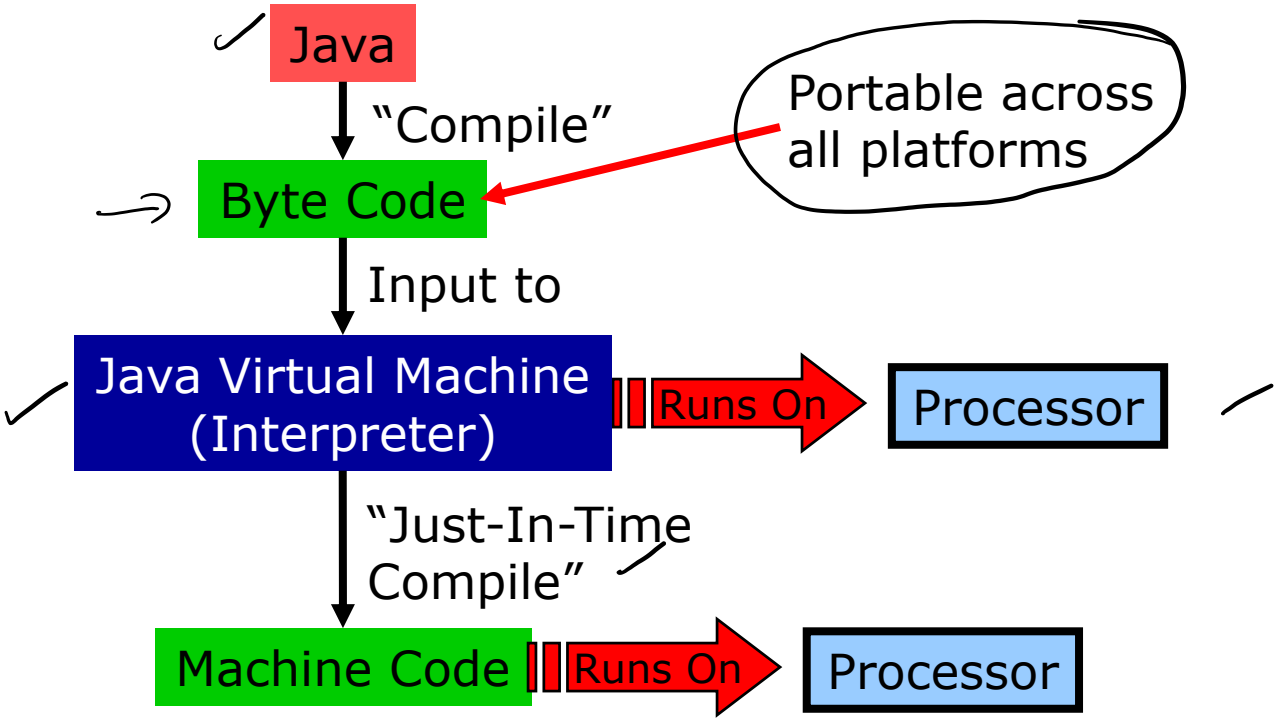
- Not all programs are compiled …

Interpretation

- Some languages are interpreted

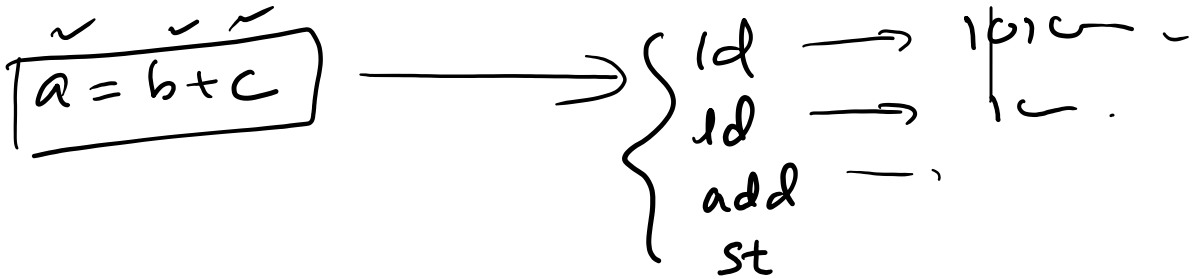


Java

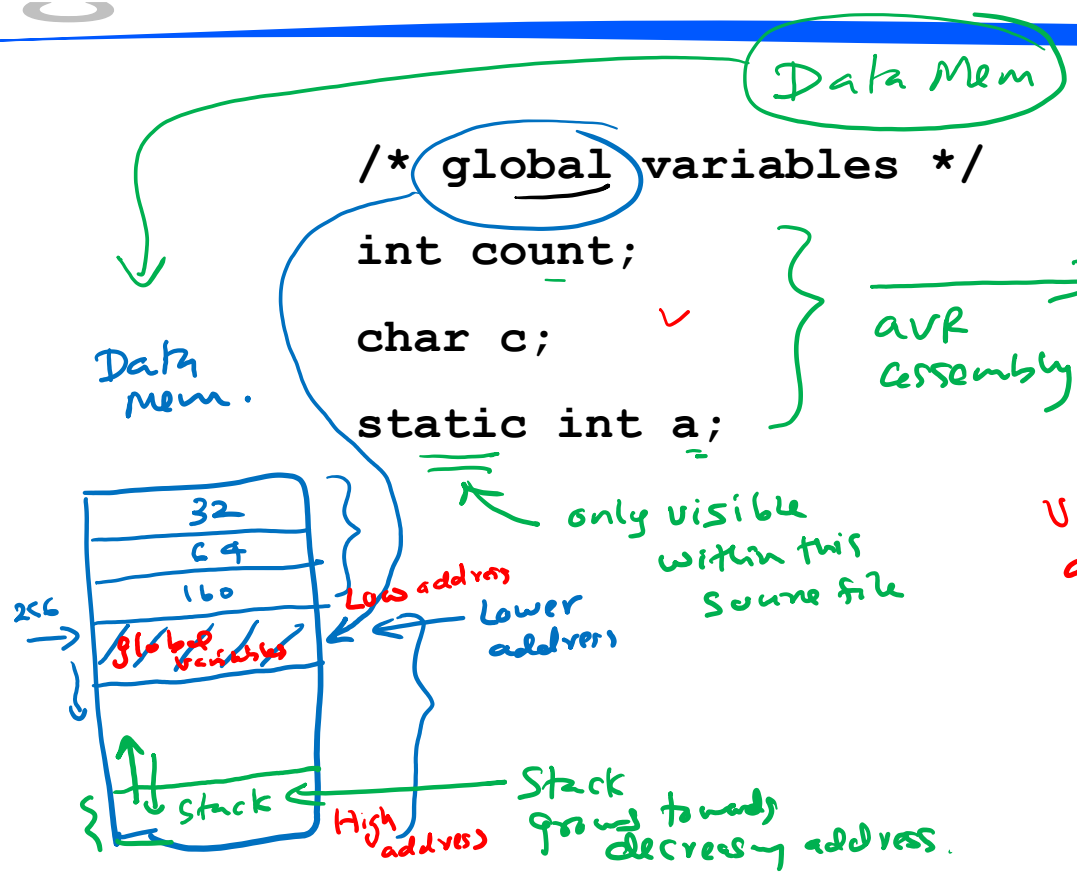

Java
"Compile"
Byte Code
Portable across all platforms
Input to
Java Virtual Machine (Interpreter)
Runs On
Processor
"Just-In-Time Compile"
Machine Code
Runs On
Processor

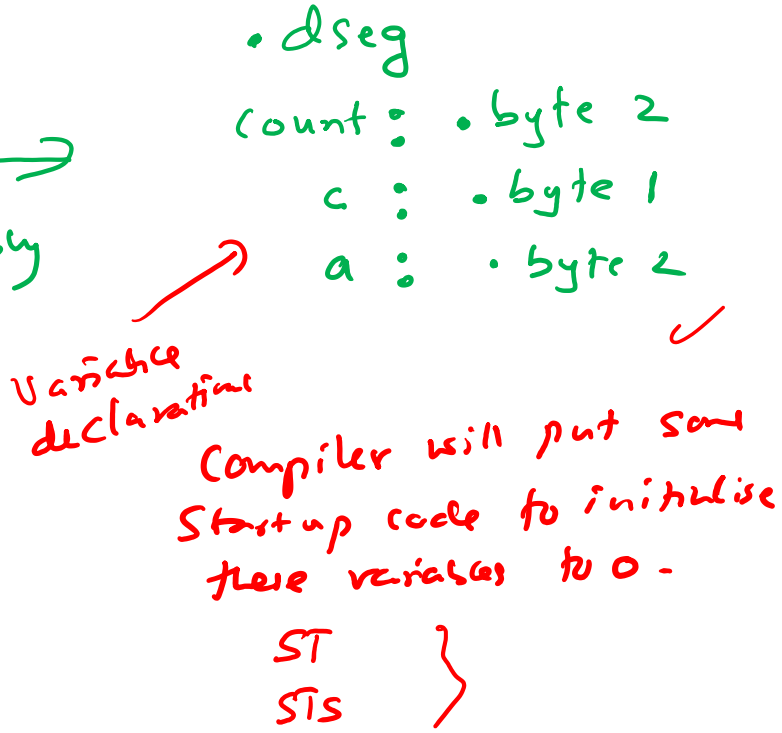
Compiling (Simplified view)

- Mapping variables to memory
- Translating C code to assembler instructions
- From assembler instructions to machine code (you already know this procedure)



How do variables in C map to memory?

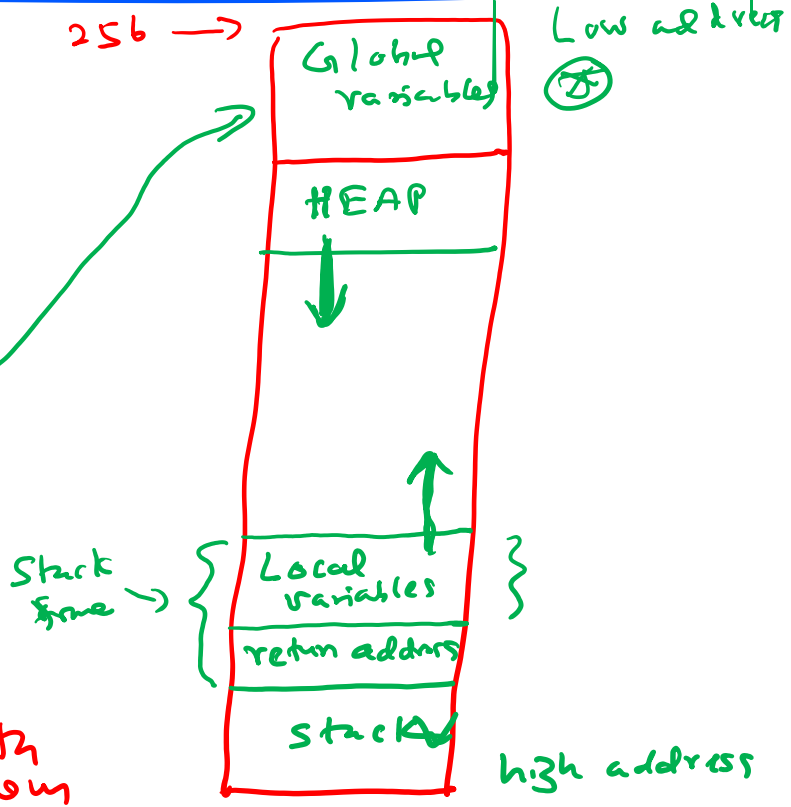




How do variables in C map to memory? (cont.)

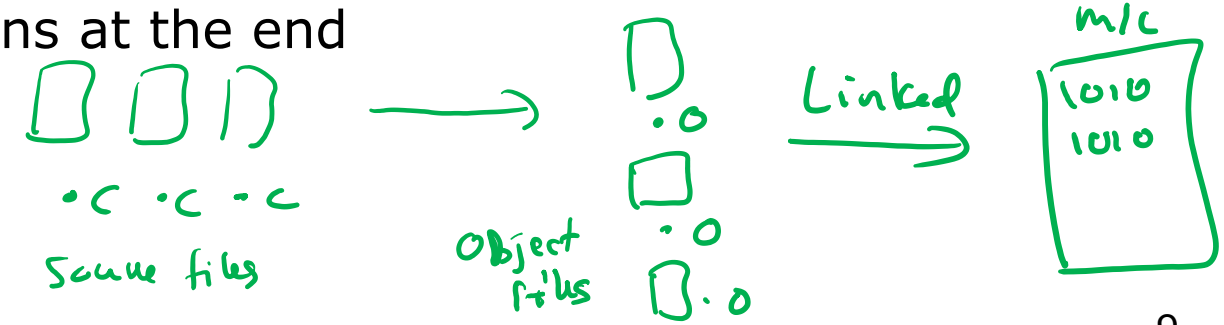
Local variables

```
void function (..) {
        int8_t a;
        uint16_t b;
        static int c;
        …
}
```



Compiling C code

- Similar problem to the assembly process
- Compiler often generates **object files** (.o) – as an intermediate step
  - (Assemblers can generate object files also)
- Generate all necessary object files first (from macros, libraries, .h files)
- Link them all together and work out the final details such as memory locations at the end

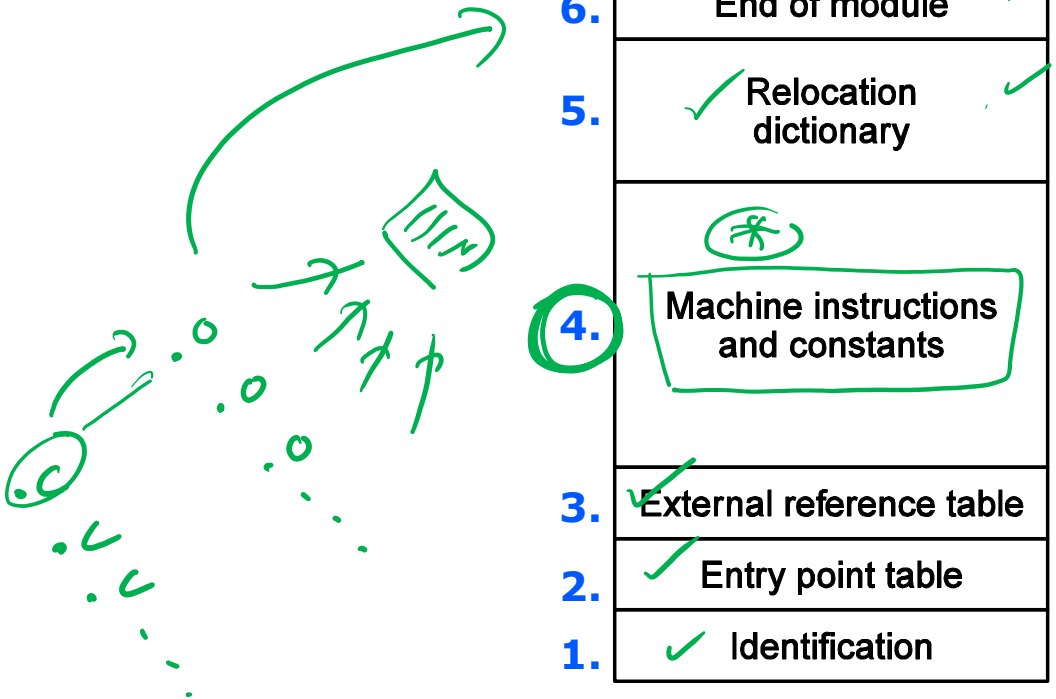


Object Files

- Often useful to be able to compile/assemble modules separately and then link them together
  - Don't need to recompile/reassemble every file if just change one
- What does this mean?
  - We don't know the location of the code (i.e. which addresses it will end up at)
  - When we do know the location, have to "fix" the code
  - To represent this we need what are called **relocatable object modules**
- We'll use terms "object module" and "object file" interchangeably

Object File Structure

object Module



There are many different object file formats, e.g.

- a.out
- COFF
- Dwarf
- ELF

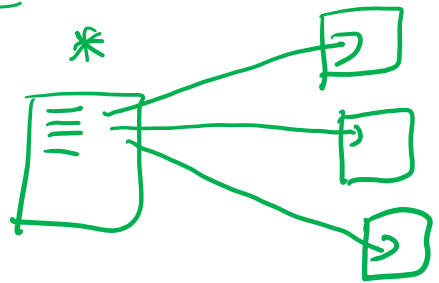
(Figures from Tanenbaum text book)

1. Identification section

- Contains
  - Name of module
  - Location of the other sections in the file
  - Assembly date

2. Entry point table

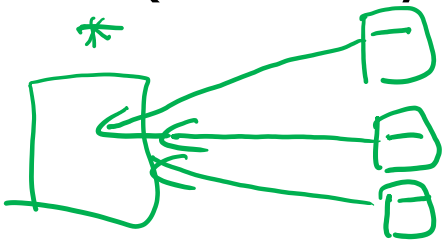
- Contains
  - List of symbols defined in the module that other modules may reference
  - Values of symbols (addresses)
    - Procedure entry points
    - Variables
    - Addresses are relative to beginning of this object module



- Example:
  - file may define a procedure called **abs_value** at address 10

3. External reference table

- Contains
  - List of all symbols that are used in the module but not defined in the module (i.e. they're defined externally)
    - Procedure names
    - Variables
  - List of instructions which use those symbols



4. Machine instructions and constants

- Contains
  - Assembled code and constants
    - Code segment
    - Data segment (initialised and uninitialised data)
  - Note, that addresses referenced in code are not correct – they need **relocation**

5. Relocation Dictionary

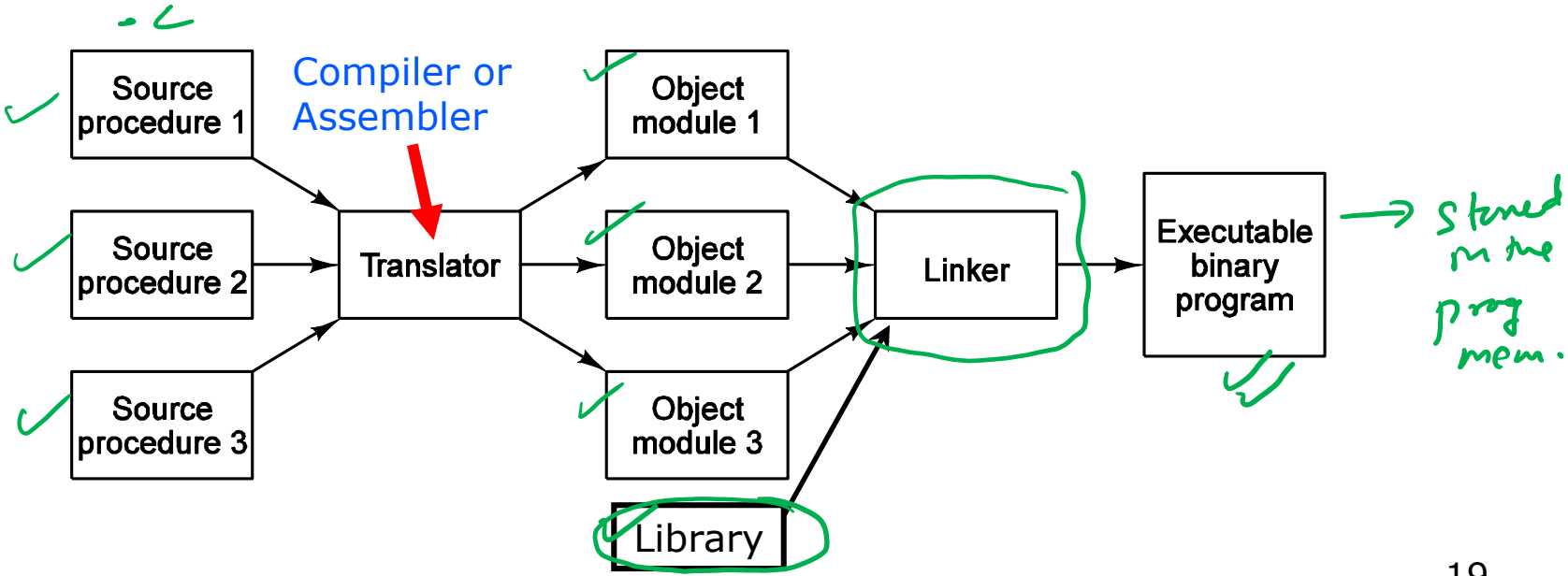
- Contains
  - List of addresses of code/data in section 4 that need to be relocated
- A **relocation constant** will be added to instructions/data at these addresses
  - Different constant for code and data

6. End of Module

- Contains
  - (Maybe) checksum – to catch errors
  - Address at which to start execution

Linker

- **Linker** combines object modules into an executable binary program (machine code)



Libraries

- Library
  - Collection of object files with additional index
  - Provides collection of useful functions/procedures
- Don't reinvent the wheel – use library code where possible
- Example: Standard C library
  - Standard input/output
  - String handling
  - Date and time functions
  - General utilities

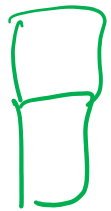
Header files vs Library

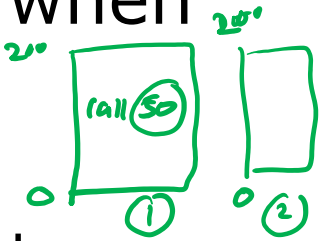
- Header (.h) files are **not** the library
- Header (.h) files
  - define preprocessor macros that are useful
  - declare the functions that are in the library (function prototypes)
  - declare variables that are available externally (extern)
- The library is the collection of object files that implements the functions

#include <math.h>

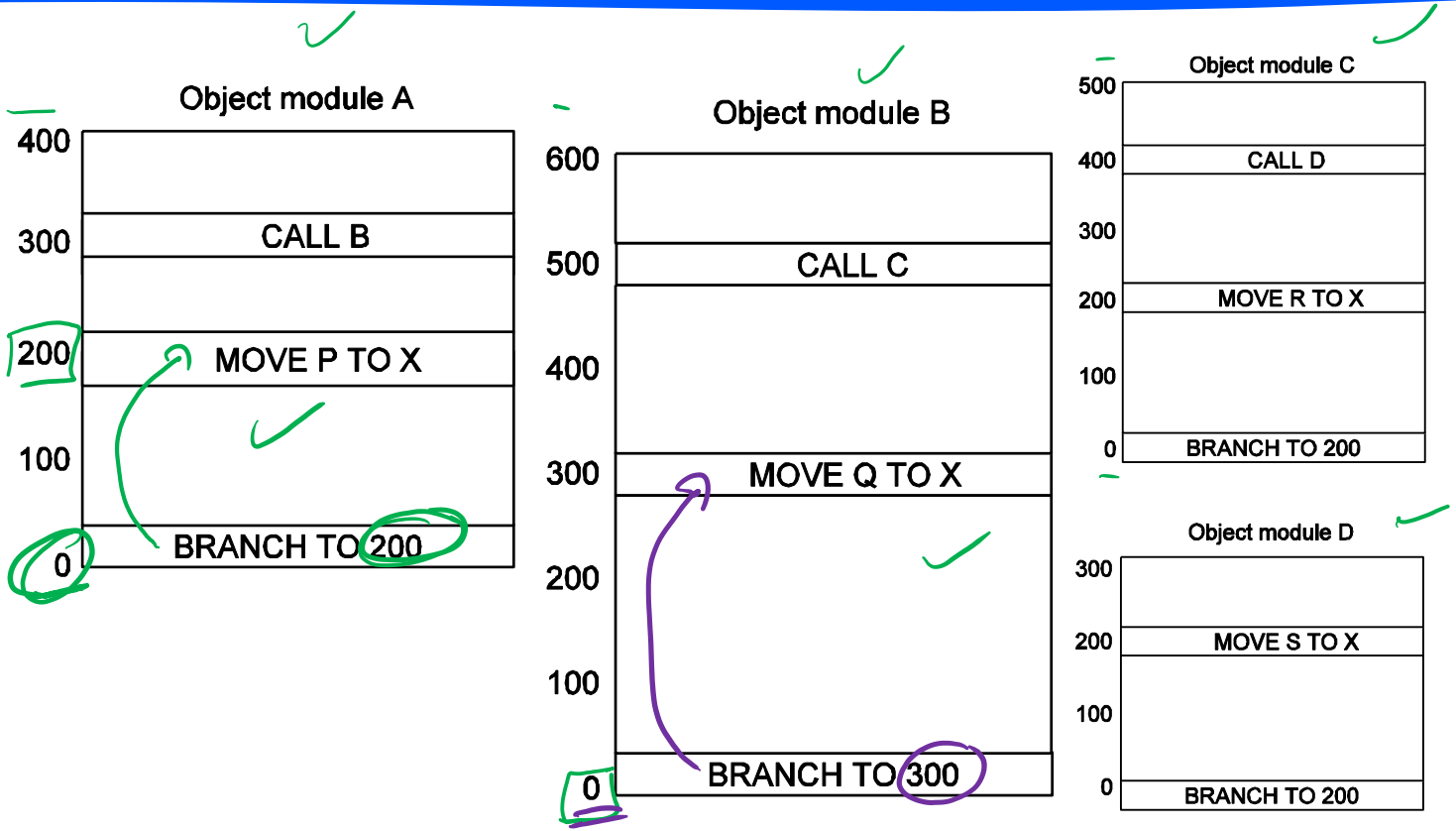
Linking

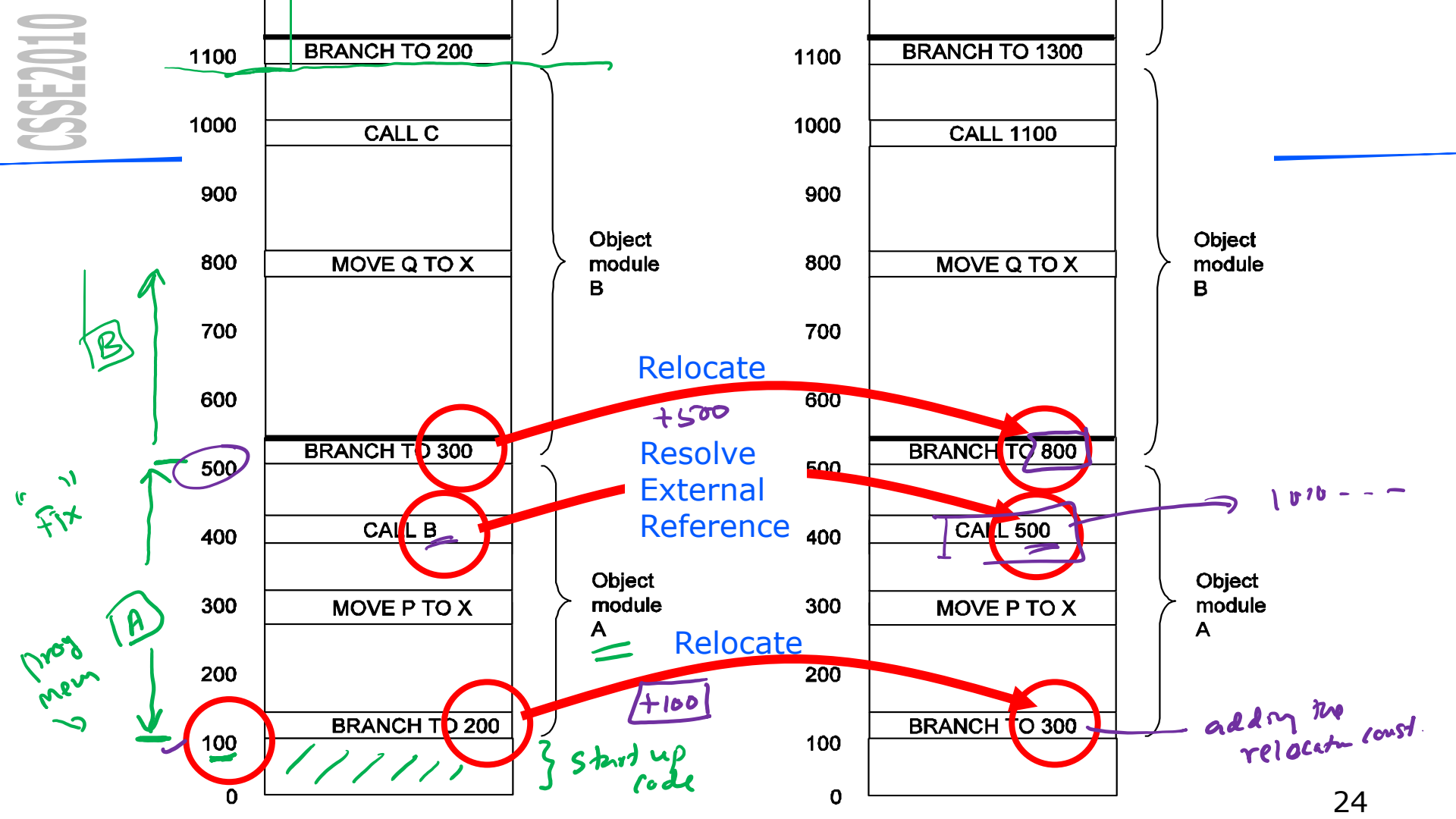
- Similar problem to assembler
  - **External reference problem** – symbols used before value known
- Linker therefore needs two passes as well
- Addresses need relocating or resolving when modules joined together
  - **Relocating** = add some constant value
  - **Resolving** = finding value of "external" address





Example


Object module A
400
300
CALL B
200
MOVE P TO X
100
0
BRANCH TO 200
Object module B
600
500
CALL C
400
300
MOVE Q TO X
200
100
0
BRANCH TO 300
Object module C
500
400
CALL D
300
200
MOVE R TO X
100
0
BRANCH TO 200
Object module D
300
200
MOVE S TO X
100
0
BRANCH TO 200


1100
BRANCH TO 200
1000
CALL C
900
800
MOVE Q TO X
700
600
500
BRANCH TO 300
400
CALL B
300
MOVE P TO X
200
100
BRANCH TO 200
0
Object module B
Object module A
Relocate
+500
Resolve External Reference
Relocate
+100
Start up code
1100
BRANCH TO 1300
1000
CALL 1100
900
800
MOVE Q TO X
700
600
500
BRANCH TO 800
400
CALL 500
300
MOVE P TO X
200
100
BRANCH TO 300
0
Object module B
Object module A
adding the relocation const.
B
A
Prog mem
"fix"

Dynamic Linking

- So far, all object codes (library & main programs) are compiled into binary
- Alternatively, linking can happen at run-time
  - Link when the procedure is first called
- Example
  - DLLs on Windows
- Advantages
  - Saves space
    - Libraries aren't linked into binary executable
  - Library can be updated independently of the programs that use it
    - Can be disadvantage also – "DLL hell" on earlier versions of Windows