Programs in Memory

Bryce Boe

2012/08/29

CS32, Summer 2012 B

Overview

- Project 2 Overview
- Inheritance and assignment operator
- Virtual Keyword / Polymorphism
- Abstract classes
- Compilation Process
- Programs on Disk and in Memory

Project 2: A semi-simple card game

- Turn-based card game where the goal is to eliminate other players by bringing their **hp** down to zero
- Resources (created only once at the start):
  – Cards
    - Can have multiple instances of the *same* card
  – Player

Classes

- Game
  - Deals cards
  - Hands control to players in order
- Player (abstract)
  - Has two sets of cards (deck and discard)
  - Plays cards on other players (or themselves)
- Deck (of cards)
  - Simple AST for holding cards and shuffling
- Card
  - Can attack or heal other players

Relevant Card Interface

- virtual void perform_action(from, to, hp)
  - Called indirectly by player to perform the card's action on another player
- virtual void discard()
  - Called by player when Card is discarded
- virtual int get_hp()
  - Report the hp this card can attack (negative value) or heal (postive value) with

Relevant Player Interface

- virtual void take_turn(const Card& card);
  - Needs to determine who to play card on.

Your Task

- Add additional Cards
  - ReflectorCard
    - Heals the attacker while performing the attack
  - RolloverHPCard
    - Left over hp can be accumulated and used on later turns
  - SnowballCard
    - Becomes stronger each time it is played

Implement Additional Players

- AttackWeakest
  - Always attack the weakest player
- ???
  - Undetermined as of yet

Inheritance and Assignment Operator

- Often want to call parent's assignment operator

Virtual Keyword

- Allows for late binding, aka dynamic dispatch
- Essentially Polymorphism
  - Associate many meanings to one function

Abstract Classes

- Classes can have purely virtual functions (no definition)
- A class with purely virtual functions are said to be abstract classes
- Cannot directly declare instances of abstract classes

```
virtual void output() const = 0;
```

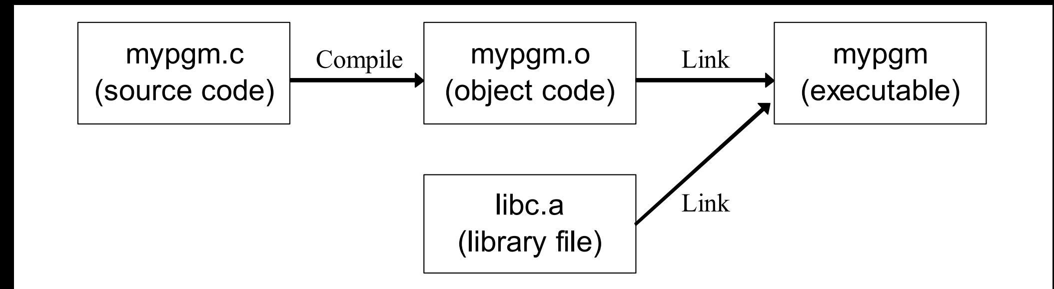
Programs on Disk and Memory

Program building

- Have: source code – human readable instructions
- Need: machine language program – binary instructions and associated data regions, ready to be executed
- clang/gcc does two basic steps: compile, then link
  - To compile means translate to object code
  - To link means to combine with other object code (including library code) into an executable program

Link combines object codes

- From multiple source files and/or libraries
  - e.g., always libc.a



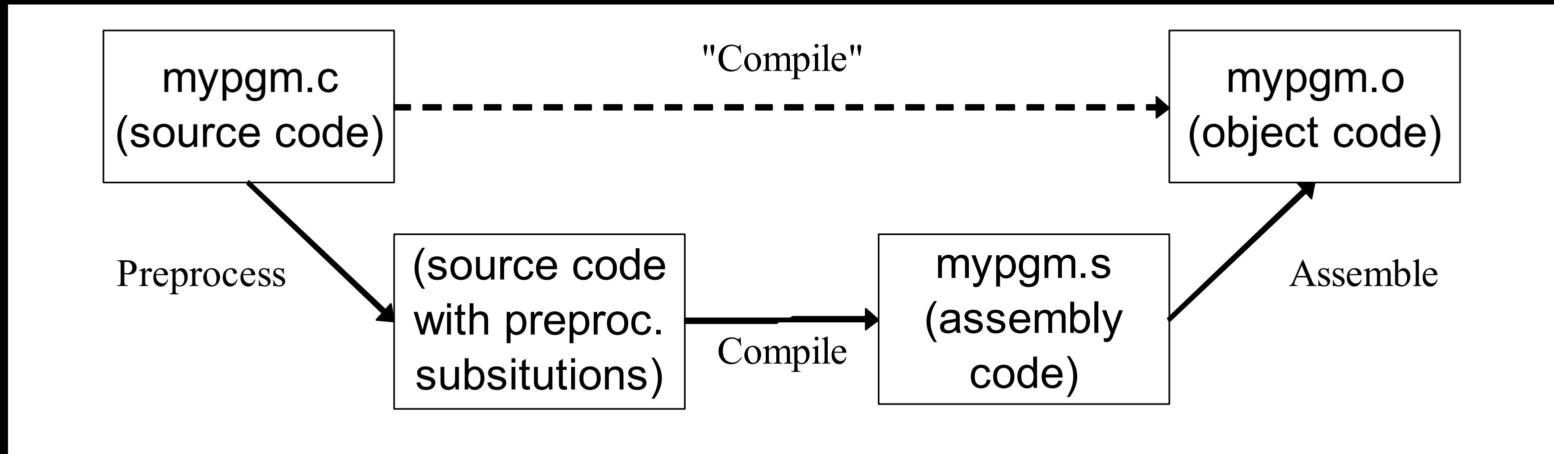
- Use -c option with clang/gcc to stop after creating .o file

```
-bash-4.1$ clang -c mypgm.c ; ls mypgm*
mypgm.c  mypgm.o
```

  - Is necessary to compile a file without a main function
- Later link it to libraries – alone or with other object files:

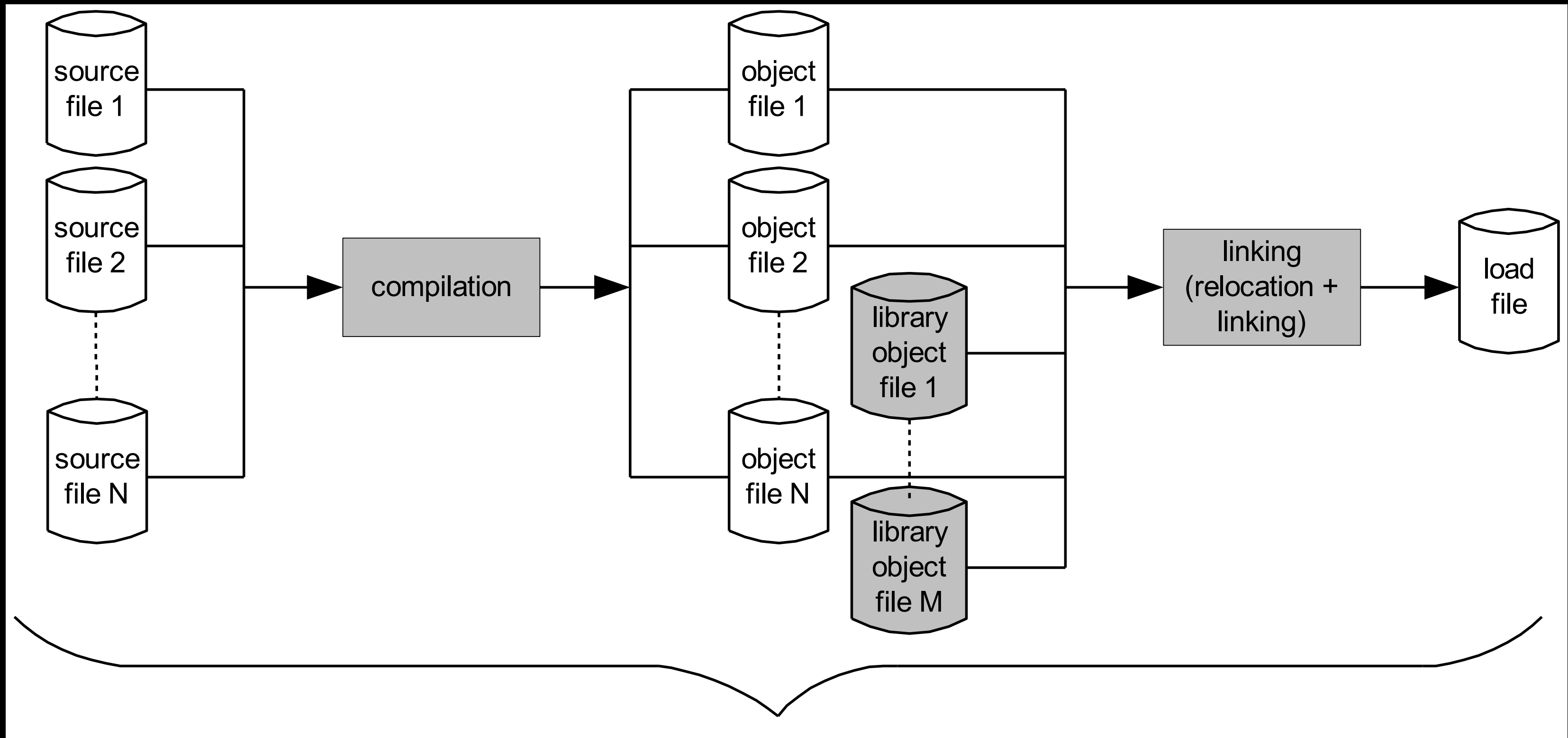
```
-bash-4.1$ clang -o mypgm mypgm.o ; ls mypgm*
mypgm  mypgm.c  mypgm.o
```

Compiling: 3 steps with C/C++



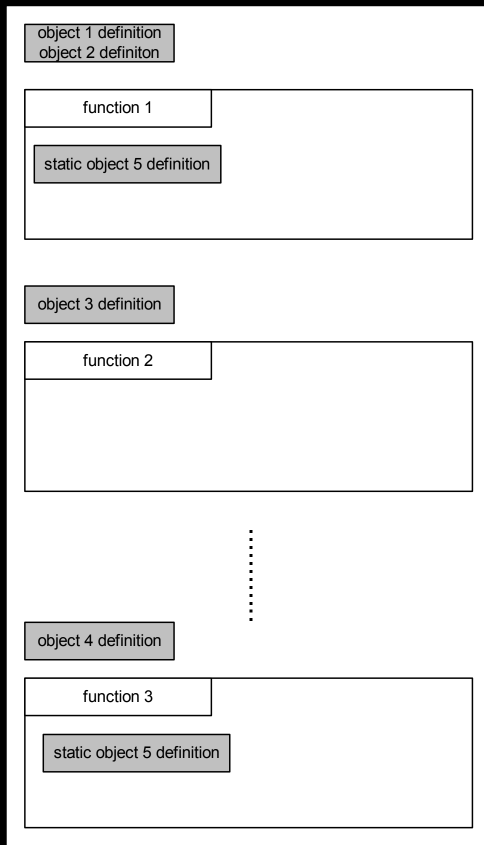
- First the preprocessor runs
  - Creates temporary source code with text substitutions as directed
  - Use `clang -E` to run it alone – output goes to `stdout`
- Then the source is actually compiled to assembly code
  - Use `clang -S` to stop at this step and save code in `.s` file
- Last, assembler produces the object code (machine language)

Another View

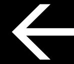


Usually performed by clang/clang++/gcc/g++ in one uninterrupted sequence

Layout of C/C++ programs

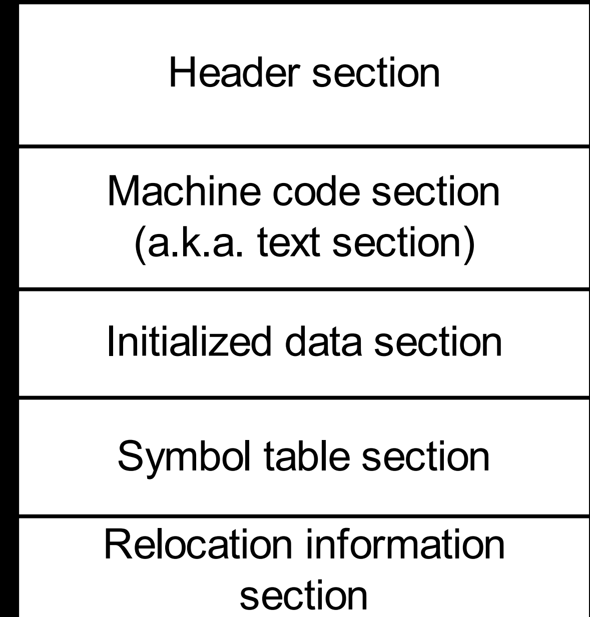


Source code



… becomes

Object module →



A sample C program – demo.c

```
#include <stdio.h>

int a[10]={0,1,2,3,4,5,6,7,8,9};
int b[10];

void main(){
   int i;
   static int k = 3;

   for(i = 0; i < 10; i++) {
    printf("%d\n",a[i]);
    b[i] = k*a[i];
   }
}
```

- Has text section: the machine code
- Has initialized global data: `a`
- Uninitialized global data: `b`
- Static data: `k`
- Has a local variable: `i`

A possible structure of demo.o

Offset	*Contents*	*Comment*
Header section		
0	124	number of bytes of Machine code section
4	44	number of bytes of initialized data section
8	40	number of bytes of Uninitialized data section (array **`b[]`**) (*not part of this object module*)
12	60	number of bytes of Symbol table section
16	44	number of bytes of Relocation information section
Machine code section (124 bytes)		
20	X	code for the top of the **`for`** loop (36 bytes)
56	X	code for call to **`printf()`** (22 bytes)
68	X	code for the assignment statement (10 bytes)
88	X	code for the bottom of the **`for`** loop (4 bytes)
92	X	code for exiting **`main()`** (52 bytes)
Initialized data section (44 bytes)		
144	0	beginning of array **`a[]`**
148	1	
:		
176	8	
180	9	end of array **`a[]`** (40 bytes)
184	3	variable **`k`** (4 bytes)
Symbol table section (60 bytes)		
188	X	array **`a[]`** : offset 0 in Initialized data section (12 bytes)
200	X	variable **`k`** : offset 40 in Initialized data section (10 bytes)
210	X	array **`b[]`** : offset 0 in Uninitialized data section (12 bytes)
222	X	**`main`** : offset 0 in Machine code section (12 bytes)
234	X	**`printf`** : external, used at offset 56 of Machine code section (14 bytes)
Relocation information section (44 bytes)		
248	X	relocation information

Object module contains neither uninitialized data (`b`), nor any local variables (`i`)

Linux object file format

- "ELF" – stands for Executable and Linking Format
  - A 4-byte magic number followed by a series of named sections
- Addresses assume the object file is placed at memory address 0
  - When multiple object files are linked together, we must update the offsets (relocation)
- Tools to read contents: `objdump` and `readelf` – not available on all systems

\177ELF
.text
…
.rodata
…
.data
…
.bss
…
.symtab
…
.rel.text
…
.rel.data
…
.debug
…
.line
…
Section header table

ELF sections

- .text = machine code (compiled program instructions)
- .rodata = read-only data
- .data = initialized global variables
- .bss = "block storage start" for uninitialized global variables – actually just a placeholder that occupies no space in the object file
- .symtab = symbol table with information about functions and global variables defined and referenced in the program

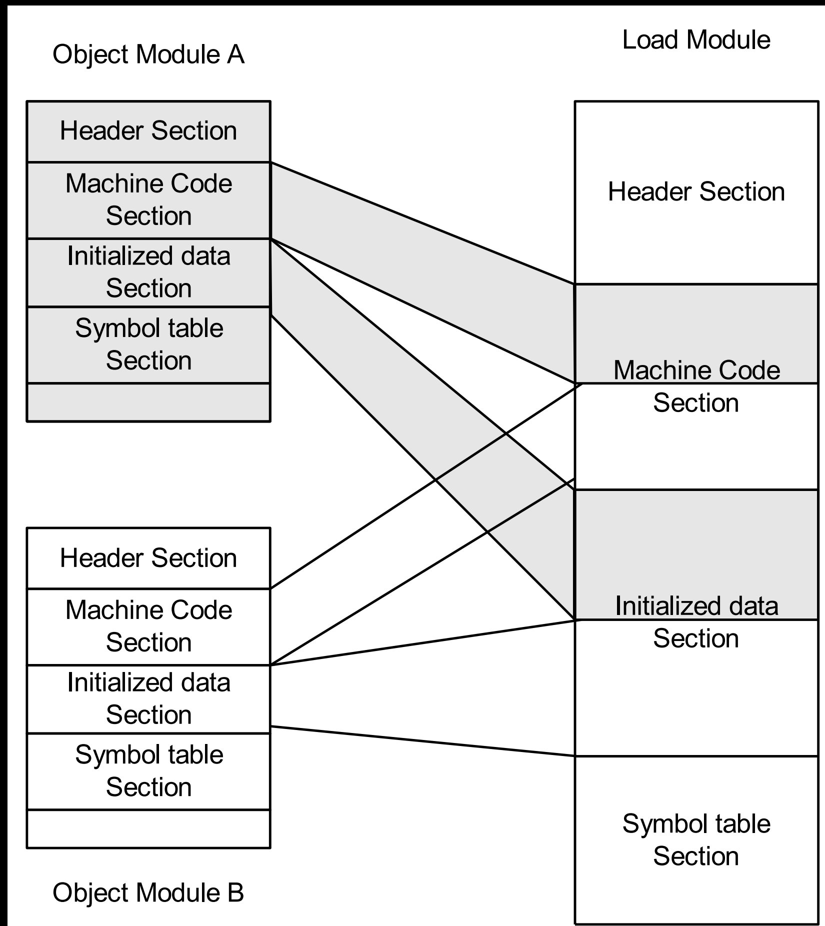
\177ELF
.text
…
.rodata
…
.data
…
.bss
…
.symtab
…
.rel.text
…
.rel.data
…
.debug
…
.line
…
Section header table

ELF Sections (cont.)

- .rel.text = list of locations in .text section that need to be modified when linked with other object files
- .rel.data = relocation information for global variables referenced but not defined
- .debug = debugging symbol table; only created if compiled with -g option
- .line = mapping between line numbers in source and machine code in .text; used by debugger programs

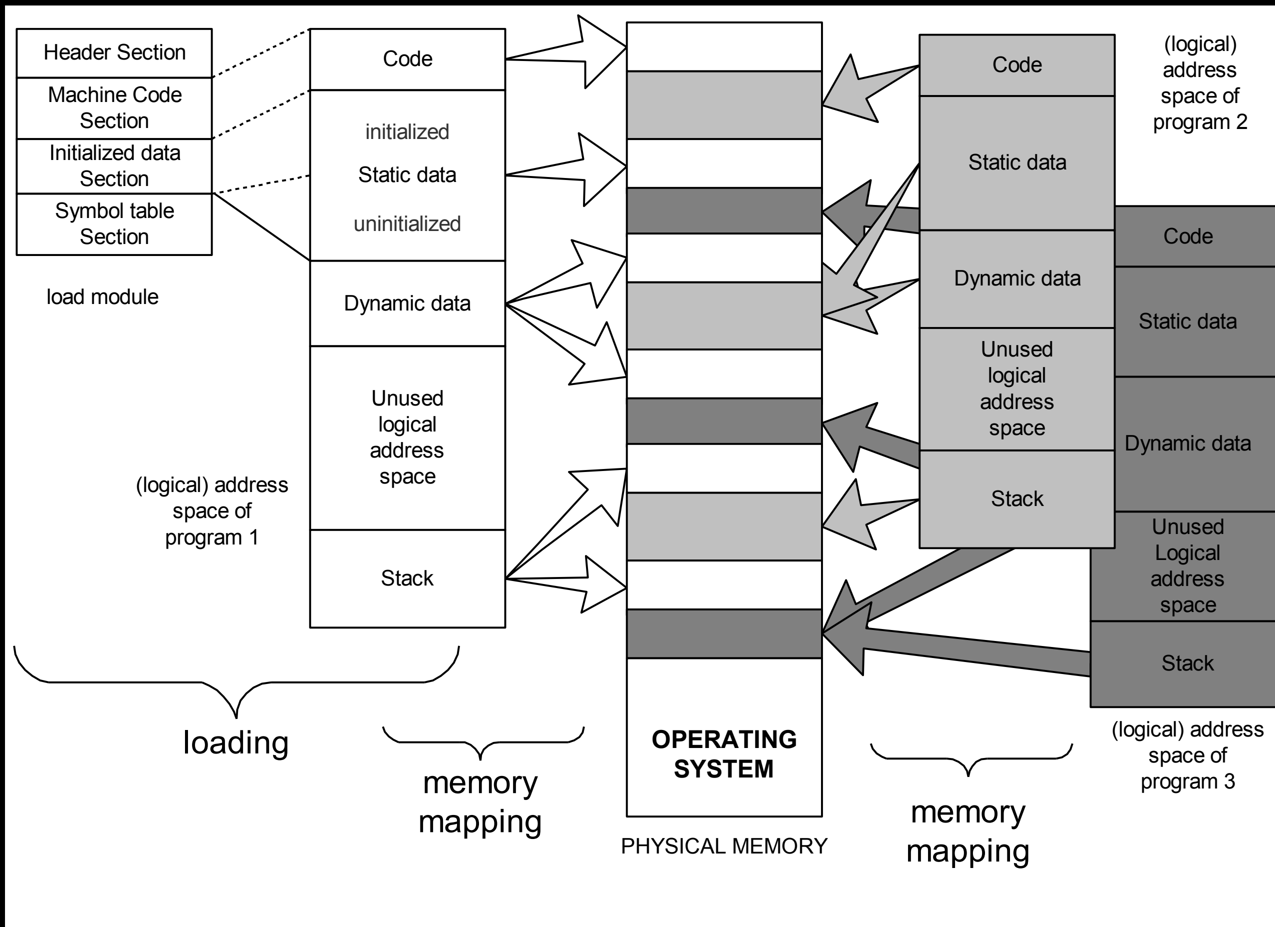
\177ELF
.text
…
.rodata
…
.data
…
.bss
…
.symtab
…
.rel.text
…
.rel.data
…
.debug
…
.line
…
Section header table

Creation of a load module



- Interleaved from multiple object modules
  - Sections must be "relocated"
- Addresses relative to beginning of a module
  - Necessary to translate from beginnings of object modules
- When loaded – OS will translate again to absolute addresses

Loading and memory mapping



- Includes memory for stack, dynamic data (i.e., free store), and un-initialized global data
- Physical memory is shared by multiple programs

Sections of an executable file