Executables & Arrays
CSE 351 Autumn 2022

Instructor: Justin Hsia

Teaching Assistants:
Angela Xu
Assaf Vayner
David Dai
James Froelich
Paul Stevans

Arjun Narendra
Carrie Hu
Dominick Ta
Jenny Peng
Renee Ruan

Armin Magness
Clare Edmonds
Effie Zheng
Kristina Lansang
Vincent Xiao

http://xkcd.com/1790/
Relevant Course Information

- Lab 2 & hw12 due Friday (10/28)
- hw13 due next Wednesday (11/2)
  - Based on the next two lectures, longer than normal

- Midterm (take home, 11/3-11/5)
  - Midterm review problems in section next week
  - Make notes and use the midterm reference sheet
  - Form study groups and look at past exams!
GDB Demo #2

❖ Let’s examine the pcount_r stack frames on a real machine!
  - Using pcount.c from the course website

❖ You will need to use GDB to get through the Midterm
  - Useful debugger in this class and beyond!

❖ Pay attention to:
  - Checking the current stack frames (backtrace)
  - Getting stack frame information (info frame <#>)
  - Examining memory (x)
Instruction Set Philosophies, Revisited

- **Complex Instruction Set Computing (CISC):**
  - Add more and more elaborate and specialized instructions as needed
  - **Design goals:** complete tasks in as few instructions as possible; minimize memory accesses for instructions

- **Reduced Instruction Set Computing (RISC):**
  - Keep instruction set small and regular
  - **Design goals:** build fast hardware; instructions should complete in few clock cycles (ideally 1); minimize complexity and maximize performance

- How different are these two philosophies, really?
Instruction Set Philosophies, Revisited

- **Complex Instruction Set Computing (CISC):** Add more and more elaborate and specialized instructions as needed
  - **Design goals:** complete tasks in as few instructions as possible; minimize memory accesses for instructions

- **Reduced Instruction Set Computing (RISC):** Keep instruction set small and regular
  - **Design goals:** build fast hardware; instructions should complete in few clock cycles (ideally 1); minimize complexity and maximize performance

- How different are these two philosophies, really?
  - Both pursue **efficiency** (**minimalism** is a means to an end)
Mainstream ISAs, Revisited

Macbooks & PCs
(Core i3, i5, i7, M)
* x86-64 Instruction Set

Smartphone-like devices
(iPhone, iPad, Raspberry Pi)
* ARM Instruction Set

 Mostly research
(some traction in embedded)
* RISC-V Instruction Set

Does anything feel “off” about this landscape?
Tech Monopolization

❖ How many “dominant” ISAs are there?
  ▪ 2: x86, ARM

❖ How many “dominant” phone brands are there?
  ▪ 4: Samsung, Apple, Huawei, Xiaomi

❖ How many “dominant” operating systems are there?
  ▪ 3/4: Android, iOS/macOS, Windows, Linux (?)

❖ How many “dominant” chip manufacturers are there?
  ▪ 3: Intel, Samsung, TSMC

❖ It wasn’t always this way!
  ▪ Combination of antitrust policies and (lack of) enforcement
Assembly Discussion Questions

❖ We taught you assembly using x86-64; you didn’t have a choice
  ▪ What are some of the advantages of this choice?

  ▪ What are some of the drawbacks of this choice?

  ▪ What are some possible assumptions we are making about our students or values we are forcing on our students with this choice?
The Hardware/Software Interface

❖ Topic Group 2: **Programs**
  ▪ x86-64 Assembly, Procedures, Stacks, **Executables**

❖ How are programs created and executed on a CPU?
  ▪ How does your source code become something that your computer understands?
  ▪ How does the CPU organize and manipulate local data?
Reading Review

❖ Terminology:

▪ CALL: compiler, assembler, linker, loader
▪ Object file: symbol table, relocation table
▪ Disassembly
▪ Multidimensional arrays, row-major ordering
▪ Multilevel arrays

❖ Questions from the Reading?
Building an Executable with C (Review)

- Code in files `p1.c p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Put resulting machine code in file `p`
- Run with command: `. /p`

```

<table>
<thead>
<tr>
<th>text</th>
<th>C program (p1.c p2.c)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compiler (gcc -Og -S)</td>
</tr>
<tr>
<td>text</td>
<td>Asm program (p1.s p2.s)</td>
</tr>
<tr>
<td></td>
<td>Assembler (gcc -c or as)</td>
</tr>
<tr>
<td>binary</td>
<td>Object program (p1.o p2.o)</td>
</tr>
<tr>
<td></td>
<td>Static libraries (.a)</td>
</tr>
<tr>
<td>binary</td>
<td>Executable program (p)</td>
</tr>
<tr>
<td></td>
<td>Linker (gcc or ld)</td>
</tr>
<tr>
<td></td>
<td>Loader (the OS)</td>
</tr>
</tbody>
</table>
```
Compiler (Review)

- **Input**: Higher-level language code (*e.g.*, C, Java)
  - `foo.c`
- **Output**: Assembly language code (*e.g.*, x86, ARM, MIPS)
  - `foo.s`

- First there’s a preprocessor step to handle `#`directives
  - Macro substitution, plus other specialty directives

- Super complex, whole courses devoted to these!

- Compiler optimizations
  - “Level” of optimization specified by capital ‘O’ flag (*e.g.* `-Og`, `-O3`)
Compiling Into Assembly (Review)

- **C Code** *(sum.c)*

```c
void sumstore(long x, long y, long *dest) {
    long t = x + y;
    *dest = t;
}
```

- **x86-64 assembly** *(gcc -Og -S sum.c)*

```assembly
sumstore(long, long, long*):
    addq %rdi, %rsi
    movq %rsi, (%rdx)
    ret
```

**Warning**: You may get different results with other versions of gcc and different compiler settings
Assembler (Review)

- **Input:** Assembly language code (e.g., x86, ARM, MIPS)
  - foo.s

- **Output:** Object files (e.g., ELF, COFF)
  - foo.o
  - Contains object code and information tables

- Reads and uses *assembly directives*
  - *e.g.*, .text, .data, .quad
  - x86: [https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html](https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html)

- Produces “machine language”
  - Does its best, but object file is *not* a completed binary

- **Example:** gcc -c foo.s
Producing Machine Language (Review)

- **Simple cases:** arithmetic and logical operations, shifts, etc.
  - All necessary information is contained in the instruction itself

- Addresses and labels are problematic because the final executable hasn’t been constructed yet!
  - Conditional and unconditional jumps
  - Accessing static data (e.g., global variable or jump table)
  - `call`

- So how do we deal with these in the meantime?
Object File Information Tables (Review)

- Each object file has its own symbol and relocation tables

- **Symbol Table** holds list of “items” that may be used by other files
  - *Non-local labels* – function names for *call*
  - *Static Data* – variables & literals that might be accessed across files

- **Relocation Table** holds list of “items” that this file needs the address of later (currently undetermined)
  - Any *label* or piece of *static data* referenced in an instruction in this file
    - Both internal and external
Object File Format

1) **object file header**: size and position of the other pieces of the object file
2) **text segment**: the machine code
3) **data segment**: data in the source file (binary)
4) **relocation table**: identifies lines of code that need to be “handled”
5) **symbol table**: list of this file’s labels and data that can be referenced
6) **debugging information**

❖ More info: ELF format
  - [http://www.skyfree.org/linux/references/ELF_Format.pdf](http://www.skyfree.org/linux/references/ELF_Format.pdf)
Linker (Review)

- **Input:** Object files (*e.g.*, ELF, COFF)
  - `foo.o`

- **Output:** executable binary program
  - `a.out`

- Combines several object files into a single executable (*linking*)
- Enables separate compilation/assembling of files
  - Changes to one file do not require recompiling of whole program
Linking (Review)

1) Take text segment from each `.o` file and put them together
2) Take data segment from each `.o` file, put them together, and concatenate this onto end of text segments
3) Resolve References
   - Go through Relocation Table; handle each entry
Disassembling Object Code (Review)

❖ Disassembled:

```
000000000000400536 <sumstore>:
  400536:  48 01 fe  add %rdi,%rsi
  400539:  48 89 32  mov %rsi,(%rdx)
  40053c:  c3    retq
```

❖ **Disassembler** *(objdump -d sum)*

- Useful tool for examining object code *(man 1 objdump)*
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can run on either executable or object file
What Can be Disassembled?

- Anything that can be interpreted as executable code
- Disassembler examines bytes and attempts to reconstruct assembly source

```
% objdump -d WINWORD.EXE

WINWORD.EXE: file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:
30001001:
30001003:
30001005:
3000100a:
```

Reverse engineering forbidden by Microsoft End User License Agreement
Loader (Review)

- **Input:** executable binary program, command-line arguments
  - ./a.out arg1 arg2
- **Output:** <program is run>

- Loader duties primarily handled by OS/kernel
  - More about this when we learn about processes
- Memory sections (Instructions, Static Data, Stack) are set up
- Registers are initialized
The Hardware/Software Interface

- Topic Group 1: **Data**
  - Memory, Data, Integers, Floating Point, **Arrays**, Structs

- How do we store information for other parts of the house of computing to access?
  - How do we represent data and what limitations exist?
  - What design decisions and priorities went into these encodings?
Data Structures in C

❖ **Arrays**
  - One-dimensional
  - Multidimensional (nested)
  - Multilevel

❖ **Structs**
  - Alignment

❖ **Unions**
Array Allocation (Review)

❖ Basic Principle
- **T A[N]; →** array of data type T and length N
- *Contiguously* allocated region of $N \times \text{sizeof}(T)$ bytes
- Identifier A returns address of array (type T*)

```c
char msg[12];
int val[5];
double a[3];
char* p[3];
```

```
x  |  x + 4  |  x + 8  |  x + 12 |  x + 16 |  x + 20
---|---------|---------|---------|---------|--------
>x  |         |         |         |         |        
```

```
x  |  x + 8  |  x + 16 |  x + 24
---|---------|---------|--------
>x  |         |         |        
```

```
x  |  x + 8  |  x + 16 |  x + 24
---|---------|---------|--------
>x  |         |         |        
```

```
x  |  x + 8  |  x + 16 |  x + 24
---|---------|---------|--------
>x  |         |         |        
```

```
x  |  x + 8  |  x + 16 |  x + 24
---|---------|---------|--------
>x  |         |         |        
```

```
x  |  x + 8  |  x + 16 |  x + 24
---|---------|---------|--------
>x  |         |         |        
```
Array Access (Review)

❖ Basic Principle

- \( T \ A[N]; \rightarrow \) array of data type \( T \) and length \( N \)
- Identifier \( A \) returns address of array (type \( T^* \))

\[
\text{int } x[5];
\]

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x[4] )</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>( x )</td>
<td>int*</td>
<td>( a )</td>
</tr>
<tr>
<td>( x+1 )</td>
<td>int*</td>
<td>( a + 4 )</td>
</tr>
<tr>
<td>&amp;( x[2] )</td>
<td>int*</td>
<td>( a + 8 )</td>
</tr>
<tr>
<td>( x[5] )</td>
<td>int</td>
<td>?? (whatever’s in memory at addr ( x+20 ))</td>
</tr>
<tr>
<td>( *(x+1) )</td>
<td>int</td>
<td>7</td>
</tr>
<tr>
<td>( x+i )</td>
<td>int*</td>
<td>( a + 4*i )</td>
</tr>
</tbody>
</table>
Array Example

// arrays of ZIP code digits
int cmu[5] = { 1, 5, 2, 1, 3 };
int uw[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };

- Example arrays happened to be allocated in successive 20 byte blocks
  - Not guaranteed to happen in general
C Details: Arrays and Pointers

❖ Arrays are (almost) identical to pointers
  ▪ `char* string` and `char string[]` are nearly identical declarations
  ▪ Differ in subtle ways: initialization, `sizeof()`, etc.

❖ An array name is an expression (not a variable) that returns the address of the array
  ▪ It *looks* like a pointer to the first (0th) element
    • `*ar` same as `ar[0]`, *(ar+2)* same as `ar[2]`
  ▪ An array name is read-only (no assignment) because it is a *label*
    • Cannot use "`ar = <anything>`"
C Details: Arrays and Functions

❖ Declared arrays only allocated while the scope is valid:

```c
char* foo() {
    char string[32]; ...;
    return string;
}
```

❖ An array is passed to a function as a pointer:
  ▪ Array size gets lost!

```c
int foo(int ar[], unsigned int size) {
    ... ar[size-1] ... 
}
```

BAD!

Must explicitly pass the size!
Data Structures in C

❖ **Arrays**
  - One-dimensional
  - **Multidimensional (nested)**
  - Multilevel

❖ **Structs**
  - Alignment

❖ **Unions**
Nested Array Example

```c
int sea[4][5] =
    {{ 9, 8, 1, 9, 5 },
     { 9, 8, 1, 0, 5 },
     { 9, 8, 1, 0, 3 },
     { 9, 8, 1, 1, 5 }};
```

- What is the layout in memory?

Remember, `T A[N]` is an array with elements of type `T`, with length `N`
Nested Array Example

```
int sea[4][5] =
{
{ 9, 8, 1, 9, 5 },
{ 9, 8, 1, 0, 5 },
{ 9, 8, 1, 0, 3 },
{ 9, 8, 1, 1, 5 }
};
```

- **“Row-major” ordering of all elements**
  - Elements in the same row are contiguous
  - Guaranteed (in C)

Remember, $\text{T } A[N]$ is an array with elements of type $\text{T}$, with length $N$. 

```
sea[3][2];
```
Two-Dimensional (Nested) Arrays

- Declaration: \( T \ A[R][C] \);
  - 2D array of data type \( T \)
  - \( R \) rows, \( C \) columns
  - Each element requires \( \text{sizeof}(T) \) bytes

- Array size?
Two-Dimensional (Nested) Arrays

❖ Declaration: \( T [R][C]; \)
  - 2D array of data type \( T \)
  - \( R \) rows, \( C \) columns
  - Each element requires \( \text{sizeof}(T) \) bytes

❖ Array size:
  - \( R \times C \times \text{sizeof}(T) \) bytes

❖ Arrangement: row-major ordering

```c
int A[R][C];
```

| A [0] [0] | \( \cdots \) | A [0] [C-1] | \( \cdots \) | A [1] [0] | \( \cdots \) | A [1] [C-1] | \( \cdots \) | A [R-1] [0] | \( \cdots \) | A [R-1] [C-1] |
|-----------|----------------|--------------|----------------|-----------|----------------|--------------|----------------|-----------|----------------|--------------|--------------|
| \( A[0][0] \) | \( \cdots \) | \( A[0][C-1] \) | \( \cdots \) | \( A[1][0] \) | \( \cdots \) | \( A[1][C-1] \) | \( \cdots \) | \( A[R-1][0] \) | \( \cdots \) | \( A[R-1][C-1] \) |

4 \( \times R \times C \) bytes
Nested Array **Row Access**

- **Row vectors**
  - Given $\mathbf{T} \ A[R][C]$, 
    - $A[i]$ is an array of $C$ elements (“row $i$”)
    - $A$ is address of array
    - Starting address of row $i = A + i*(C \times \text{sizeof}(T))$

```c
int A[R][C];
```

- $A[0]$ is $A[0][0]$ to $A[0][C-1]$
- $A[i]$ is $A[i][0]$ to $A[i][C-1]$
- $A[R-1]$ is $A[R-1][0]$ to $A[R-1][C-1]$
Nested Array Element Access

- **Array Elements**
  - $A[i][j]$ is element of type $T$; let $\text{sizeof}(T) = t$ bytes
  - Address of $A[i][j]$ is

```
int A[R][C];
```

![Diagram of nested array and element access](image)
Nested Array Element Access

- **Array Elements**
  - $A[i][j]$ is element of type $T$; let $\text{sizeof}(T) = t$ bytes
  - Address of $A[i][j]$ is
    $$A + i*(C*t) + j*t = A + (i*C + j)*t$$

```c
int A[R][C];
```
Data Structures in C

❖ Arrays
  ▪ One-dimensional
  ▪ Multidimensional (nested)
  ▪ Multilevel

❖ Structs
  ▪ Alignment

❖ Unions
Multilevel Array Example

Multilevel Array Declaration(s):

```java
int cmu[5] = { 1, 5, 2, 1, 3 };
int uw[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };

int* univ[3] = {uw, cmu, ucb};
```

- Variable `univ` denotes array of 3 pointer elements
- Each pointer points to a separate array of `ints`
  - *Could* have inner arrays of different lengths!

Note: this is how Java represents multidimensional arrays!
Multilevel Array Element Access

- Mem[Mem[univ+8*index]+4*digit]
  - Must do two memory reads: (1) get pointer to row array, (2) access element within array

```cpp
int get_univ_digit (int index, int digit) {
    return univ[index][digit];
}
```
Array Element Accesses

Multidimensional array:

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

Multilevel array:

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

- **Accesses look** the same, but aren’t:
  \[ \text{Mem[sea+20*index+4*digit]} \neq \text{Mem[Mem[univ+8*index]+4*digit]} \]

- **Memory layout is different:**
  - One array declaration = one contiguous block of memory
Summary

❖ Building an executable
  ▪ Multistep process: compiling, assembling, linking
  ▪ Object code finished by linker using symbol and relocation tables to produce machine code (with finalized addresses)
  ▪ Loader sets up initial memory from executable

❖ Arrays
  ▪ Contiguous allocations of memory
  ▪ No bounds checking (and no default initialization)
  ▪ Can usually be treated like a pointer to first element
  ▪ Multidimensional → array of arrays in one contiguous block
  ▪ Multilevel → array of pointers to arrays
    • Each array/part separate in memory