CSE 374 Lecture 8

Introduction to C

C v. Java

C

- Lower level (closer to assembly)
- No guaranteed memory safety
- Procedural
- Compiled (not interpreted like bash)
- Conditional controls (if, while)
- Modern syntax (human readable)
- Small standard library

Java

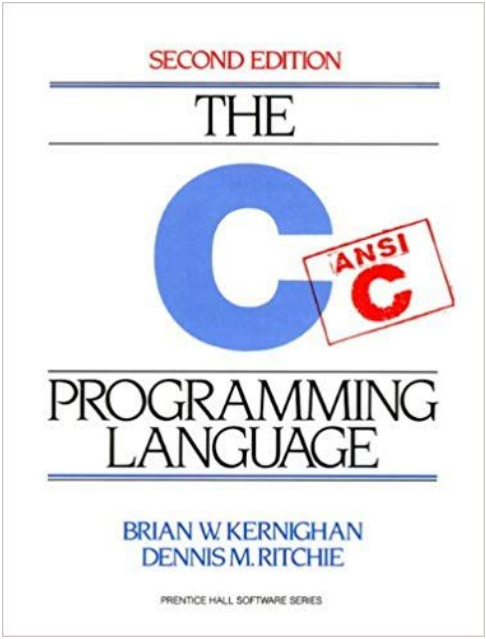
- Higher level (lots of compilation)
- Safe (sand-boxed in jvm, compiled limits)
- Object Oriented
- Compiled
- Conditional controls (if, while)
- Modern syntax (human readable)
- Large standard library, huge extended libraries

Why C?

- ➔ C is a fairly compact language - fewer features than Java, but easier to implement efficiently
- ➔ Provides lower level (closer to assembly) language
- ➔ Understanding C can give insight into how computers (and memory) work
- ➔ Still used for
  - ◆ Embedded programming
  - ◆ Systems programming
  - ◆ High-performance code
  - ◆ GPU Programming

C reference books

The standard reference. Available on Kindle and in the UW library.



Essential C - Stanford pdf
http://cslibrary.stanford.edu/101/EssentialC.pdf

http://www.cplusplus.com/

- O'Reilly books (C in a Nutshell, etc.)

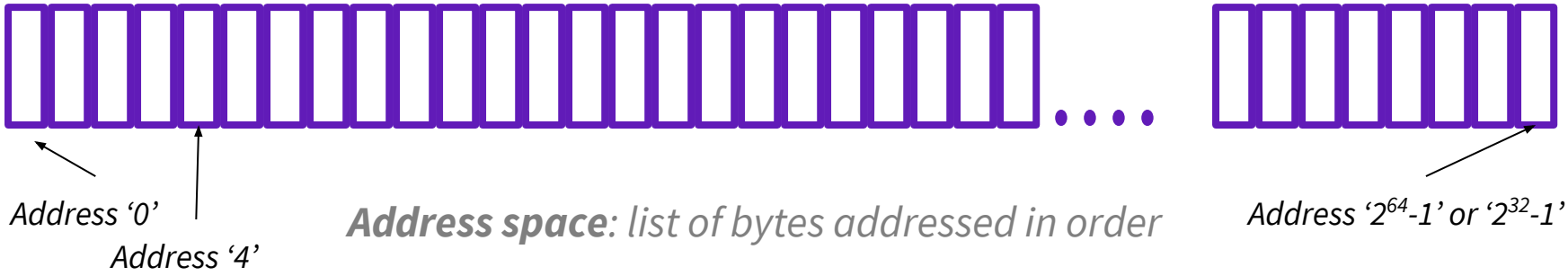
Computers & Memory

CPU - the ‘central processing unit’: computer circuitry that follows computer instructions with simple logic, arithmetic, and I/O

Hard disc storage (modernly often solid state memory instead of traditional drive): holds long-term memory which can persist across re-starts

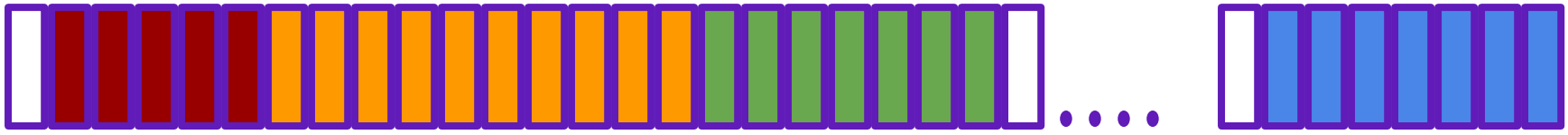
RAM (memory) : where data is stored during operation - short term memory

Working memory, cont.



- Programs are said to have access to this 2^{64} byte space
  - '64 bit' system refers to needing 64 bits to index the space
  - But really don't - many other things are also using this space
- Location in array is the 'address' of a byte
- Programs keep track of addresses of each of their pieces of memory
- Accessing unused address causes a 'segmentation fault'

Working memory, cont.



- Lowest memory stores program instructions, then global variables (static constants, string literals)
- ‘Heap’ holds dynamically allocated variables (‘new’ or ‘malloc’ variables)
- ‘Stack’ holds current instructions, each function in a frame
  - ‘Stack’ memory implies that a frame is added, and then the last frame added is removed first
- The heap and stack grow dynamically. Meet in the middle ?= ‘out of memory’ error

Pointers

“Point to memory location”

```
int x = 4;
```

Variable called ‘x’ of type ‘int’ given value of ‘4’

```
int *xPtr = &x;
```

Variable called ‘xPtr’ of type ‘pointer to an integer’, given value of the location of ‘x’

```
int xCopy = *xPtr;
```

Variable called xCopy given the value stored at the location pointed to by xPtr

```
int* noPtr = NULL;
```

Variable ‘noPtr’ correctly set when location is not yet known

Arrays

Contiguous blocks in memory

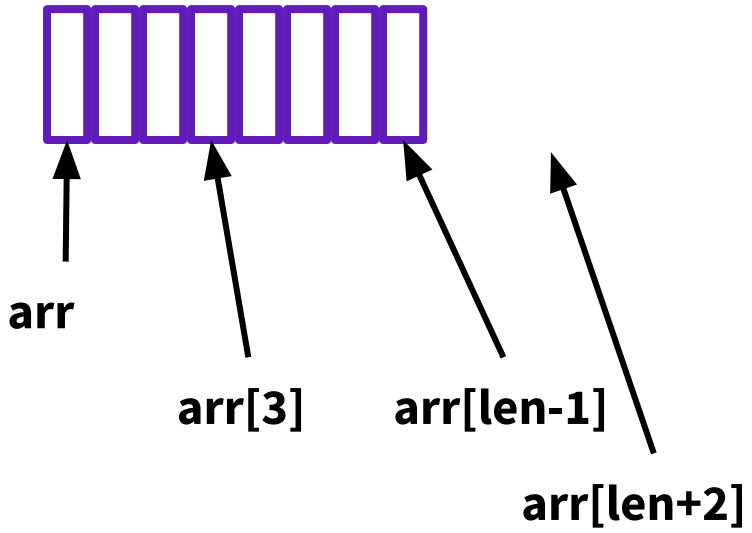
Declare as

```
Datatype arr[len]
```

Has type

```
Datatype*
```

Stores the location in memory of the first value



Danger, Will Robinson!!

Hello World in C

```
#include <stdio.h>

/**
 * Compile this file with:
 *      gcc -o hello hello.c
 */
int main(int argc, char **argv)
{
  printf("Hello, World!\n");
  return 0;
}
```

- ➔ Compile: `gcc hello.c`
  - ◆ creates executable `a.out`
- ➔ Or: `gcc -Wall -std=c11 -o hello hello.c`
  - ◆ Wall - turns all warnings on
  - ◆ C11 - specifies using C11 standard libraries
  - ◆ Creates executable `hello`
- ➔ Run: `./a.out` or `./hello`
  - ◆ Exits with ‘0’ (`return 0;`)

Hello World in C

```
#include <stdio.h>

/**
 * Compile this file with:
 *      gcc -o hello hello.c
 */
int main(int argc, char **argv)
{
  printf("Hello, World!\n");
  return 0;
}
```

- ➔ Include the stdio library (printf, stdout, etc)
- ➔ Other standard libraries
  - ◆ Stdlib, math, assert, etc
- ➔ Also include developer files
  - ◆ #include “myFile.h”

Hello World in C

```
#include <stdio.h>

/**
 * Compile this file with:
 *     gcc -o hello hello.c
 */
int main(int argc, char **argv)
{
  printf("Hello, World!\n");
  return 0;
}
```

➔ Comment block

- /* long form comments */
- // shorter comments

Hello World in C

```
#include <stdio.h>

/**
 * Compile this file with:
 *      gcc -o hello hello.c
 */
int main(int argc, char **argv)
{
  printf("Hello, World!\n");
  return 0;
}
```

- ➔ C functions look a lot like Java methods.
  - ◆ Have return type, arguments
  - ◆ Code block set off with ‘{‘ and ‘}’
- ➔ Program runs through ‘main’
  - ◆ But not part of class!!
- ➔ Return value - program exit
  - ◆ >> echo “$?”

What is char **argv ??

- Char - datatype
- char* - pointer to a place in memory that stores a char
- char** - pointer to a place in memory that stores pointers to chars
- The variables `argv` hold `argc` points to char* ptrs
  - In c array lengths must be sent as separate arguments, as is done here
- Also access values with `argv[0]`, `argv[1]`, ... `argv[argc-1]`

Okay, so, argv[i] ?

- Any argv[i] points to a char* (pointer to characters)
- char* - pointer to a place in memory that stores a char or multiple chars
- If char* points to an array of characters ending in \0 (a zero byte)
- Aka a string!!
- Argv are usually has arguments coded into strings

“Hello, World!\n”

Is a string of length 15 (\n is one character, but contains \0)

In this case, is a ‘string literal’ - evaluates to a global, immutable array.

“printf”

Prints to stdout, which is defined in stdio.h

Strings

No real strings - just arrays of characters.

```
[ "h", "e", "l", "l", "o", " ", "w", "o", "r", "l", "d", "!", \0 ]
```

Strings terminate with \0 so their length can be determined

```
char str[] = "hello";  // array syntax
char *str2 = "hello";  // pointer syntax
char *arrStr[] = {"ant", "bee"}; // array containing char*'s
char **arrStrPtr = arrStr;  // pointer to an array containing char*'s
arrStr[0] = "cat";
```