Introduction to C (Part 2)

Devin J. Pohly <djpohly@cse.psu.edu>
(adapted from original slides by Steve Gribble at UWash)

Differences from C++/Java

- Syntax is very similar
- But C gives you more control
  - Or "does less for you"
    - Manual vs. automatic
  - Do-it-yourself approach
    - Keep track of array bounds, memory used, files opened, etc.
    - No exceptions or objects
    - Be consistent and disciplined!

Arrays

- Array implementation
  - Just a raw, contiguous block of memory of the correct size
  - Array of 6 `int32_t` requires 6×4 bytes = 24 bytes of memory
- Arrays have no methods, do not know their own length (DIY)
  - C doesn't stop you from overstepping the end of an array!!
  - Most common security bugs come from this

```
int x[6];
```

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
34	11	-129	49	708	-11

Arrays

- In this array, `x[7] = 45;` is completely legal!
  - But it's a bad idea: may write to invalid memory or worse!
    - Same for `x[6] = 45;` – remember the first index is 0
  - Whenever you use an array, you should have either a constant or a variable to keep track of the length.

```
int x[6];
```

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
34	11	-129	49	708	-11

Strings

- Strings are simply arrays of char
  - "Null-terminated": end is marked by the null character `'\0'`
    - Remember: this will take up an extra byte!
  - Not objects, don't have methods
  - Helpful utilities found in the `<string.h>` header file


```
char *x = "hello\n";
```

Exceptions

- C does not use exceptions (try/catch)
- Errors are indicated by returning an integer error code (DIY)
  - Zero usually means success
  - Common idiom:

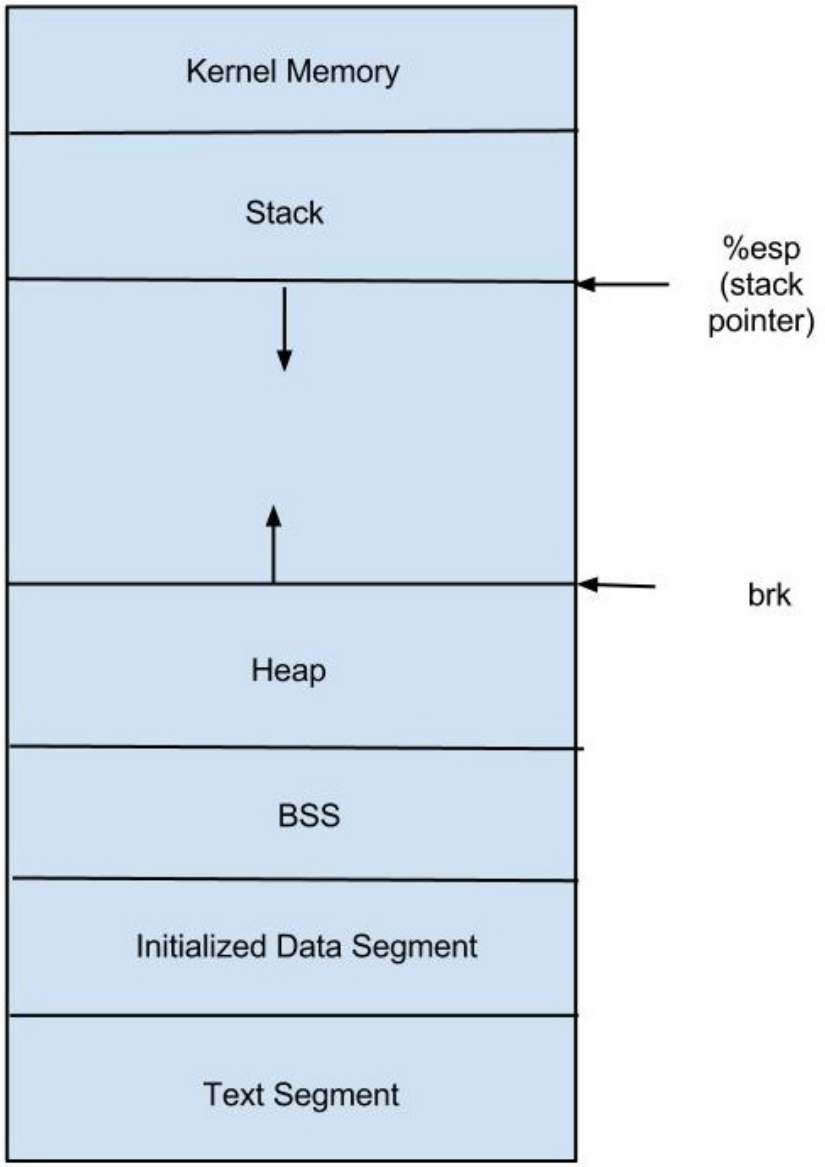
```
int ret = try_something(arg);
if (ret) {
  // print an error, clean up
  return 1;
}
```

Objects

- No objects
  - No constructors or destructors
  - No instance methods
- Similar functionality can be implemented with C features (DIY)
  - Fields → struct
  - Methods → functions
  - Inheritance → pointers
  - Libraries and the Linux kernel do a *lot* of this

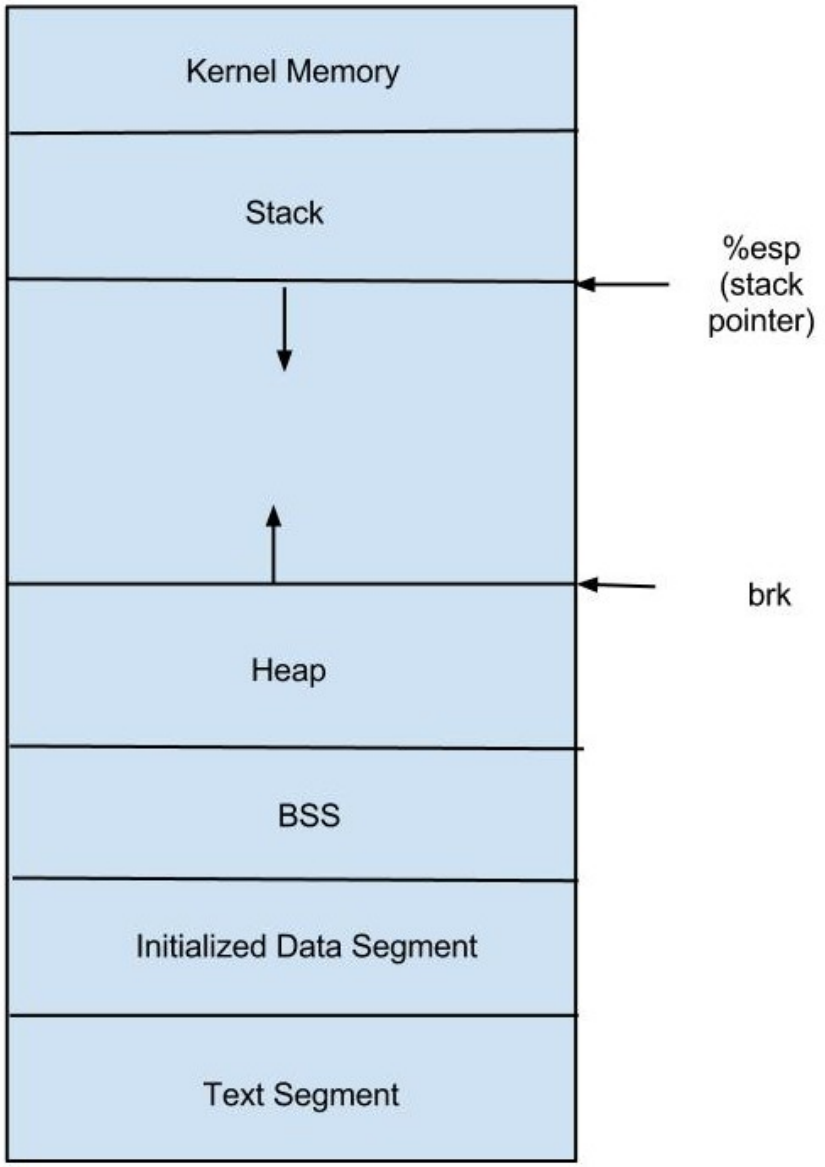
Memory management

- Similarities
  - Local (automatic) variables are allocated on the stack
    - Automatically freed when you return from the function
  - Global and static variables are allocated in a data segment
    - Freed by the OS when your program exits



Memory management

- Differences
  - No garbage collector
    - Anything you allocate you have to free (DIY)
  - Allocate memory from the heap using malloc()
    - When you're done, free the memory with free()
    - Failing to free is a memory leak
    - Double-freeing is an error (hopefully it crashes)

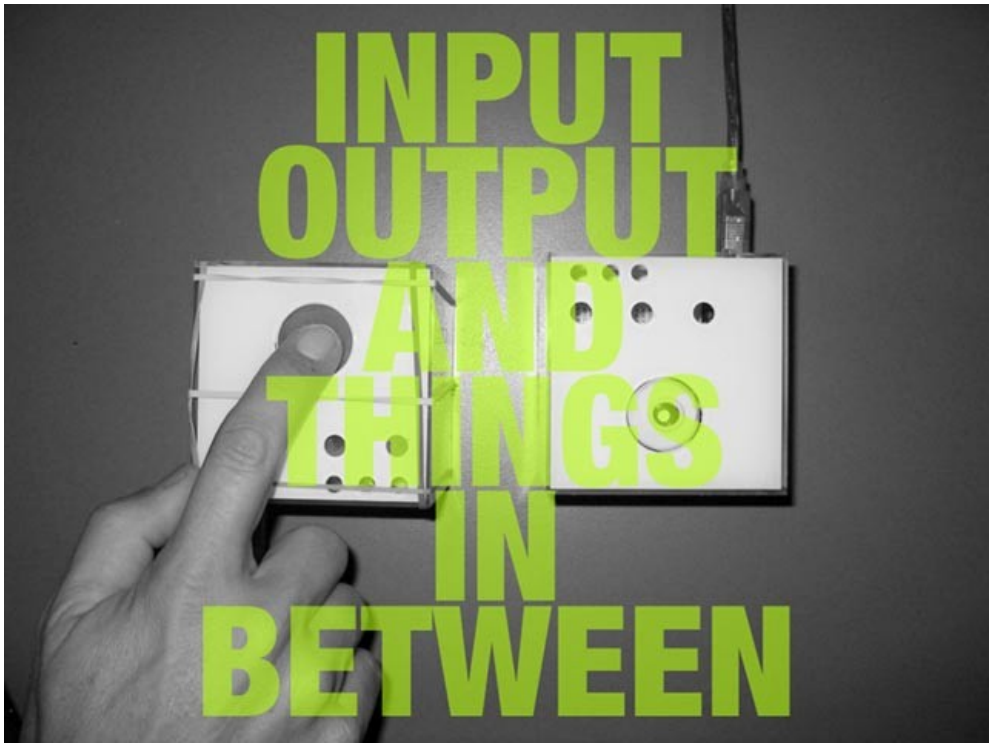


Crashes

- Same as C++, different from Java
- Major bugs often try to write to memory that isn't yours.
  - "Segmentation fault" and crash
  - Debugger (if any) takes over
- Many hard-to-find bugs don't crash
  - So if you get a crash, you are one of the lucky ones!

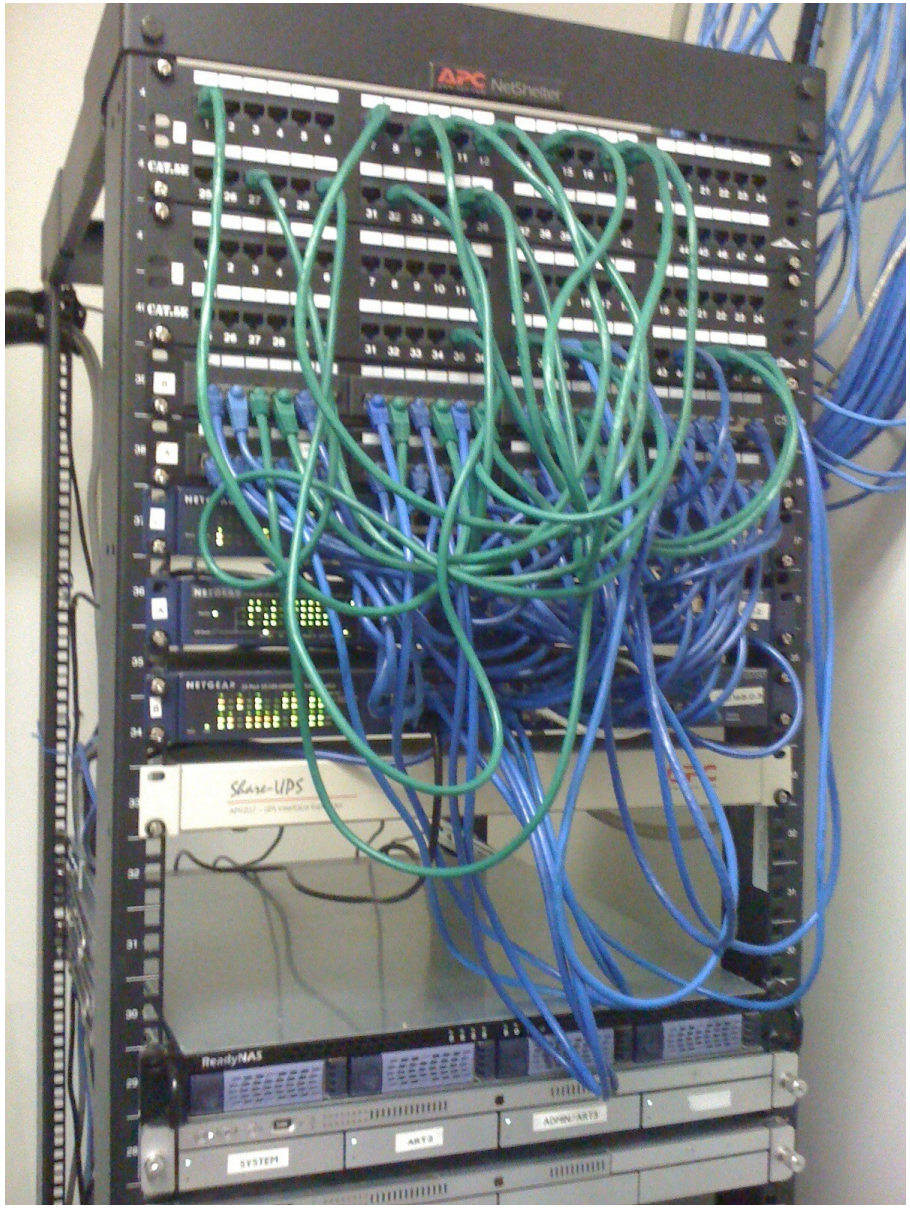
I/O

- Standard (console) I/O
  - C library has portable functions
    - scanf(), printf()
- File I/O
  - C library has portable functions
    - fopen(), fread(), fwrite(), fclose()
    - Buffered by default, blocking by default
  - POSIX defines low-level functions
    - open(), read(), write(), close()
    - We'll be using these: more control over buffering and blocking



I/O

- Network I/O
  - C standard library has no notion of network I/O
    - POSIX specifies standard functions, provided by most OSes
  - Lots of complexity here
    - Errors: network can fail
    - Performance: network can be slow or variable
    - Concurrency: servers speak to thousands of clients simultaneously



Libraries

- Fewer *built-in* libraries
  - No trees, hashtables, linked lists, etc.
  - You may have to DIY
    - One major reason why some people think C is a less productive language.
- Many *available* libraries
  - GLib, SDL, zlib, libpng, …
  - Someone already DIYed for the cause

Defining a function

```
returnType name(type name, …, type name) {
    statements;
}
```

```
// sum integers from 1 to max
int sumTo(int max) {
  int i, sum = 0;

  for (i=1; i<=max; i++) {
    sum += i;
  }
  return sum;
}
```

Problem: ordering

- You can't call a function that hasn't been declared yet.
  - Why?


```c
#include <stdio.h>

int main(int argc, char **argv) {
  printf("sumTo(5) is: %d\n", sumTo(5));
  return 0;
}

// sum integers from 1 to max
int sumTo(int max) {
  int i, sum = 0;

  for (i=1; i<=max; i++) {
    sum += i;
  }
  return sum;
}
```

Problem: ordering

- One solution: reverse order of definition

```c
#include <stdio.h>

// sum integers from 1 to max
int sumTo(int max) {
  int i, sum = 0;

  for (i=1; i<=max; i++) {
    sum += i;
  }
  return sum;
}

int main(int argc, char **argv) {
  printf("sumTo(5) is: %d\n", sumTo(5));
  return 0;
}
```

Problem: ordering

- Another solution: provide a declaration of the function
  - teaches the compiler the argument and return types of the function
- Often split out into a header (.h) file

```c
#include <stdio.h>

// this function prototype is a
// declaration of sumTo
int sumTo(int);

int main(int argc, char **argv) {
  printf("sumTo(5) is: %d\n", sumTo(5));
  return 0;
}

// sum integers from 1 to max
int sumTo(int max) {
  int i, sum = 0;

  for (i=1; i<=max; i++) {
    sum += i;
  }
  return sum;
}
```

Unix default files (std*)

- Unix gives every process three default files
  - Standard input (stdin)
    - Default: keyboard/terminal
  - Standard output (stdout)
    - Default: screen/terminal
  - Standard error (stderr)
    - Default: screen/terminal
- Most Unix utilities use these automatically
  - Filter from stdin to stdout

Unix environment variables

- Environment variables are variables you can set from the shell
  - Used to set up the environment for programs (we will see this more later)
  - Set with `export VARNAME=value`
  - `$VARNAME` to use the value
    - `echo $VARNAME` to print
  - Use `printenv` to show all environment variables

Running a program

- You can specify the path when running a program

```
$ ./hello
$ /usr/bin/firefox
```

- Otherwise, Unix looks in specific directories
  - Listed by the PATH environment variable
  - To add to the search path, just add more colon-separated paths:

```
export PATH=$PATH:/new/path
```