File I/O, Project 1: List ADT

Bryce Boe

2013/07/02

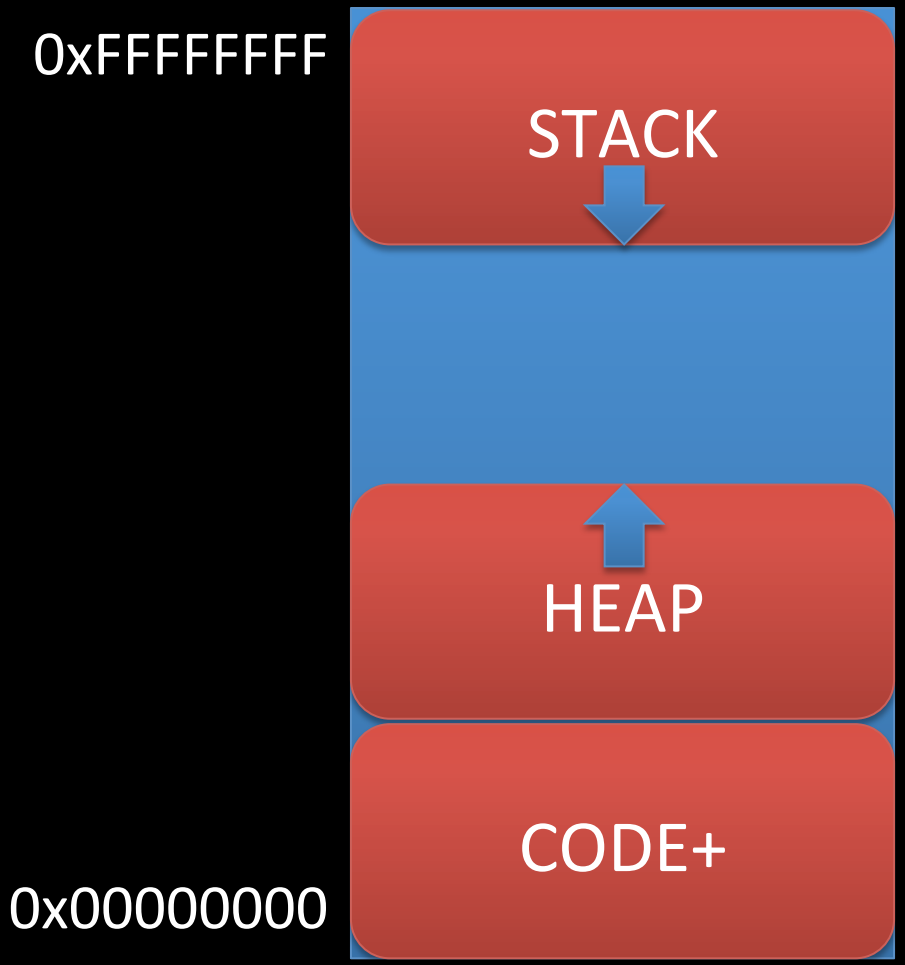
CS24, Summer 2013 C

Outline

- Memory Layout Review
- Pointers and Arrays Example
- File I/O
- Project 1 – List ADT

MEMORY LAYOUT REVIEW

Simplified process's address space



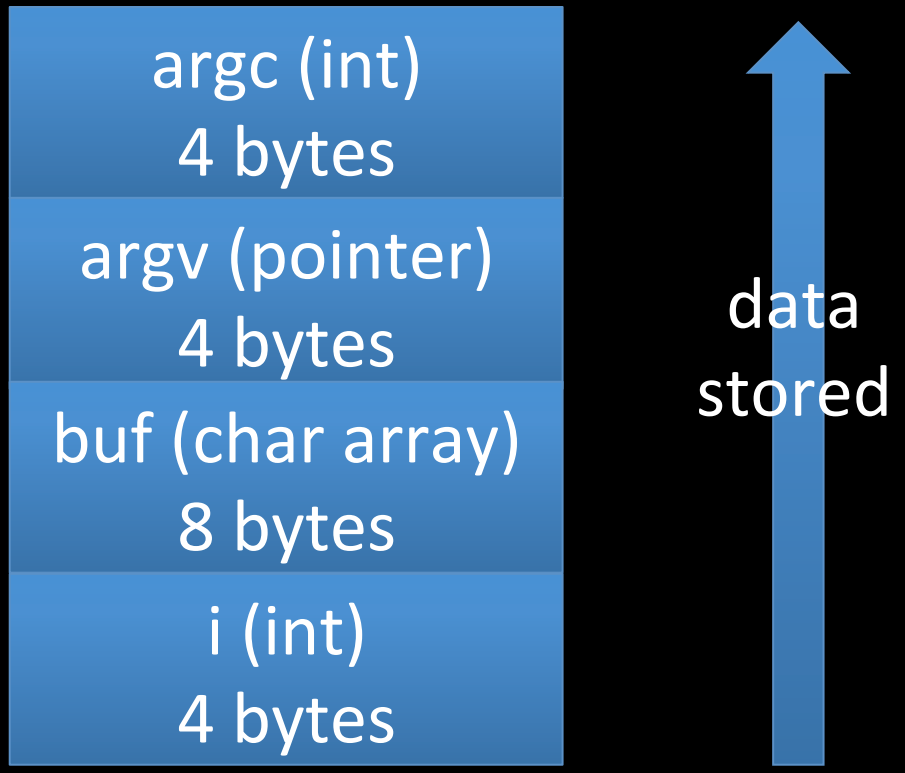
How did we get here?

- func_foo calls func_blah
- func_blah calls func_bar

What two things does the (simplified) activation record store?

- Function parameters
- Function local variables

main's simplified activation record

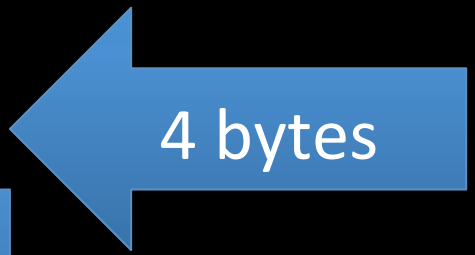


Total: 20 bytes

How many bytes is the simplified activation record?

```
struct Point {
        int x, y;
        char *name;
};
int some_func(void *something) {
        struct Point point;
        ...
}
```

TOTAL: 52 bytes



How many bytes is the simplified activation record?

```
struct Point {
        int x, y;
        char *name;
};
int some_func(void *something) {
        struct Point *points;
        …
}
```

Think about it

```
int main() {
        char msg[] = “hello”;
        char buf[] = “1234567”;
        char msg2[] = “world”;
}
```

- What value does buf[8] hold?
- What about msg[7]?

See (pointer_v_array.c)

POINTERS AND ARRAY EXAMPLE

FILE I/O

File I/O

- I/O stands for input/output
- Provided by the stdio library (stdio.h)
- Allows reading and writing to/from streams (type: FILE *)
  - stdin (read only)
  - stdout / stderr (write only)
  - named files

stdio.h

- Explore via opengroup: http://pubs.opengroup.org/onlinepubs/009695399/basedefs/stdio.h.html
- Explore via `man stdio.h` (same info as opengroup, but not easily searchable or linkable)

Opening named files and closing streams

- FILE* fopen(char *filename, char *mode)
  - open a file specifying whether to open for reading, writing, appending, and create or truncate if desired
- int fclose(FILE *stream) – close an open FILE*
  - Flush and close the stream
  - Return 0 on success

Reading from streams

- **int** fgetc(FILE *stream)
  - Returns the next character in the stream or EOF (this is why it returns an int)
- char *fgets(char *buf, int n, FILE *stream)
  - Read at most n-1 bytes from stream into buf
  - Also stops when ‘\n’ or EOF is reached
  - Terminates buf with the null character ‘\0’

Other read functions

- fread – very useful (especially when input is unbounded), we won't use in this class
- fscanf – useful, but tricky, don't use in this class
- Functions with similar names less the 'f'
  - Uses the **stdin** stream thus doesn't require the *stream* argument

Writing to streams

- int fputc(**int** c, FILE *stream)
  - Write a single character **c** to **stream**
- int fputs(char *buf, FILE *stream)
  - Write the null-terminated string in **buf** to **stream**
- int fprintf(FILE *stream, char *format, …)
  - Write formatted string to stream making the specified replacements

SECURITY WARNING

- **NEVER** do the following:

fprintf(stdout, buf); // buf is some c-string

- Could allow an attacker to inspect and change your program (format string exploit)
- Use either *fputs* or *fprintf(stdout, "%s", buf)*
- <See bad_format_string.c>

Other write functions

- fwrite – generally very useful, we won't use in this class
- Functions with similar names less the 'f'
  - Uses the **stdout** stream thus doesn't require the *stream* argument

Other useful stream functions

- int feof(FILE *stream)
  - Return non-zero if the stream has reached the end of the file
- int fflush(FILE *stream)
  - Force writing any buffered data to the stream
  - Flushing *typically* occurs when a newline is encountered, thus fflush is often needed when newlines aren't used

I/O Questions

- Why does fgetc/fputc return/get an integer?
- If a file with only a single newline at the end is 32 bytes long, how many bytes does the buffer for fgets require to read the entire file?

More I/O Questions

- When using **fgets**, how can you determine if the string is longer than the value of ‘n’ (the number of bytes to read)
- What will happen if the ‘n’ parameter to **fgets** is larger than the buffer?

Real-time cat program writing

- <In class creation of copy.c>
  - We'll finish this next Tuesday

PROJECT 1: LIST ADT

Abstract Data Types

- A container for data
  - Container provides a set of operations
  - Abstract in that the *costumer* does not need to concern themselves with the implementation

Project 1 Purpose

- Implement the List ADT using two distinct storage models
- Understand the tradeoffs between the two implementations

List Operations

- Consult the project 1 description for the list of methods to implement

For Next Tuesday

- Finish reading chapter 1 in the text book (if you haven't already)
- Begin reading chapter 3 (might want skim/ read chapter 2) as it's helpful for project 1
  - Note the book uses C++ so (for now) think about how to do similar in C