CS107 Winter 2020, Lecture 6
More Pointers and Arrays

reading:
*K&R (5.2-5.5) or Essential C section 6*
CS107 Topic 3: How can we effectively manage all types of memory in our programs?
Plan For Today

- Pointers and addresses, a review
- Arrays of different types
- **Break**: Announcements
- Programming with stack memory

```bash
cp -r /afs/ir/class/cs107/samples/lectures/lect6 .
```
### Warmup exercise

Compare and contrast the following declarations of `x` (some may not compile):

1. `char *x;`  
   `char x[4];`

2. `char *x;`  
   `char &x;`

3. `char *x;`  
   `char **x;`

4. `char *x;`  
   `char* x;`
A most unfortunate page break

• K&R, top of p. 100

```c
char s[];
and
char *s;
```

are equivalent; we prefer the latter because it says more explicitly that the parameter is a pointer. When an array name is passed to a function, the function can at its convenience believe that it has been handed either an array or a pointer, and manipulate it accordingly. It can even use both notations if it seems appropriate and clear.

• K&R, bottom of p. 99

```c
strlen(ptr); /* char *ptr; */
```

⚠ Arrays are **NOT** pointers, even if they sometimes behave like them.
In variable **declaration**, * creates a **pointer**.

```c
char ch = 'r';  // ch stores a char
char *cptr = &ch;  // cptr stores an address of a char (points to a char)
char **strptr = &cptr;  // strptr stores an address of a char * (points to a char *)
```
Warmup exercise

Compare and contrast the following declarations of x (some may not compile):

1. `char *x;`  
   Pointer variable: stores an address to a char
   `char x[4];`  
   Array variable: contiguous block of memory on stack

2. `char *x;`  
   `char &x;`  
   Not a valid type; & operator means “address of”

3. `char *x;`  
   Variable stores an address of a char
   `char **x;`  
   Variable stores an address of a char *

4. `char *x;`  
   No difference; left is preferred stylistically
   `char* x;`
In reading values from/storing values, * dereferences a pointer.

```c
char ch = 'r';
ch = ch + 1;

char *cptr = &ch;

char **strptr = &cptr;
```

Increment value stored in `ch`

```
ch
```

```
's'
```

```
0xe8
```

```
0xe0
```

```
0xf0
```
In **reading values from/storing values**, *dereferences* a pointer.

```c
char ch = 'r';
ch = ch + 1;
char *cptr = &ch;
*cptr = *cptr + 1;
char **strptr = &cptr;
```

- Increment value stored in `ch`
- Increment value stored at memory address in `cptr`
- (increment char *pointed to*)

---

- `ch` is referenced as `0xf0`
- `cptr` is referenced as `0xe8`
- `strptr` is referenced as `0xe0`

---

Review
In reading values from/storing values, * dereferences a pointer.

```c
char ch = 'r';
ch = ch + 1;

char *cptr = &ch;
*cptr = *cptr + 1;

char **strptr = &cptr;
*strptr = *strptr + 1;
```

Increment value stored in `ch`

Increment value stored at memory address in `cptr` (increment char pointed to)

Increment value stored at memory address in `cptr` (increment address pointed to)
Pen and paper: A * Wars Story

void binky() {
    int a = 10;
    int b = 20;
    int *p = &a;
    int *q = &b;
    *p = *q;
    p = q;
}
```c
void binky() {
    int a = 10;
    int b = 20;
    int *p = &a;
    int *q = &b;
    *p = *q;
    p = q;
}
```

- Line 7: Update your diagram.
- Line 8: Update your diagram.

![Diagram showing memory addresses and variables a, b, p, and q]
```c
void binky() {
    int a = 10;
    int b = 20;
    int *p = &a;
    int *q = &b;
    *p = *q;
    p = q;
}
```

- Line 7: Update your diagram.
- Line 8: Update your diagram.
Beyond char* and char[]

Recall: `sizeof()` returns the size of a variable type in `bytes` at compile time.

\[
\text{sizeof(char *) // 8}
\]

What will `sizeof(arr)` return in the following cases?

1. char arr[] = "rey";
   
   A. 4     D. 24
   B. 8     E. 32
   C. 16    F. Other

2. int arr[] = {1, 32, 128, 256};

3. char *arr[] = {
   "have a",
   "nice",
   "day"
   };

3. How will this affect the size of `sizeof(arr)`?
   
   A. 4     D. 24
   B. 8     E. 32
   C. 16    F. Other

4. Will the size of `sizeof(arr)` change if we change the type of `arr` to `unsigned int`?
   
   A. Yes
   B. No
Array indexing is “syntactic sugar” for pointer arithmetic:

\[ \text{ptr} + i \Leftrightarrow \&\text{ptr}[i] \]
\[ *(\text{ptr} + i) \Leftrightarrow \text{ptr}[i] \]

⚠ Pointer arithmetic does not work in bytes; it works on the type it points to. On int* addresses scale by sizeof(int), on char* scale by sizeof(char).

• This means too-large/negative subscripts will compile 😊
  \[ \text{arr}[99] \quad \text{arr}[-1] \]

• You can use either syntax on either pointer or array.
Demo: Pointer arithmetic

arrptr.c
Useful Linux commands

Working with multiple terminals:
- Create new terminal, click and drag to resize, then ssh into both
- Useful to keep gdb open in one window, editor in another
- ⚠ Don’t edit the same file in two different windows

Working with one terminal:
- clear (or Ctrl-k/⌘-k) clears current terminal output
- Ctrl-z temporarily suspends current job (editor, gdb, etc.)
- jobs lists all jobs
- fg %1 continues job [1] (see jobs for job list)
- kill %1 kills job (“force quit” – only use if necessary)

Whatever your workflow is, be sure to quit all jobs (“close all windows”) when you log out/disconnect internet
Some notes:
- The `size_t` (unsigned long) represents size of objects in bytes.
- Line 4: Recall the character '\0' has ASCII value 0.
This function exclusively uses pointer arithmetic.
- Line 4: `++` operator on pointers needs to know that `s` points to chars!
- Line 5: `-` operator on pointers needs to know both `s` and `str` point to chars!
/* Using array indexing */
size_t my_strlen(char *str) {
    int n = 0;
    while (___?___) {
        n++;
    }
    return n;
}

Line 4: Which loop condition?
A. str[n], or equivalently, str[n] != '\0'
B. str + n != ""
C. str + n != NULL
D. Something else

Bonus: Can we write a strlen function for other array types (e.g., int arrays)?
strstr for other array types?

- **No!** Arrays in general are not null-terminated.
- For strings, C designer Dennis Ritchie decided to use a special terminating character ("\0"). Other array types don’t have this (e.g., what’s a “null-terminating integer”?)

When writing functions for other array types, you need to explicitly pass in the length of an array:

```c
void traverse_int_array(int arr[], int len);
```

- Note: assign2’s `const char *envp[]` has a NULL pointer placed after its last entry, but this is extremely uncommon.
Questions?
Plan For Today

• Pointers and addresses, a review
• Arrays of different types
• Double pointers, a review
• Programming with stack memory
• Announcements
• Other topics: const, struct and ternary

```bash
cp -r /afs/ir/class/cs107/samples/lectures/lect6 .
```
C parameters are pass-by-value

A parameter is the callee’s variable—distinct from caller’s variable. In all cases (other than array-as-parameter case), values are copied.

**callee**
```c
void change(char ch) {
    ch = toupper(ch);
}
```

**caller**
```c
int main(...) {
    char letter = 'a';
    change(letter);
    return 0;
}
```

To achieve pass-by-reference, manually use pointers.

- letter is **unchanged**
- letter is **changed**
The same applies to pointer parameters

**caller**

```c
void change(char *str) {
    str = str + 1;
}
```

```c
int main(...) {
    char *name = "drain";
    change(name);
    return 0;
}
```

*name is **unchanged***

**callee**

```c
void change(char **p_str) {
    *p_str = *p_str + 1;
}
```

```c
int main(...) {
    char *name = "drain";
    change(&name);
    return 0;
}
```

*name is **changed***

**Tip: Always draw a picture!!**
void skip_spaces(char **p_str) {
    int num = strspn(*p_str, " ");
    *p_str = *p_str + num;
}

What diagram most accurately depicts program state at Line 4 (before skip_spaces returns to main)?
What diagram most accurately depicts program state at Line 4 (before `skip_spaces` returns to `main`)?

A. 

B. C++ pass by reference

C.
Plan For Today

• Pointers and addresses, a review
• Arrays of different types
• Double pointers, a review

• Break: Announcements
• Programming with stack memory
• lab2 check-in
• assign2: lots of string practice, some double pointers
• Read Piazza FAQ for each assignment

• Social Good Fellowship: https://web.stanford.edu/class/cs107/resources/cs- sg-fellowship.pdf
• CS198: deadline Thursday 1/30, 11:59PM cs198.stanford.edu
Joke break

What data type do you need to store this number?

INT_MAX: 2147483647
UINT_MAX: 4294967295
FLOAT_MAX: 3.40282e+38

(we’ll come back to this in 2 weeks)
Plan For Today

• Pointers and addresses, a review
• Arrays of different types
• Double pointers, a review
• Break: Announcements
• Programming with stack memory

cp -r /afs/ir/class/cs107/samples/lectures/lect6 .
Memory Layout**

- The **stack** is the place where all local variables and parameters live for each function.*
- A function’s **stack frame** goes away when the function returns.
- The stack grows **downwards** when a new function is called. It shrinks **upwards** when the function is finished.

*gcc, our compiler, uses option `-O0` to force everything (including parameters) on the stack (see week 6, “registers,” for more info)

**Assume all 18 exabytes of system memory accessible (read about “virtual memory” for more info)
The Stack

• The stack behaves like a...well...stack! A new function call **pushes** on a new frame. A completed function call **pops** off the most recent frame.

• A *stack overflow* is when you use up all stack memory. E.g. a recursive call with too many function calls.

⚠ *Interesting and important fact:* C does not clear out memory when a function’s frame is removed. Instead, it just marks that memory as usable for the next function call. This is more efficient!
The Stack

```c
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

`stack_chaos.c`
The Stack

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

stack_chaos.c
The Stack

```c
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

stack_chaos.c
The Stack

1 char *create_string(char ch, int num) {
2     char new_str[num + 1];
3     for (int i = 0; i < num; i++) {
4         new_str[i] = ch;
5     }
6     new_str[num] = '\0';
7     return new_str;
8 }

1 int main(int argc, char *argv[]) {
2     char *str = create_string('a', 4);
3     printf("%s", str); // want "aaaa"
4     return 0;
5 }

stack_chaos.c
The Stack

```c
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Returns e.g. 0xff50

---

Memory

```
main
argc: 1  str: [ ]
argv: 0xffff0

create_string
ch: 'a'  num: 4

\0'
'a'
'a'
'a'
'a'

new_str: 'a'
```

0x0

stack_chaos.c
```c
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Stack:
- `main`
- `argv` at `0xffff0`
- `create_string`
  - `ch: 'a'`
  - `num: 4`
- `new_str: 'a'`
- `str` at `0xff50`

Memory:
- `main:
  - argc: 1
  - str: 0xff50`
- `argv: 0xffff0`
The Stack

```c
1 char *create_string(char ch, int num) {
2     char new_str[num + 1];
3     for (int i = 0; i < num; i++) {
4         new_str[i] = ch;
5     }
6     new_str[num] = '\0';
7     return new_str;
8 }

1 int main(int argc, char *argv[]) {
2     char *str = create_string('a', 4);
3     printf("%s", str);  // want "aaaa"
4     return 0;
5 }
```

stack_chaos.c
The Stack

```c
1 char *create_string(char ch, int num) {
2     char new_str[num + 1];
3     for (int i = 0; i < num; i++) {
4         new_str[i] = ch;
5     }
6     new_str[num] = '\0';
7     return new_str;
8 }
9
10 int main(int argc, char *argv[]) {
11     char *str = create_string('a', 4);
12     printf("%s", str);  // want "aaaa"
13     return 0;
14 }
```

⚠ Stack memory gets reused when a function returns!

stack_chaos.c
Demo: Stack chaos

stack_chaos.c
Overflow
(for next time)
How do we make sure memory content persists after a function exits?

If we’re committed to stack-only memory: let the caller allocate memory and pass it to the callee.

(we’ll learn heap memory next week)
Recall that arrays passed in as parameters are converted to pointers.

Strategy #1: Allow callee to modify existing arrays.

```c
void binky(char *ptr) {
    ptr[0] = 'B';
}

int main(int argc, char *argv[]) {
    char arr[] = "asdf";
    modify_array(arr);
    printf("%s", arr);  // Bsdf
    return 0;
}
```
Recall that arrays passed in as parameters are converted to pointers.

Strategy #2: Pass in a pre-allocated buffer that the caller can write to.

```c
void pig_latin(char *out, char *in) {
    /* write changes to out */
}

int main(int argc, char *argv[]) {
    char str[] = "be";
    char converted[50]; // output buffer
    pig_latin(converted, str);
    printf("%s %s", str, converted); // be ebay
    return 0;
}
```
Write a function copy_pos that copies positive integers from in to out and returns the number of integers copied (i.e., the length of out).

arr

| -2 | 1 | 0 | 4 | 6 | -8 |

buf

| 1 | 4 | 6 | ? | ? | ? |

```
int main(int argc, char *argv[]) {
    int arr[] = {-2, 1, 0, 4, 6, -8};
    int len = ?; // 6
    int buf[len];
    int buf_len = copy_pos(buf, arr, len); // returns 3
    return 0;
}
```
Write a function `copy_pos` that copies positive integers from `in` to `out` and returns the number of integers copied (i.e., the length of `out`).

```c
int copy_pos(int *out, int *in, int len) {
    int num = 0;
    for (int i = 0; i < len; i++) {
        /* fill this part in */
    }
    return num;
}
```

```c
int main(int argc, char *argv[]) {
    int arr[] = {-2, 1, 0, 4, 6, -8};
    int len = ____?______;
    int buf[len];
    int buf_len = copy_pos(buf, arr, len);
    return 0;
}
```
Practice: Copy positive integers

(int_array.c)

(lecture: with array indexing)
Bonus: Try writing this with pointer arithmetic!