

CS 241

Section Week #1

About Sections

- Each week:
 - We'll spend additional time on topics that the instructors feel should be reviewed.
 - We'll prepare you for the upcoming homework or MP submissions.
 - We'll provide extra review/guidance for upcoming exams.

Topics This Section

- Reading and Writing to the Console
- Memory
- Precedence
- Casting
- Strings

Console I/O

- In lecture, you saw the `printf()` command.
 - `printf("%s: %d", str, i);`
 - `printf("%c%c%c", c1, c2, c3);`
 - ...
- In C I/O, you will provide a format string with a parameter list of values to populate the string with.

Console I/O

- In lecture, you saw the printf() command.
 - `printf("%s: %d", str, i);`
 - `printf("%c%c%c", c1, c2, c3);`
 - ...
- The embedded format tags tell C how to format the variables you provide.

Console I/O

- The printf() man page describes all the different types of specifies you can use.
- Common specifies:
 - `%c` A single character
 - `%d` An integer value
 - `%f` A floating point value
 - `%s` A string
 - `%p` A pointer

Console I/O

- Example #1:


```
char *s = "the cat and the hat";  
printf("%s", s);  
printf("%c", s);
```

Console I/O

- Example #1:


```
char *s = "the cat and the hat";  
printf("%s", s);      the cat and the hat  
printf("%c", s);      t
```

Console I/O

- Example #1:

```
char *s = "the cat and the hat";  
printf("%s", s);  
printf("%c", s);
```

the cat and the hat

Why?

Console I/O

- Example #2:

```
int i = 42;  
printf("%d", i);  
printf("%c", i);
```

Console I/O

- Example #2:

```
int i = 42;  
printf("%d", i);  
printf("%c", i);
```

42

*

Console I/O

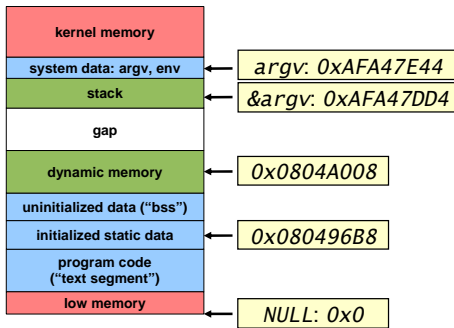
- Example #2:

```
int i = 42;  
printf("%d", i);  
printf("%c", i);
```

42

Why?

Process memory layout



Pointer arithmetic

```
char a[8];    /* array of bytes */
... [array diagram] ...
char *p = a;   /* p, a: 0xAF99EFDC */
... [array diagram] ...
char *q = a+3; /* q: 0xAF99EFDf */
... [array diagram] ...
```

The diagrams show memory arrays where each cell represents a byte. Arrows indicate pointer positions: `p` points to the first byte, and `q` points to the fourth byte (index 3).

Pointer arithmetic (2)

```
char a[8];    /* array of bytes */
... [array diagram] ...
char *q = a+3; /* q: 0xAF99EFDf */
... [array diagram] ...
char *r = &a[3]; /* r: 0xAF99EFDf */
... [array diagram] ...
```

The diagrams show memory arrays where each cell represents a byte. Arrows indicate pointer positions: `q` points to the fourth byte (index 3), and `r` points to the address of the third element (index 3).

Pointer arithmetic (3)

```
int b[2];    /* array of 4-byte words */
... [array diagram] ...
int *q = b+1; /* q: 0xAF99EFE0 */
... [array diagram] ...
char *r = &b[1]; /* r: 0xAF99EFE0 */
... [array diagram] ...
```

The diagrams show memory arrays where each cell represents a 4-byte word. Arrows indicate pointer positions: `q` points to the second word (index 1), and `r` points to the address of the second word (index 1).

Memory

- Three main categories of memory that we'll concern ourselves with in CS 241:

- **Static Memory:**

- Memory that is declared with the 'static' keyword.
- Memory is only allocated once.
- Memory is always of fixed size.
- Memory is never freed.

Memory

- Three main categories of memory that we'll concern ourselves with in CS 241:

- **Heap Allocated Memory:**

- Memory that is allocated with memory-allocating functions.
 - malloc(), calloc(), etc
- Allocated only when the memory-allocating function is called.
- Freed only when free() is called.

Memory


- Three main categories of memory that we'll concern ourselves with in CS 241:

- **Stack Allocated Memory:**

- Memory that is allocated within the scope of a function.
- Stores local variables and function parameters
- Allocated when the function begins execution.
- Freed when the function finishes execution.
- The stack memory associated with a given function is referred to as a "stack frame".

Memory

- Code Execution:



```
void foo(int myInt)
{
    int *x = (int *)malloc(sizeof(int));
    free(x);
}
```

- Memory:



Memory

- Code Execution:

```
void foo(int myInt)
{
    int *x = (int *)malloc(sizeof(int));
    free(x);
}
```

- Memory:

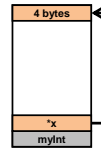


Memory

- Code Execution:

```
void foo(int myInt)
{
    int *x = (int *)malloc(sizeof(int));
    free(x);
}
```

- Memory:



Memory

- Code Execution:

```
void foo(int myInt)
{
    int *x = (int *)malloc(sizeof(int));
    free(x);
}
```

- Memory:



Memory

- Code Execution:

```
void foo(int myInt)
{
    int *x = (int *)malloc(sizeof(int));
    free(x);
}
```

- Memory:



Memory

- Code Execution:
... start with dog();

- Memory:



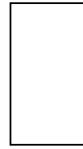
```
char *cat(char *x, int i)
{
    static int r = 4;
    x[3] = 'h';
    char *result = (char *)malloc(20);
    sprintf(result, "%s x %d", x, i);
    return result;
}

void dog()
{
    char s[] = "my cat";
    int z = 12;
    cat(s, z);
}
```

Memory

- Code Execution:

- Memory:



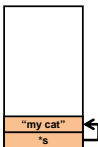
```
char *cat(char *x, int i)
{
    static int r = 4;
    x[3] = 'h';
    char *result = (char *)malloc(20);
    sprintf(result, "%s x %d", x, i);
    return result;
}

void dog()
{
    char s[] = "my cat";
    int z = 12;
    cat(s, z);
}
```

Memory

- Code Execution:

- Memory:



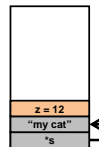
```
char *cat(char *x, int i)
{
    static int r = 4;
    x[3] = 'h';
    char *result = (char *)malloc(20);
    sprintf(result, "%s x %d", x, i);
    return result;
}

void dog()
{
    char s[] = "my cat";
    int z = 12;
    cat(s, z);
}
```

Memory

- Code Execution:

- Memory:



```
char *cat(char *x, int i)
{
    static int r = 4;
    x[3] = 'h';
    char *result = (char *)malloc(20);
    sprintf(result, "%s x %d", x, i);
    return result;
}

void dog()
{
    char s[] = "my cat";
    int z = 12;
    cat(s, z);
}
```

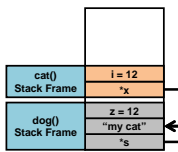

Memory

- Code Execution:

```
char *cat(char *x, int i)
{
    static int r = 4;
    x[3] = 'h';
    char *result = (char *)malloc(20);
    sprintf(result, "%s x %d", x, i);
    return result;
}

void dog()
{
    char s[] = "my cat";
    int z = 12;
    cat(s, z);
}
```

- Memory:



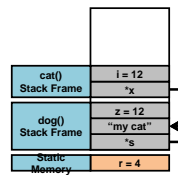
Memory

- Code Execution:

```
char *cat(char *x, int i)
{
    static int r = 4;
    x[3] = 'h';
    char *result = (char *)malloc(20);
    sprintf(result, "%s x %d", x, i);
    return result;
}

void dog()
{
    char s[] = "my cat";
    int z = 12;
    cat(s, z);
}
```

- Memory:



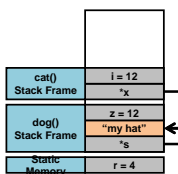
Memory

- Code Execution:

```
char *cat(char *x, int i)
{
    static int r = 4;
    x[3] = 'h';
    char *result = (char *)malloc(20);
    sprintf(result, "%s x %d", x, i);
    return result;
}

void dog()
{
    char s[] = "my cat";
    int z = 12;
    cat(s, z);
}
```

- Memory:



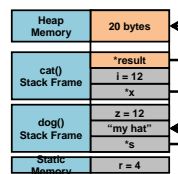
Memory

- Code Execution:

```
char *cat(char *x, int i)
{
    static int r = 4;
    x[3] = 'h';
    char *result = (char *)malloc(20);
    sprintf(result, "%s x %d", x, i);
    return result;
}

void dog()
{
    char s[] = "my cat";
    int z = 12;
    cat(s, z);
}
```

- Memory:



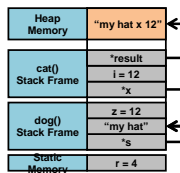
Memory

- Code Execution:

```
char *cat(char *x, int i)
{
    static int r = 4;
    x[3] = 'h';
    char *result = (char *)malloc(20);
    sprintf(result, "%s x %d", x, i);
    return result;
}

void dog()
{
    char s[] = "my cat";
    int z = 12;
    cat(s, z);
}
```

- Memory:



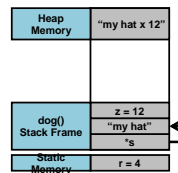
Memory

- Code Execution:

```
char *cat(char *x, int i)
{
    static int r = 4;
    x[3] = 'h';
    char *result = (char *)malloc(20);
    sprintf(result, "%s x %d", x, i);
    return result;
}

void dog()
{
    char s[] = "my cat";
    int z = 12;
    cat(s, z);
}
```

- Memory:



Precedence

- When multiple operations are applied to variables in C, an order of precedence is applied.

– Ex: `if (p++ <= 3) { /* ... */ }`

- Does p get incremented by one and then checked if it is less than or equal to 3?
- Does p get checked if it is less than or equal to 3 and then incremented by one?

Precedence

- Two examples

– Ex1: `if (p++ <= 3) { /* ... */ }`

– Ex2: `if (++p <= 3) { /* ... */ }`

- Result:

– Example 1's if statement is TRUE.

- (p <= 3) is done before (p++)

– Example 2's if statement is FALSE.

- (p++) is done before (p <= 3)

Precedence

- There are plenty of tables of precedence for the C language all over the Internet.
 - General Rule: If you're not sure, always enclose your statements in params!
 - Eg: `*z++;` \rightarrow `*(z++);`
 - Eg: `(q <= 3) ? (q++) : ((q <= 8) ? (q--) : (q++));`

Memory Casting

- One of the most useful built in functions in C is `sizeof()`.
- On most the machines you'll be working on:
 - `sizeof(int)` `== 4`
 - `sizeof(void *)` `== 8` (on 64-bit machines)
 - `sizeof(double)` `== 8`
 - `sizeof(char)` `== 1`

Memory Casting

- One observation that can be quickly made:
 - `sizeof(int *)` `== 8`
 - `sizeof(void *)` `== 8`
 - `sizeof(double *)` `== 8`
 - `sizeof(char *)` `== 8`
 - `sizeof(zzz *)` `== 8`

Memory Casting

- One observation that can be quickly made:
 - `sizeof(int *)` `== 8`
 - `sizeof(void *)` `== 8`
 - `sizeof(double *)` `== 8`
 - `sizeof(char *)` `== 8`
 - `sizeof(zzz *)` `== 8`

When functions don't care what the data is: they'll return a void !

Memory Casting

- Function definition for malloc():
– `void * malloc (size_t size);`
- However, your code may look something like:
– `char *s = malloc(1024);`

Memory Casting

Pointers may freely be cast from one type to another since they're of the same size!

- However, your code may look something like:
– `char *s = (char *)malloc(1024);`

Memory Casting

This is a blessing and a curse...

- However, your code may look something like:
– `char *s = (char *)malloc(1024);`

Memory Casting

This is a blessing and a curse...

- However, your code may look something like:
– `char *s = (char *)calloc(1024, 1);`
 `strcpy(s, "some data");`
 `float *f = (float *)s;`

Memory Casting

Modifying f now corrupts your string s!

- However, your code may look something like:

```
– char *s = (char *)calloc(1024, 1);  
  strcpy(s, "some data");  
  float *f = (float *)s;
```

Strings

Review of strings

- Sequence of zero or more characters, terminated by `NULL` (literally, the integer value 0)
- `NULL` terminates a string, but isn't part of it
 - important for `strlen()` – length doesn't include the `NUL`
- Strings are accessed through pointers/array names
- `#include <strings.h>` at program start

String literals

- Evaluating `" dog "` results in memory allocated for three characters `' d ' , ' o ' , ' g '`, plus terminating `NULL`

```
char *m = " dog " ;
```
- Note: If `m` is an array name, subtle difference:

```
char m[10] = " dog " ;
```

String literals

- Evaluating "dog" results in memory allocated for three characters 'd', 'o', 'g', plus terminating NULL

```
char *m = "dog";
```

- Note: If m is an array name, subtle difference:

10 bytes are allocated for this array ;

String literals

- Evaluating "dog" results in memory allocated for three characters 'd', 'o', 'g', plus terminating NULL

```
char *m = "dog";
```

- Note: If m is an array name, subtle difference:

10 bytes are allocated for this array ;

This is not a string literal;
It's an array initializer in disguise!
Equivalent to
{ 'd', 'o', 'g', '\0' }

String manipulation functions

- Read some "source" string(s), possibly write to some "destination" location

```
char *strcpy(char *dst, char const *src);
char *strcat(char *dst, char const *src);
```

- Programmer's responsibility to ensure that:
 - destination region large enough to hold result
 - source, destination regions don't overlap
 - "undefined" behavior in this case – according to C spec, anything could happen!

```
char m[10] = "dog";
strcpy(m+1, m);
```

String manipulation functions

- Read some "source" string(s), possibly write to some "destination" location

```
char *strcpy(char *dst, char const *src);
char *strcat(char *dst, char const *src);
```

- Programmer's responsibility to ensure that:
 - destination region large enough to hold result
 - source, destination regions don't overlap
 - "undefined" behavior in this case – according to C spec, anything could happen!

```
char m[10] = "dog";
strcpy(m+1, m);
```

Assuming that the implementation of strcpy starts copying left-to-right without checking for the presence of a terminating NUL first, what will happen?

strlen() and size_t

```
size_t strlen(char const *string);  
/* returns length of string */
```

- `size_t` is an unsigned integer type, used to define sizes of strings and (other) memory blocks
 - Reasonable to think of “size” as unsigned...
 - But beware! Expressions involving `strlen()` may be unsigned (perhaps unexpectedly)
- ```
if (strlen(x) - strlen(y) >= 0) ...
```
- avoid by casting:  

```
((int) (strlen(x)) - strlen(y) >= 0)
```

    - Problem: what if `x` or `y` is a very large string?
  - a better alternative: `(strlen(x) >= strlen(y))`

## strlen() and size\_t

```
size_t strlen(char const *string);
/* returns length of string */
```

- `size_t` is an unsigned integer type, used to define sizes of strings and (other) memory blocks
    - Reasonable to think of “size” as unsigned...
    - But beware! Expressions involving `strlen()` may be unsigned (perhaps unexpectedly) **always true!**
- ```
if (strlen(x) - strlen(y) >= 0) ...
```
- avoid by casting:

```
((int) (strlen(x)) - strlen(y) >= 0)
```

 - Problem: what if `x` or `y` is a very large string?
 - a better alternative: `(strlen(x) >= strlen(y))`

strcmp() “string comparison”

```
int strcmp(char const *s1, char const *s2);
```

- returns a value less than zero if `s1` precedes `s2` in lexicographical order;
 - returns zero if `s1` and `s2` are equal;
 - returns a value greater than zero if `s1` follows `s2`.
- Source of a common mistake:
 - seems reasonable to assume that `strcmp` returns “true” (nonzero) if `s1` and `s2` are equal; “false” (zero) otherwise
 - In fact, *exactly the opposite* is the case!

Restricted vs. unrestricted string functions

- Restricted versions: require an extra integer argument that bounds the operation

```
char *strncpy(char *dst, char const *src, size_t len);  
char *strncat(char *dst, char const *src, size_t len);  
int strncmp(char const *s1, char const *s2, size_t len);
```

- “safer” in that they avoid problems with missing `NULL` terminators
- safety concern with `strncpy`:

If bound isn't large enough, terminating `NUL` won't be written

Safe alternative:

```
strncpy(buffer, name, BSIZE);  
buffer[BSIZE-1] = '\0';
```

String searching

```
char *strstr(const char *haystack, const char
             *needle);
/* return a pointer to first occurrence of the
   substring needle in the string haystack. or NULL
   if the substring is not found */
```