W22

Thursday, October 6, 2022 6:03 PM

W22

CS 354 - Machine Organization & Programming
Thursday 9/8, 2022

Instructor: Deb Deppeler, 5376 CS, deppeler@wisc.edu

Office Hours: See Lectures on course web site: https://canvas.wisc.edu/courses/308770

Lectures

- Lecture 001: **145 Birge Hall,TR: 9:30 AM - 10:45 AM**
  LiveStream: **http://128.104.155.144/ClassroomStreams/birge145_stream.html**
- Lecture 002: **6210 Sewell Social Sciences Building,TR: 1:00 PM - 2:25 PM**
  LiveStream: **http://128.104.155.144/ClassroomStreams/socsci145_stream.html**

Description

An introduction to fundamental structures of computer systems and the C programming language with a focus on the low-level interrelationships and impacts on performance. Topics include the virtual address space and virtual memory, the heap and dynamic memory management, the memory hierarchy and caching, assembly language and the stack, communication and interrupts/signals, assemblers/linkers and compiling.

Today

Getting Started	C Program Structure
Welcome Watch recordings (Canvas) Coding in C Remotely	C Program Structure (L2-6) C Logical Control Flow Recall Variables Meet Pointers

Next Week

Topics: Pointers - 1D Arrays & Address Arithmetic, Passing Addresses
Read:
K&R Ch. 2: Types, Operators, and Expressions
K&R Ch. 3: Control Flow
K&R Ch. 4: Functions & Program Structure
K&R Ch. 5.1: Pointers and Addresses
K&R Ch. 5.2: Pointers and Function Arguments
K&R Ch. 5.3: Pointers and Arrays
K&R Ch. 5.4: Address Arithmetic

Do: Trace bingbangboom example on L2-6 to determine output, code up to verify
Start on project p1 (available soon)

Course Information

Textbooks

- The C Programming Language, Kernighan & Ritchie, 2nd Ed., 1988
- Computer Systems: A Programmer's Perspective, Bryant & O'Hallaron, 2nd Ed, 2010
  Note: 3rd edition or finding an online pdf is fine. (I cannot post a link)

Piazza

- is used for online course discussions and questions with classmates and the TAs about homeworks, projects, and course concepts as well as course logistics

CS Account

- provides access to CS Linux Computers with dev tools (rooms 1366, 1355, 1358, **1368**)
- is needed to access your CS 354 student folder used for some course projects
- same user name/password as your prior CS 200/300 CS accounts
- IF YOU ARE NEW TO CS, go to "My CS Account" on the csl.cs.wisc.edu web page
  URL: https://apps.cs.wisc.edu/accountapp/ (or see TA or Deb)

TAs: Teaching Assistants

- are graduate students with backgrounds in computer architecture and systems
- help with course concepts, Linux, C tools and language, homeworks and projects
- do consulting in 1366 or 1368 CS Linux Computer Lab during scheduled hours, which are posted on course website's "TA Consulting" page

PMs: Peer Mentors (available for in-person support for students)

- are undergraduate students that have recently completed CS 354
- hold drop-in hours and do a variety of activities to help students succeed, which are posted on course website's "PM Activities" page
- limited availability this semester as fewer students were available to hire

Coursework

Canvas will have all coursework hand in deadlines.

Exams (55%)

- Midterm (15%): Thursday Oct 6th, 7:30 - 9:30 PM
- Midterm (18%): Thursday Nov 10th, 7:30 - 9:30 PM
- Final (22%): Dec 21st, 7:25 PM - 9:25 PM

Conflict with these times? Complete the form at: http://tiny.cc/cs354-conflicts

Projects (30%): 6 projects, posted on course website

Homeworks (15%): ~10 homework quizzes, posted on course website

Coding in C Remotely - Get Connected to CS

❋ *Use the CS Linux lab computers for CS 354 programming.*

Access CS Linux Computers

Windows: get ssh program like MobaXterm and configure to connect to CS machines
Macs: open terminal and enter ssh <cs_account>@<machine>

machine names:
best-linux.cs.wisc.edu (Macs might cause issues with security certificates)
emperor-01.cs.wisc.edu through emperor-07.cs.wisc.edu
rockhopper-01.cs.wisc.edu through rockhopper-09.cs.wisc.edu
royal-01.cs.wisc.edu through royal-30.cs.wisc.edu
snares-01.cs.wisc.edu through snares-10.cs.wisc.edu

Learn some Linux Commands

command shell

→ How do you:
list the contents of a directory? Show details?

display what directory you're currently in?

copy a file?

remove a file?

move to another directory? Up a directory?

make a new directory?

rename a file or directory?

remove a directory?

get more information about commands?

Coding in C Remotely - Create your Source

1. Edit your Source File

```
$vim prog1.c
$vimtutor
```

→ Why vim?

```
/* title:  First C Program
 * file:   prog1.c
 * author: Jim Skrentny
 */

#include <stdio.h>     // for printf fprintf fgets
#include <stdlib.h>    // for malloc
#include <string.h>    // for strlen

int main() {

   // Prompt and read user's CS login
   char *str = malloc(50);

   printf("Enter your CS login: ");

   if (fgets (str, 50, stdin) == NULL)
      fprintf(stderr, "Error reading user input.\n");

   // Terminate the string
   int len = strlen(str);
   if (str[len - 1] == '\n') {
      str[len - 1] = '\0';
   }

   // Print out the CS login
   printf("Your login: %s\n", str);

   return 0;
}
```

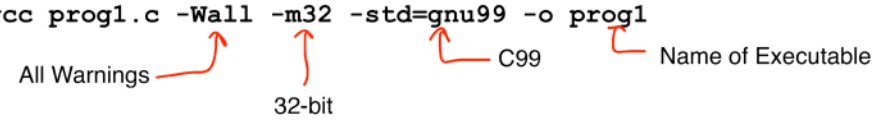
Coding in C Remotely - Compile/Run/Debug/Submit

2. Compile

```
$gcc prog1.c
```

OR

```
$gcc prog1.c -Wall -m32 -std=gnu99 -o prog1
```



3. Run

```
$a.out
```

→ Why a.out?

OR

```
$prog1
```
./prog1

4. Debug

printf() and GDB (Gnu Debugger)

5. Submit (required for projects)

- Download your source from the lab computer to your local machine
  Windows: drag and drop in MobaXterm window
  Macs: scp <csLogin>@<machine>:/path/to/remote/directory/file /path/to/local/destination
- Upload your source from your local machine to the course website

C Program Structure

- *Variables and functions must be declared before they're used.*
  - What is output by the following code?

```
#include <stdio.h>

int bing(int x) {
   x = x + 3;
   printf("bing %d\n", x);
   return x - 1;
}

int bang(int x) {
   x = x + 2;
   x = bing(x);
   printf("BanG %d\n", x);
   return x - 2;
}

int main(void) {
   int x = 1;
   bang(x);
   printf("BOOM %d\n", x);

   return 0;
}
```

OUTPUT:
Bing 6
BanG 5
BOOM 1

Functions

function: A module of code (NOT behaviors since C is not Object Oriented)

caller function: The process that called the function

callee function: The function that was called

Functions Sharing Data

argument: The data set by the caller function

parameter: The variable name that the callee assigns to a value passed.

pass-by-value (passing in): The function directly passes the value of a parameter into a function (No references in C)

return-by-value (passing out): Return a value directly (No references)

C Logical Control Flow

Sequencing

Execution starts in main.

Flows top to bottom (defining the order of functions does matter)

One statement then next

Selection

→ Which value(s) means true? Not Real `true` T 42 T -17 F 0

Use a Macro

Anything other than zero is true.

if - else

Like Java but with fewer guard rails.

→ What is output by this code when `money` is 11, -11, 0?

```
if (money = 0)      printf("you're broke\n");
else if (money < 0) printf("you're in debt\n");
else                printf("you've got money\n");
```

It will always result in "you've got money" as that first line with ALWAYS set money to zero, so it just goes to the else block.

→ What is output by this code when the date is 10/31? month = 10 day = 31

```
if (   10 -> month)
   if (   31 ->   day)
      printf("Happy Halloween!\n");
else
   printf("It's not October.\n");
```

It prints out Happy Halloween…but even dates that are invalid will work as long as there are no zeroes in the date as anything other than zero will evaluate to true.

switch like Java, but no strings!

C Logical Control Flow (cont.)

Repetition

```
int i = 0;
while (i < 11) {
   printf("%i\n", i);
   i++;
}
```

What is i? i = 11

```
for (int j = 0; j < 11; j++) {
   printf("%i\n", j);
}
```

What is j? Technically 11, but only exists in the loop scope.
In C it is common to declare loop variables outside of the loop definition.

```
int k = 0;
do {
   printf("%i\n", k);
   k++;
} while (k < 11);
```

What is k? Also 11.

Recall Variables

What? A *scalar variable* is a primitive unit of storage. int, char, short, long, etc.

→ Draw a basic memory diagram for the variable in the following code:

```
void someFunction(){
   int i = 44;
```

i | 44

Aspects of a Variable

identifier: name

value: data stored in variable's memory location

type: int, double, short, long, float -> representation of memory

address: starting location of the variable's memory (int*, char*…)

size: number of bytes for a variable. (use sizeof(type))

❋ *A scalar variable used as a source operand*

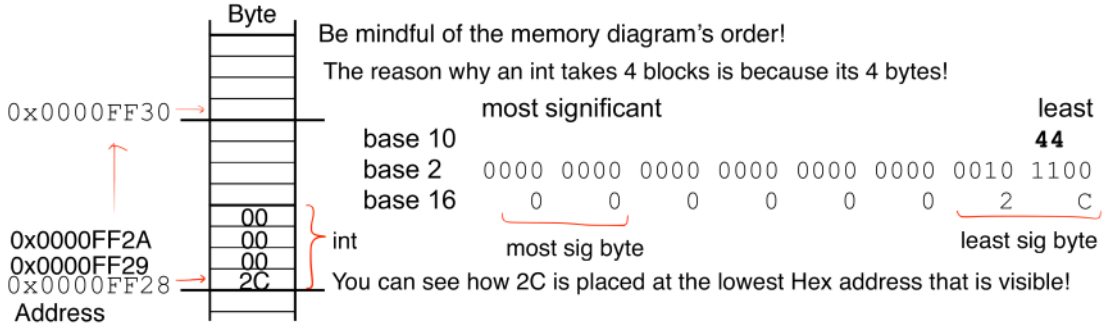
This is source, as i is being read. e.g., `printf("%i\n", i);`

❋ *A scalar variable used as a destination operand*

This is a destination, as i is being assigned. e.g., `i = 11;`

Linear Memory Diagram

A linear memory diagram is a view of memory as a sequence of bytes.



byte addressability: each byte has its own address.
(All of our variables in these diagrams will be "word-aligned" aka they will start at a multiple of the number of bytes that they need. So an integer is 4 bytes and will start at a multiple of 4 bytes for a pointer address.
endianess: byte ordering of variable's bytes when size > 1 byte

little endian: (CS 354 IA 32) Least significant (rightmost) byte is in the lowest address.

big endian: Most significant byte (leftmost) is in the lowest address.

Meet Pointers

What? A *pointer* variable is

- A scalar variable who's value is a memory address
- Similar to Java references

Why?

- For indirect access to memory
- Gives indirect address to functions
- Commonly used in C libraries
- Give access to the machine's memory mapped hardware.

How?

→ Consider the following code:

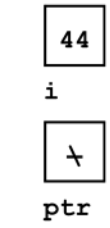
```
void someFunction(){
   int i = 44; Decimal Still! //0x0..2C

   int *ptr = NULL;
```

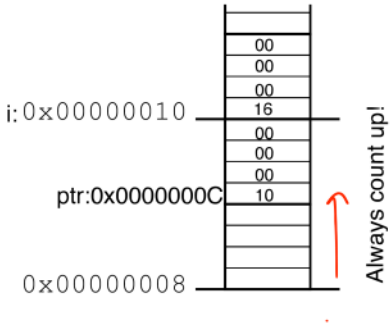
ptr = &i; //So now 10 is stored in the lowest point of the memory block for ptr.

*ptr = 22; //Deref'd pointer and now i is 22, so we change its value to 0x0…16.

Basic Diag.



Linear Diag.



→ What is `ptr`'s initial value? '\0' address? type? size?
0x0000… 0x0000000C int pointer 4 bytes

pointer: does the pointing, contains the address. Dereferenced using *.

pointee: is what's pointed to.

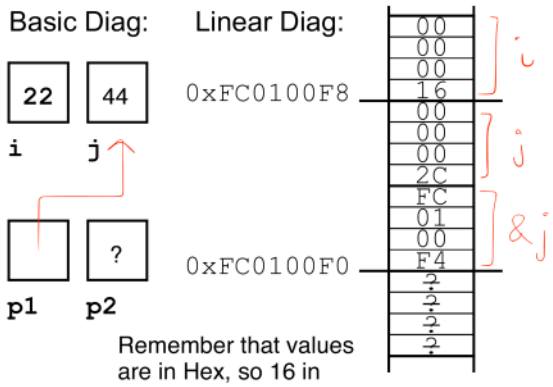
& *address of* operator: returns the address of its operand.

* *dereferencing* operator: access a pointer variable's value.

Practice Pointers

→ Complete the following diagrams and code so that they all correspond to each other:

```
void someFunction(){
   int i = 22
   int j = 44;
   int *p1 = &j
   int *p2;  //at addr 0xFC0100EC
```



Remember that values are in Hex, so 16 in hex is 22 in decimal!

Furthermore, Integers take 4 bytes, aka 4 cells!

→ What is `p1`'s value?

Pointer to J or 0xFC0100F8

→ Write the code to display `p1`'s pointee's value.

printf("%i\n", *p1);

→ Write the code to display `p1`'s value.

printf("%p\n", p1);

→ Is it useful to know a pointer's exact value?

Not really, but sometimes for debugging.

→ What is `p2`'s value?

Undefined, REALLY DANGEROUS! If you read from this you could get undefined behavior!

→ Write the code to initialize `p2` so that it points to nothing.

p2 = NULL;

→ What happens if the code below executes when `p2` is NULL?

```
printf("%i\n", *p2);
```

SEG FAULT (better than an intermettiant or undefined error)

→ What happens if the code below executes when `p2` is uninitialized?

```
printf("%i\n", *p2);
```

May be a segfault, 0, or some random garbage value.

→ Write the code to make `p2` point to `i`.

p2 = &i

→ How many pointer variables are declared in the code below?

```
void someFunction(){
   int* p1, p2;
```

Just one. p2 is an int.

→ What does the code below do?

```
int **q = &p1;
```

Pointer to a pointer to an int. (Double Pointer)

W23

CS 354 - Machine Organization & Programming
Tuesday Sept 13th and Thursday, Sept 15, 2022

Project p1: DUE on or before Friday 9/23 (get it done and submitted by 9/16 if possible)
Project p2A: Released Friday and due on or before Friday 9/30
Homework hw1: Assigned soon
Exam Conflicts (check entire semester): Report by 9/30 to: **http://tiny.cc/cs354-conflicts**
TA Lab Consulting & PM Activities are scheduled. See links on course front page.

Last Week

Welcome Watch recordings (p2-5) Welcome Coding in C Remotely (see recordings)	C Program Structure (L2-6) C Logical Control Flow Recall Variables Meet Pointers

This Week

`Tuesday`	`Thursday`
Meet Pointers (from L02) Practice Pointers (from L02) Recall 1D Arrays 1D Arrays and Pointers Passing Addresses	1D Arrays on the Heap Pointer Caveats Meet C Strings Meet `string.h`
`Read before Thursday` K&R Ch. 7.8.5: Storage Management K&R Ch. 5.5: Character Pointers and Functions K&R Ch. 5.6: Pointer Arrays; Pointers to Pointers	

Next Week

Topic: 2D Arrays and Pointers
Read:
- K&R Ch. 5.7: Multi-dimensional Arrays
- K&R Ch. 5.8: Initialization of Pointer Arrays
- K&R Ch. 5.9: Pointers vs. Multi-dimensional Arrays
- K&R Ch. 5.10: Command-line Arguments

Do: Finish project p1
Start project p2A

Recall 1D Arrays

What? An *array* is

- A compound unit of storage having elements of the same type.
- Accessed using an identifier and index.
- Allocated as a contigious fixed size block of memory.

Why?

- To store a collection of data of same type with fast access.
- easier than declaring an individual variable for each item.

How?

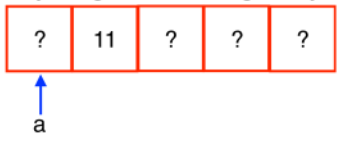
```
void someFunction(){
   int a[5]; //Declared and allocated space (Yes this is different from Java)!
```

→ How many integer elements have been allocated memory? 5

→ Where in memory was the array allocation made? Stack

→ Write the code that gives the element at index 1 a value of 11. a[1] = 11;

→ Draw a basic memory diagram showing array `a`.



❋ *In C, the identifier for a stack allocated array (SAA)*
IS NOT A VARIABLE, IT IS CONSIDERED CONSTANT! (so a = <whatever> is illegal!)

❋ *A SAA identifier used as a source operand* Provides the array address (pointer).

%p = print pointer e.g., `printf("%p\n", a);`

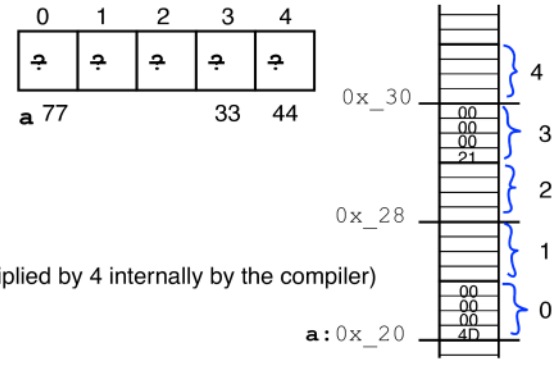
❋ *A SAA identifier used as a destination operand*

Bad! You cannot do a = ??? as it is not necessarily a variable anymore.

1D Arrays and Pointers

Given:

```
void someFunction(){
   int a[5];
```



Address Arithmetic is big fast

❋ `a[i]` = *(a + i) (scaled index, so i is multiplied by 4 internally by the compiler)

1. compute the address

   Start at a's beginning address
   Add a byte offset to get element i.

2. dereference the computed address to access the element *(a + i) [Parenthesis are required!]

→ Write address arithmetic code to give the element at index 3 a value of 33.

*(a + 3) = 33;

Jumped by 12 bytes as 3 * 4 = 12. So at 0x_2C we store 33 in converted to hex, which is 0x_21

→ Write address arithmetic code equivalent to `a[0] = 77;`

*(a + 0) = 77 or *a = 77;

Now we store 0x_4D at the start of a.

Using a Pointer

→ Write the code to create a pointer `p` having the address of array `a` above.

int* p = a;

If you tried &a, C would be okay with it, it would just drop the & and warn you.

Now p will hold the address of a, which is 0x_20

→ Write the code that uses p to give the element in `a` at index 4 a value of 44.

*(p + 4) = 44;

This does affect a!

❋ *In C, pointers and arrays* are closely related but not the same.

pointers are mutable, arrays are not.

Passing Addresses

Recall *Call Stack Tracing*:

- Manually trace function calls.
- Each function gets a "box" (stack frame)
- The top box is the currently running function.

➢ What is output by the code below?

```
void f(int pv1, int *pv2, int *pv3, int pv4[]) {
   int lv = pv1 + *pv2 + *pv3 + pv4[0];
   pv1    = 11;
   *pv2   = 22;
   *pv3   = 33;
   pv4[0] = lv;
   pv4[1] = 44;
}
int main(void) {
   int lv1 = 1, lv2 = 2;
   int *lv3;
   int lv4[] = {4,5,6};
   lv3 = lv4 + 2;
   f(lv1, &lv2, lv3, lv4);
1, 22, 33 printf("%i,%i,%i\n",lv1,lv2,*lv3);
13, 44, 33 printf("%i,%i,%i\n",lv4[0],lv4[1],lv4[2]);
   return 0;
}
```

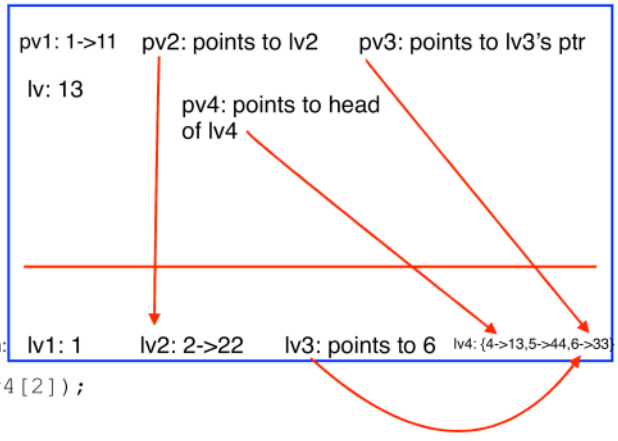
1 + 2 + 6 + 4 = 13

changes lv2 to 22 (orig 2)

changes lv3 to 33 (orig 6)

First index changed to 13

2nd index changed to 44



Pass-by-Value which parameters do what?

- scalars: param is a scalar variable that gets a copy of its scalar argument pv1
- pointers: param is a ptr var, it gets a copy of pv2 and pv3
- arrays: param is a array variable, it gets a copy of the starting address.

❋ *Changing a callee's parameter*

Changes the callee's copy!!! Doesn't affect caller.

❋ *Passing an address*

Requires trust (Arrays and Pointers), as changing the values associated with an address WILL change both the callee and caller's value!

THIS is how you change values in a different scope in C without returning!

1D Arrays on the Heap

What? Two key memory segments used by a program are the

STACK Fixed in size while the program is running. and HEAP dynamic memory allocation during runtime.
static (fixed in size) allocations
allocation size known during compile time

Why? Heap memory enables

- To access more memory than available at compile time.
- Allows you to have blocks of memory allocated and freed in an arbitrary order (No Garbage Collection).

How? Do #include <stdlib.h>

Function `void* malloc(size_in_bytes)`
Reserves a block of heap memory and returns an address to the start of that block.

Function `void free(void* ptr)`
Will release the heap block that the ptr given points to (any pointer works)

Operator `sizeof(operand)` returns the size in bytes of a type/operand. So do malloc(sizeof(int)) not malloc(4)

→ For IA-32 (x86), what value is returned by `sizeof(double)`? `sizeof(char)`? `sizeof(int)`?
sizeof(double) = 8, sizeof(char) = 1, sizeof(int) = 4

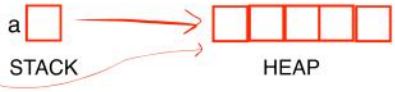
→ Write the code to dynamically allocate an integer array named `a` having 5 elements.

```
void someFunction(){
  int* a = malloc(sizeof(int)* 5);
```

heap

Square brackets for array def only works if you're putting it on the stack!

→ Draw a memory diagram showing array `a`.



Remember to free your memory!

→ Write the code that gives the element at indexes 0, 1 and 2 a values of 0, 11 and 22 by using pointer dereferencing, indexing, and address arithmetic respectively.

```
*a = 0; (Because an array is its first index)
a[1] = 1; (Indexing is safe if heap or stack)
*(a + 2) = 22;
```

→ Write the code that uses a pointer named `p` to give the element at index 3 a value of 33.

```
int* p = a; (a is an address so no deref needed)
*(p + 3) = 33;
```

OR

```
int* p = a + 3;
*p = 33; (This also results in negative indexing)
```

→ Write the code that frees array `a`'s heap memory.

```
free(a); //This results in a dangling pointer! Dangerous as if read it could result in a segfault
a = NULL; //This will make it more safe.
p = NULL; //We assume p still exists so we need to do the same to it.
Freeing p would work as well. But you can only do one or the other!!
```

Pointer Caveats

- *Don't dereference uninitialized or NULL pointers!*

```
int *p;
*p = 11;
```

```
int *q = NULL;
*q = 11;
```

Definitely a segfault!

What does this pointer point to??? Literally junk data. Intermettiant error!

- *Don't dereference freed pointers!*

```
int *p = malloc(sizeof(int));
int *q = p;
. . .
free(p);
. . .
*q = 11;
```

The heap memory that p and q points to just..doesn't exist! So trying to deref the dangling ptr q is bad Intermettiant error!

dangling pointer: A pointer variable to an address to heap memory that has been freed. (Alias or original ptr)

- *Watch out for heap memory leaks!*

memory leak: Heap memory that is unusable as it has not been freed properly, just sitting there.

```
int *p = malloc(sizeof(int));
int *q = malloc(sizeof(int));
. . .
p = q;
```

Now p points to q, so the heap memory allocated to p's int is basically lost as nothing points to it. We can't free it so it just wastes space, memory leak! If you save p's pointer somewhere, its not a leak.

- *Be careful with testing for equality!*

assume p and q are pointers

p = q	compares nothing because it's assignment	
p == q	compares values in pointers	only true when p and q point to the same memory
*p == *q	compares values in pointees	only true when p and q have the same deref'd value

- *Don't return addresses of local variables!*

```
int *ex1() {
   int i = 11;
   return &i;
}
```

Out of scope as the memory gets cleared!
Local variable on stack

```
int *ex2(int size) {
   int a[size];
   return a;
}
```

Out of scope as the memory gets cleared!
Local variable on stack again

You gotta to put it on the heap

Meet C Strings

What? A *string* is

- A sequence of characters terminated with a null character ('\0')
- A 1D array of char with size/length + 1 (The +1 is for the null char)

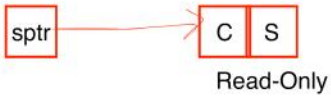
What? A *string literal* is "CS 354"

- A constant source code string
- Doesn't include the null char in length! The length is 6!

❋ *In most cases, a string literal used as a source operand*

How? Initialization

```
void someFunction(){
   char *sptr = "CS 354";
```

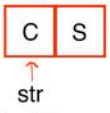


→ Draw the memory diagram for `sptr`.

→ Draw the memory diagram for `str` below.

```
char str[9] = "CS 354";
```

Stack Allocated Array
str is the first element of the array.



→ During execution, where is `str` allocated?
STACK

How? Assignment

→ Given `str` and `sptr` declared in `somefunction` above, what happens with the following code?

```
sptr = "mumpsimus";
```
//Okay sptr can point to a different char.

```
str = "folderol";
```
//Not okay, compiler error as the array is considered constant.

Cannot assign to SAA identifier.
You can loop through and change each char individually.

❋ *Caveat: Assignment cannot be used* to copy character arrays.

Meet `string.h`

What? *`string.h`* is a collection of useful functions to manipulate C strings. (Null terminated characters)
(All are const char* str, meaning they do not affect inputs)

```
int strlen(const char *str)
```
Returns the length of string `str` up to but *not* including the null character.

```
int strcmp(const char *str1, const char *str2)
```
Compares the string pointed to by `str1` to the string pointed to by `str2`.

returns:		
	< 0 (a negative)	if str1 comes before str2
	0	if str1 is the same as str2
	>0 (a positive)	if str1 comes after str2

```
char *strcpy(char *dest, const char *src)
```
Copies the string pointed to by `src` to the memory pointed to by `dest` and terminates with the null character.

Allows you to set a string to another string. Make sure you have space in dest!

```
char *strcat(char *dest, const char *src)
```
Appends the string pointed to by `src` to the end of the string pointed to by `dest` and terminates with the null character.

- *Ensure the destination character array* is large enough for the result including the null terminating character.

buffer overflow: Exceed the bounds of the destination array.

How? `strcpy`

→ Given `str` and `sptr` as declared in `somefunction` on the previous page, what happens with the following code?

```
strcpy(str, "folderol");
```
It's 8 + 1 (8 chars + 1 null char), so its okay as we have space for 9.

```
strcpy(str, "formication");
```
Runtime error, buffer overflow might touch memory you don't want!

```
strcpy(sptr, "vomitory");
```
Cannot copy into code segment! Just set it to a new string directly.

- *Rather than assignment, strcpy (or strncpy) must be used to* copy a C string from one array to another.
- *Caveat: Beware of* BUFFER OVERFLOW or attempting to write to CODE or DATA.