Version 1.03

UCSD CSE 30

Computer Organization and Systems Programming

C Programming Part 3

Lecture 8 – Oct 18, 2022

Keith Muller

C Precedence and Pointers

- ++ -- pre and post increment combined with pointers can create code that is complex, hard to read and difficult to maintain
- Use () to help readability

Operator	Description	Associativity
() [] . -> ++ --	Parentheses or function call Brackets or array subscript Dot or Member selection operator Arrow operator Postfix increment/decrement	left to right
++ -- + - ! ~ (type) * & sizeof	Prefix increment/decrement Unary plus and minus not operator and bitwise complement type cast Indirection or dereference operator Address of operator Determine size in bytes	right to left
* / %	Multiplication, division and modulus	left to right
+ -	Addition and subtraction	left to right
<< >>	Bitwise left shift and right shift	left to right
< <= > >=	relational less than/less than equal to relational greater than/greater than or equal to	left to right
== !=	Relational equal to or not equal to	left to right
&&	Bitwise AND	left to right
^	Bitwise exclusive OR	left to right
|	Bitwise inclusive OR	left to right
&&	Logical AND	left to right
||	Logical OR	left to right
? :	Ternary operator	right to left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment operator Addition/subtraction assignment Multiplication/division assignment Modulus and bitwise assignment Bitwise exclusive/inclusive OR assignment	right to left
,	comma operator	left to right

x

Pointer Practice

Operator	Description	Associativity
() [] . -> ++ --	Parentheses or function call Brackets or array subscript Dot or Member selection operator Arrow operator Postfix increment/decrement	left to right
++ -- + - ! ~ (type) * & sizeof	Prefix increment/decrement Unary plus and minus not operator and bitwise complement type cast Indirection or dereference operator Address of operator Determine size in bytes	right to left

common	Alternate	Meaning
`*p++`	`*(p++)`	The Rvalue is the object that p points at; then increment pointer p to next element
`(*p)++`		The Rvalue is the object that p points at; then increment the object
`*++p`	`*(++p)`	Increment pointer p first to the next element; the Rvalue is the object that the incremented pointer points at
`++*p`	`++(*p)`	The Rvalue is the incremented value of the object that p points at

x

Pointer Practice

```
int x;
int *p;
x = *(p+1); //contents of p[1]
x = *p + 1;//p[0] + 1
x = (*p)++;
  ⇒x = *p ; *p = *p + 1;

x = *p++;
x =(*p++);
x = *(p)++;
x = *(p++);
  ⇒x = *p ; p =  p + 1;

x = *++p;
  ⇒ p = p + 1 ; x = *p ;
```

Operator	Description	Associativity
() [] . -> ++ --	Parentheses or function call Brackets or array subscript Dot or Member selection operator Arrow operator Postfix increment/decrement	left to right
++ -- + - ! ~ (type) * & sizeof	Prefix increment/decrement Unary plus and minus not operator and bitwise complement type cast Indirection or dereference operator Address of operator Determine size in bytes	right to left

common	Alternate	Meaning
`*p++`	`*(p++)`	The Rvalue is the object that p points at; then increment pointer p to next element
`(*p)++`		The Rvalue is the object that p points at; then increment the object
`*++p`	`*(++p)`	Increment pointer p first to the next element; the Rvalue is the object that the incremented pointer points at
`++*p`	`++(*p)`	The Rvalue is the incremented value of the object that p points at

x

Example of a hard-to-understand pointer statement

```
int array[] = {2, 5, 7, 9, 11, 13};
int *ptr = array;
int x;
```

common	Alternate	Meaning
*p++	*(p++)	The Rvalue is the object that p points at; then increment pointer p to next element
(*p)++		The Rvalue is the object that p points at; then increment the object
*++p	*(++p)	Increment pointer p first to the next element; the Rvalue is the object that the incremented pointer points at
++*p	++(*p)	The Rvalue is the incremented value of the object that p points at

```
x = 1 + (*ptr++)++; // yuck!!
```

```
/* Same as the one line above */
x = 1 + *ptr;           // x = 1 + *ptr (2) = 3;

*ptr = *ptr + 1;        // (*ptr)++ is array[0]= 2 + 1;

ptr = 1 + ptr;          // ptr = &array[1] = now points at 5
```


x

Using Pointers to Traverse an array

```
char x[] = "Word:One Two Three;
int cnt = (int)(sizeof(x) / sizeof(*x));
char *ptr;
int j = 0;
ptr = x;

while (j < cnt) {
    if (*(ptr + j++) == ':')
        break;
}
printf("%s\n", ptr + j); // One Two Three
```

Brute force translation to pointers

```
char x[] = "Word:One Two Three;
int cnt = (int)(sizeof(x) / sizeof(*x));

char *ptr;
char *xpt;
ptr = x; //or &x[0
xpt = ptr + cnt;

while (ptr < xpt) {
    if (*ptr++ == ':')
        break;
}
printf("%s\n", ptr); // One Two Three
```

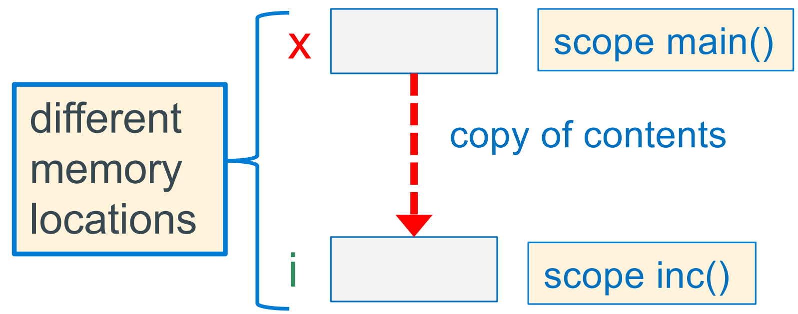
More common way to use pointers

x

Passing Parameters – Call by Value Example

```
int main(void)
{
    int x = 5;
    inc(x);                 // makes a copy of x
    printf("%d\n", x); // 5 or 6 ?
}

void inc(int i)         // i is local to inc
{
    ++i;
}
```

if this was an expression like inc(x+1) it evaluates and stores the result in the memory allocated for the copy



- when `inc(x)` is called, a copy of x is made to another memory location
  - `inc()` cannot change the variable x since `inc()` does not have the address of x, it is local to `main()` so, 5 is printed
- The `inc()` function is free to change it's copy of the argument (just like any local variable) remember it does <u>NOT</u> change the parameter in `main()`

x

Output Parameters (Mimics call by reference)

- Passing a pointer parameter with the **intent** that the called function will use the address it to store values for use by the calling function, then pointer parameter is called an **output parameter**
- To pass the address of a variable x use the **address operator** (&x) **or** the contents of a pointer variable that points at x, or the name of an array (the arrays address)
- To be receive an address in the called function, define the corresponding parameter type to be a pointer
  - It is common to describe this method as: "pass a pointer to x
- C is still using "*pass by value*"
  - we pass the **value** of the address/pointer in a **parameter copy**
  - **The called routine** uses the address to change a variable in the caller's scope

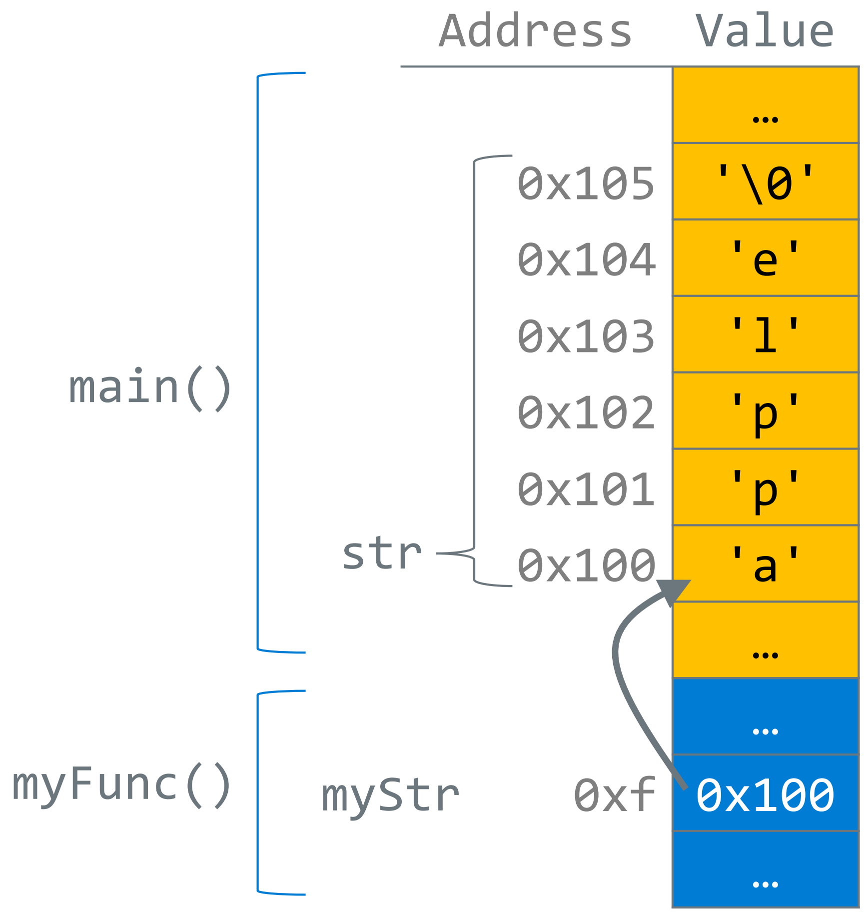
```
void inc(int *, char *, char **);
int main(void)
{
    int x = 5;
    char str[] = "string";
    char *ptr;
    inc(&x, str, &ptr);
    printf("%d %s\n", x, ptr );
    return EXIT_SUCCESS;
}

void
inc(int *p, char *cp, char **pcp)
{
    *p += 1; // or (*p)++
    printf("%s\n",cp);
    *pcp = cp + 1;
}
```

```
prints:
string
6 tring
```


x

Passing Arrays (Strings)

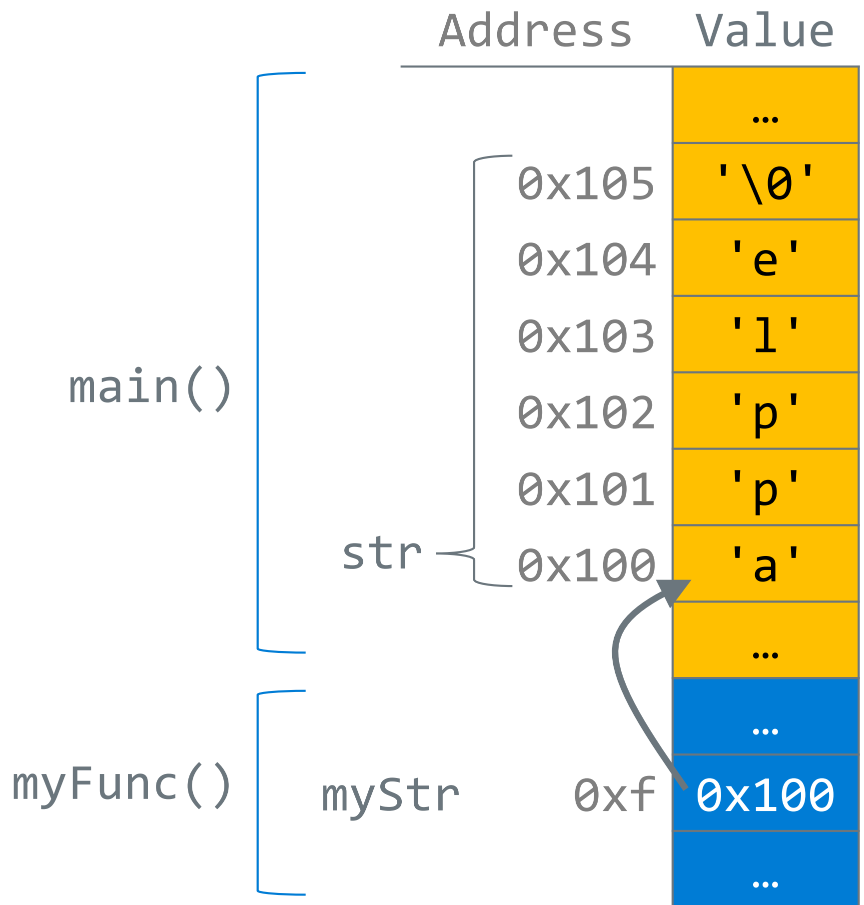
```
void
myFunc(char *myStr) {
	...
}

int
main(void) {
	char str[6];
	strcpy(str, "apple");
	myFunc(str);
	...
}
```



x

Passing Arrays (Strings)

```
void
myFunc(char *myStr) {
        myStr[4] = 'y'; // not safe!
}

int
main(void) {
        char str[6];
        strcpy(str, "apple");
        myFunc(str);
        ...
}
```

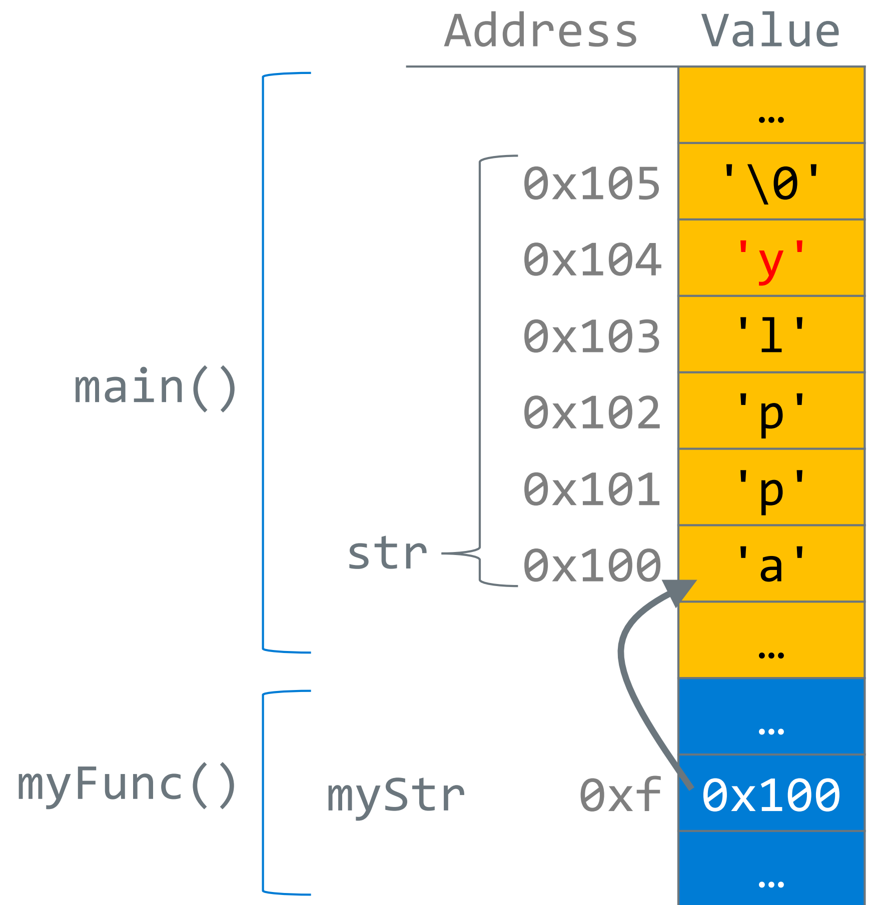


if we modify characters in **myFunc**, the changes will persist back in **main**!

x

Passing Arrays (Strings)

```
void
myFunc(char *myStr) {
        myStr[4] = 'y'; // not safe!
}

int
main(void) {
        char str[6];
        strcpy(str, "apple");
        myFunc(str);
        ...
}
```



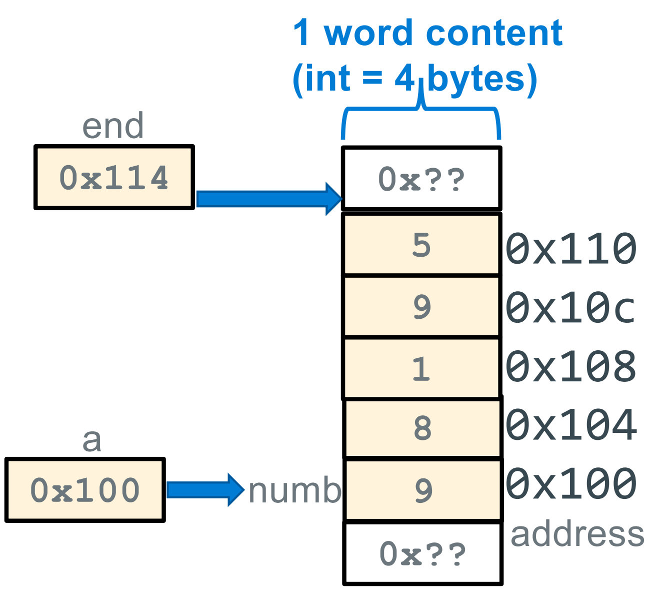
if we modify characters in **myFunc**, the changes will persist back in **main**!

Arrays As Parameters, Approach 1: Pass the size

Two ways to pass array size

1. pass the count as an additional argument
2. add a sentinel element as the last element

remember you can only use sizeof() to calculate element count where the array is defined



Arrays do not know their own size

```c
int sumAll(int *a, int size);
int main(void)
{
  int numb[] = {9, 8, 1, 9, 5};
  int cnt = sizeof(numb)/sizeof(numb[0]);

  printf("sum is: %d\n", sumAll(numb, cnt););
  return EXIT_SUCCESS;
}
```

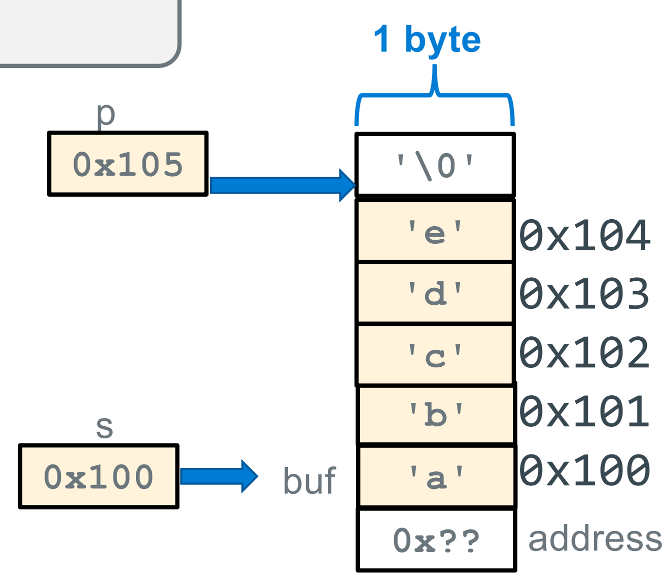
```c
int sumAll(int *a, int size)
{
  int sum = 0;
  int *end;
  end = a + size;

  while (a < end)
    sum += *a++;
  return sum;
}
```

Arrays As Parameters, Approach 2: Use a sentinel element

- A sentinel is an element that contains a value that is not part of the normal data range
  - Forms of 0 are often used (like with strings). Examples: '\0', NULL

```
int strlen(char *a);
int main(void)
{
  char buf[] = {'a', 'b', 'c', 'd', 'e', '\0'}; // string

  printf("Number of chars is: %d\n", strlen(buf));
  return EXIT_SUCCESS;
}
```

```
/* Assumes parameter is a terminated string */
int strlen(char *s)
{
   char *p = s;
   if (p == NULL)
       return 0;
   while (*p++)
       ;
    return (p – s - 1);
}
```



x

Copying Strings: Use the Sentinel; libc: strcpy(), strncpy()

- To copy an array, you must copy each character from source to destination array
- Watch overwrites: strcpy assumes the target array size is equal or larger than source array

index	0	1	2	3	4	5
char	'H'	'e'	'l'	'l'	'o'	'\0'

```
char str1[80];
strcpy(str1, "hello");
```

```
// strncpy adds a length limit on copy
char str1[6];
strncpy(str1, "hello", 5); // \0 not copied
str1[5] = '\0'; // make sure \0 terminated
```

```
char *strcpy(char *s0, char *s1)
{
    char *str = s0;

    if ((s0 == NULL) || (s1 == NULL))
       return NULL;
    while (*s0++ = *s1++)
       ;
    return str;
}
```

```
char *strncpy(char *s0, char *s1, int len)
{
    char *str = s0;
    if ((s0 == NULL) || (s1 == NULL))
       return NULL;

    while ((*s0++ = *s1++) && --len)
       ;
    return str;
}
```


x

2D Arrays

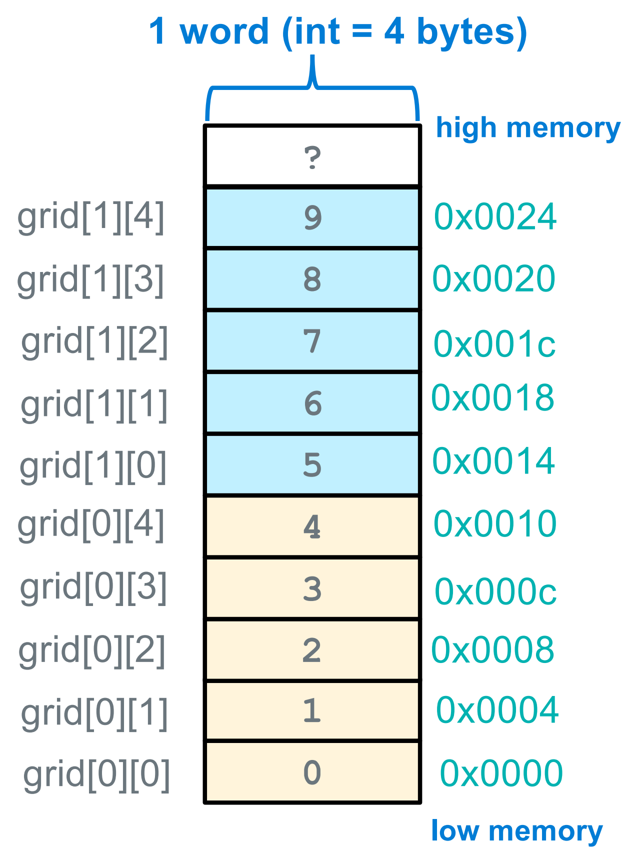
- Generic (uniform) 2D array format:

```
type name[rows][cols] = {{values},…,{values}};
```

  - allocates a single, <u>contiguous</u> block of memory
  - The array is organized in ***row-major*** *format*

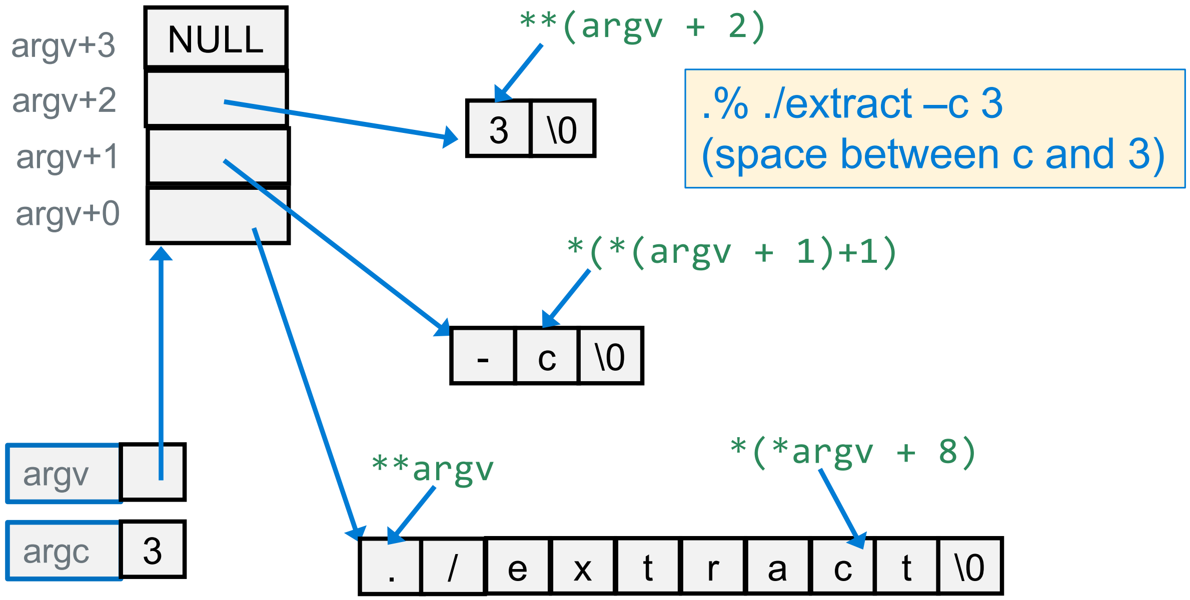
```
// a 2-row, 3-column array of char
char matrix[2][3];

// a 2-row, 5-column (row length) array of ints
// Must specify row length, compiler counts rows

int grid[][5] = {
  {0, 1, 2, 3, 4},
  {5, 6, 7, 8, 9}
};

grid[1][2] using pointers is *( *(grid + 1) + 2)
```

[1][0]	[1][1]	[1][2]	[1][3]	[1][4]
[0][0]	[0][1]	[0][2]	[0][3]	[0][4]



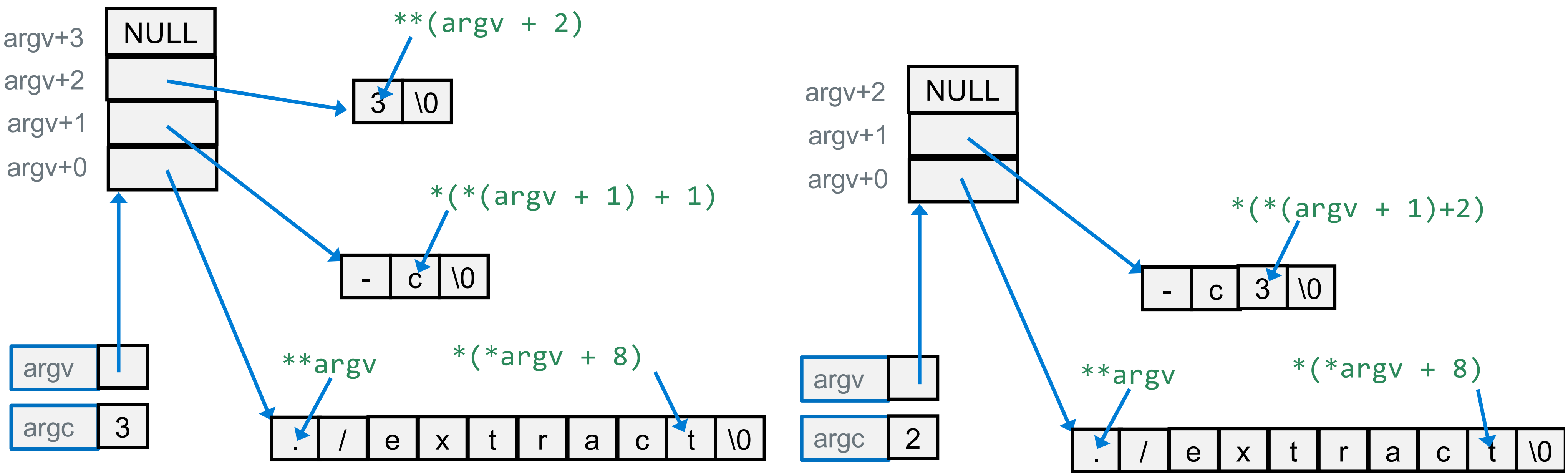
x

Array of Pointers: main() : argc, argv




```
int main(int argc, char **argv)
{
    char *pt;
    while ((pt = *argv++) != NULL) {
       while (*pt != '\0')
           putchar(*pt++);
       putchar('\n');
    }
    return EXIT_SUCCESS;
}
```

main() Command line arguments: argc, argv



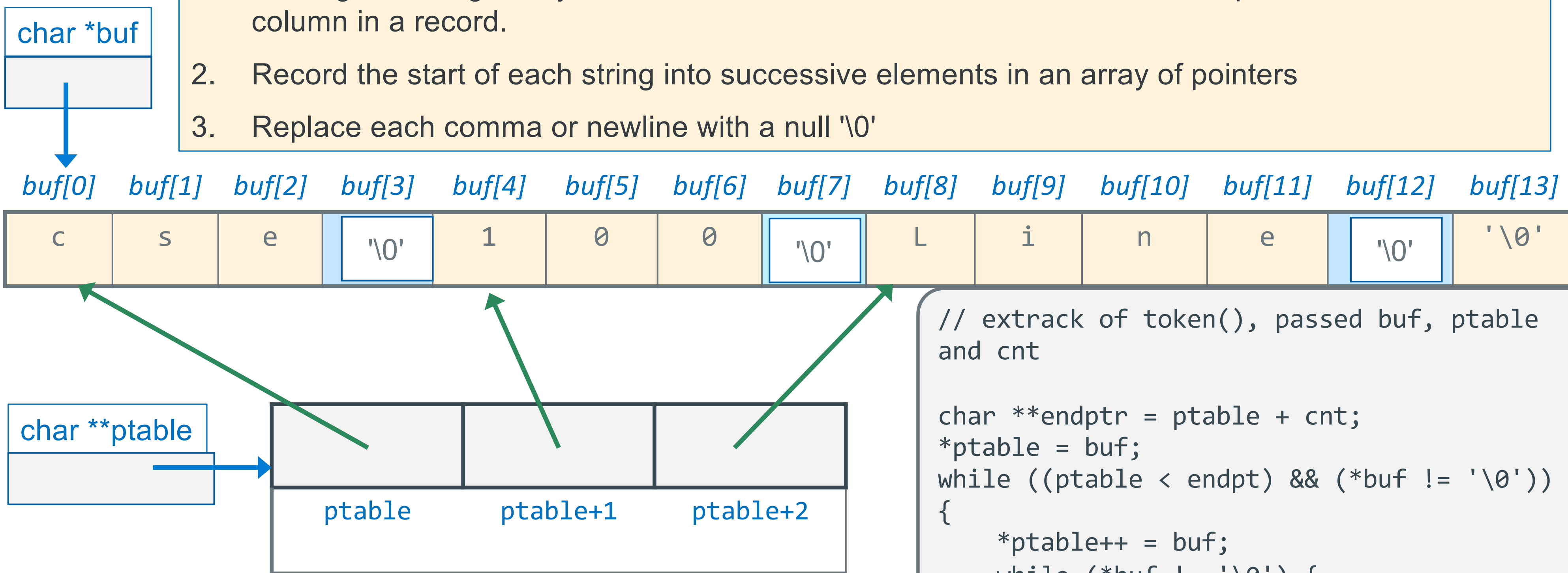
.% ./extract –c 3
(space between c and 3)

.% ./extract –c3
(No space between c and 3)

x

PA4: Creating a 2D Array of Mutable String Pointers

1. Break a string of comma separated words into individual strings without copying. Do This by walking the string until you see an either a comma , or a newline \n. Each points at a field or column in a record.
2. Record the start of each string into successive elements in an array of pointers
3. Replace each comma or newline with a null '\0'




```
./extract -c3
```

```
// extrack of token(), passed buf, ptable
and cnt

char **endptr = ptable + cnt;
*ptable = buf;
while ((ptable < endpt) && (*buf != '\0'))
{
    *ptable++ = buf;
    while (*buf != '\0') {
        /* process the chars, inc buf++ */
    }
}
// check for too many or too few fields
```

strtol() and strtoul() examples of passing a pointer to a pointer

```
long int strtol(const char *str, char **endptr, int base);

unsigned long int strtoul(const char *str, char **endptr, int base);
```

reruns the string converted to a long or unsigned long

str pointer to the string to convert

endptr pass the address of a variable that is a char pointer (output variable)

base: number base used by the string

- **Example**: string is to contain just positive numbers >= 0 (in ascii) with no extra stuff
- If the string is not valid, then
  - `*endptr != '\0'` then string contains more than just numbers (bad input)
  - `*endptr` stores the address of the first invalid character found in the buffer pointed (`str`)
- How to use endptr when it does not contain NULL:
  - If there are other conversion errors (you can read the man page) then errno != 0
  - When conversion is ok, `errno` is unaltered (always clear it before calling these routines)

x

strtol() and strtoul() examples of passing a pointer to a pointer

```
#include <stdlib.h>
#include <errno.h>
char *endptr;
char buf[] = "33";  // test buffer string
int number;

errno = 0; // set errno to 0 (zero) before each call
number = (int)strtol(buf, &endptr, 10)
// check if the string was a proper number
// *entpr should be at the end of the string == '\0'

if ((*endptr != '\0') || (errno != 0)) {
   // handle the error
}
printf("%d\n", number);
```


x

String Literals (Read-Only) in Expressions

- When strings in quotations (*e.g.*, "string") are **part of** an **expression** (*i.e.*, *not* part of an *array initialization*) they are called ***string literals***

```
printf("literal\n");
printf("literal %s\n", "another literal");
```

- What is a ***string literal:***
  - Is a null-terminated string in a **`const char array`**
  - Located in the **read-only data** segment of memory
  - Is not assigned a variable name by the compiler, so it is only accessible by the location in memory where it is stored
- **String literals** are a type of ***anonymous variable***
  - Memory containing data without a name bound to them (only the address is known)
- The *string literal* in the printf()'s, are replaced with the starting address of the corresponding array (first or [0] element) when the code is compiled

x

String Literals, Mutable and Immutable arrays

```
char mess1[] = "Hello World";
char *ptr = mess1;
*(ptr + 5) = '\0'; // shortens string to "Hello"
```

- mess1 is a **mutable** array (type is char [ ]) with enough space to hold the string + ’\0’
  - You **can change** array contents

```
char *mess2 = "Hello World";  // "Hello World" is a string literal
                              // mess2 is a pointer NOT an array!
```

- In the example above, "Hello World" is immutable string literal (array)
  - "Hello World" is not associated with a variable name; anonymous variable
  - "Hello World" has space to hold the string + ’\0’
  - "Hello World" is read only (immutable) and cannot be modified at runtime
- mess2 is a **pointer** to an **immutable** array with space to hold the string + ’\0’

x

Be Careful with C Strings and Arrays of Chars

mess2 **pointer** to an **immutable** array with space to hold the string + '\0'

- you **cannot change** array contents, but you can change what mess2 points at

```
char *mess2 = "Hello World";  // "Hello World" is a string literal
                              // mess2 is a pointer NOT an array!
*mess = 'h';                  // undefined in C, linux seg fault
mess2 = mess1;                // where mess2 points can be changed
```

- mess3 is an array but does not contain a '\0'
  - SO, IT IS **NOT** A VALID STRING

```
char mess3[] = {'H','e','l','l','o',' ','W','o','r','l','d'};
```


x

Returning a Pointer To a Local Variable (Dangling Pointer)

- There are many situations where a function will return a pointer, but a function must never return a pointer to a memory location that is no longer valid such as:

1. Address of a passed parameter copy as the caller may or will deallocate it after the call
2. Address of a local variable (automatic) that is invalid on function return

- These errors are called a dangling pointer

n is a parameter with the scope of bad_idea it is no longer valid after the function returns

```
int *bad_idea(int n)
{
    return &n; // NEVER do this
}
```

a is an automatic (local) with a scope and **lifetime** within bad_idea2 a is no longer a valid location after the function returns

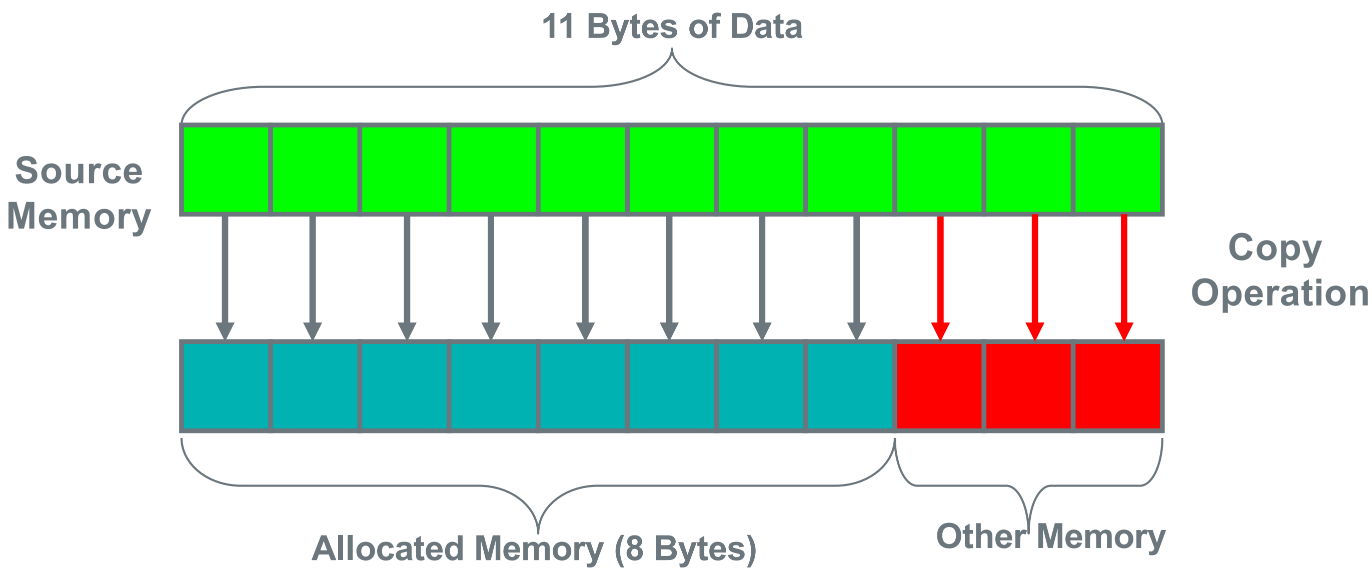
```
int *bad_idea2(int n)
{
    int a = n * n;
    return &a; // NEVER do this
}
```

```
/*
 * this is ok to do
 * it is NOT a dangling
 * pointer
 */

int *ok(int n)
{
    static int a = n * n;
    return &a; // ok
}
```


x

string buffer overflow: common security flaw

- A buffer overflow occurs when data is written outside the boundaries of the memory allocated to target variable (or target buffer)
- `strcpy()` is a very *common source of buffer overrun security flaws*:
  - always ensure that the destination array is **large enough** (and don't forget the null terminator)
- `strcpy()` can cause problems when the ***destination*** and **source** regions ***overlap***



x

strcpy() buffer overflow: over-write of an adjacent variable

```c
int main(void)        /* file test.c */
{
    char s1[] = "before";
    char r2[4] = "xyz";
    char s2[] = "after";

    printf("s1: %s\ns2: %s\nr2:%s\n", s1, s2, r2);

    strcpy(r2,"hello"); // length > buffer size

    printf("\ns1: %s\ns2: %s\nr2:%s\n", s1, s2, r2);
    return EXIT_SUCCESS;
}
```

these are mutable arrays, not literals

compile on pi-cluster with gcc test.c

```
./a.out
s1: before
s2: after
r2: xyz

s1: o
s2: after
r2: hello
```

s2[0]	s2[1]	s2[2]	s2[3]	s2[4]	s2[5]	r2[0]	r2[1]	r2[2]	r2[3]	s1[0]	s1[1]	s1[2]	s1[3]	s1[4]	s1[5]	s1[6]
'a'	'f'	't'	'e'	'r'	'\0'	'x'	'y'	'z'	'\0'	'b'	'e'	'f'	'o'	'r'	'e'	'\0'

low memory address

before strcpy() overflow

high memory address

s2[0]	s2[1]	s2[2]	s2[3]	s2[4]	s2[5]	r2[0]	r2[1]	r2[2]	r2[3]	s1[0]	s1[1]	s1[2]	s1[3]	s1[4]	s1[5]	s1[6]
'a'	'f'	't'	'e'	'r'	'\0'	'h'	'e'	'l'	'l'	'o'	'\0'	'f'	'o'	'r'	'e'	'\0'

after strcpy() overflow

X

Pointer to Pointers (Double, Triple and … Indirection

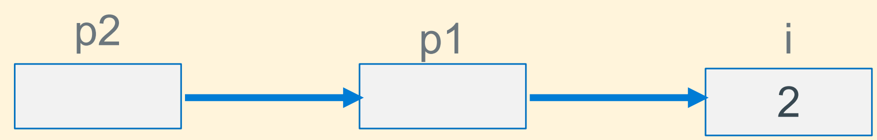
- A pointer **<u>cannot</u>** point at itself, why?

```
int *p = &p; /* is not legal – type mismatch */
```

  - p is defined as (int *), a pointer to an int, **but**
  - the type of &p is (int **), a pointer to a pointer to an int

- Define a pointer to a pointer (p2 below)

```
int i = 2;
int *p1;
int **p2;
p1 = &i;
p2 = &p1;
printf("%d\n", **p2 * **p2);
```



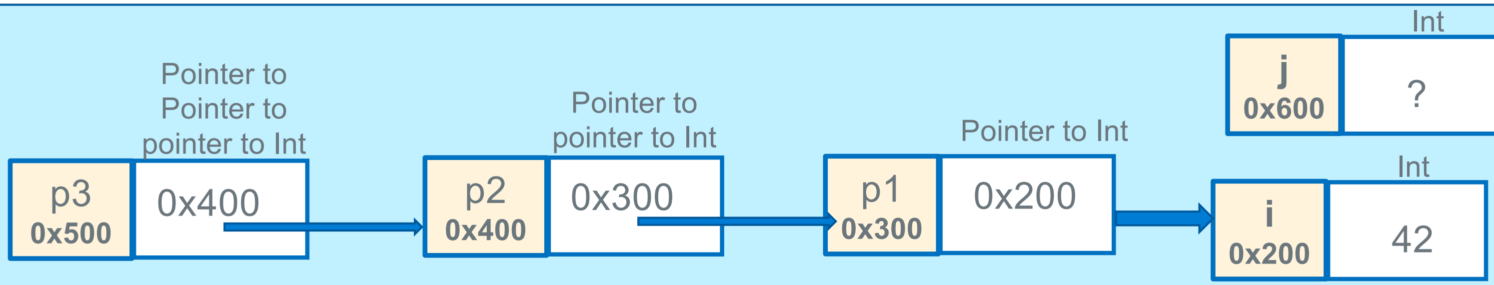
number of * in the definition tells you how many reads it takes to get to the base type
\# reads = number of * + 1
e.g., int **p2 requires 3 reads to get to the int

- C allows any number of pointer indirections
  - more than three levels is very uncommon in real applications as it reduces readability and generates at lot of memory reads

Pointers to Pointers to Pointers…. Rside Practice

```
int i = 42;
int j;
int *p1;
int **p2;
int ***p3;
```

number of ***"reads + 1"*** to *base type on Rside*




```
p1 = &i;
p2 = &p1;
p3 = &p2;

j = *p1 + 1;
**p2 = *p1 + 2;
printf("j:%d i:%d\n",j, i);

j = ***p3 + **p2 + *p1;
printf("j:%d i:%d\n",j, i);
```

Rside evaluations

	Address	Contents	*contents	**contents	***contents
j	0x600	?			
i	0x200	42			
p1	0x300	0x200	42		
p2	0x400	0x300	0x200	42	
p3	0x500	0x400	0x300	0x200	42

```
% ./a.out
j:43 i:44
j:132 i:44
```

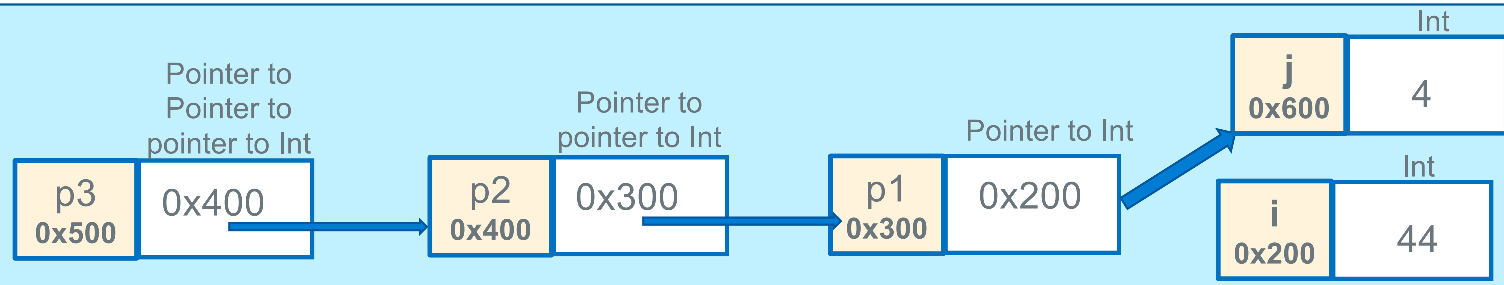
Key:

memory address or name	Memory contents

x

Pointers to Pointers to Pointers…. Rside Practice

```
int i = 42;
int j;
int *p1;
int **p2;
int ***p3;
```



number of ***"reads + 1"*** to *base type on Rside*

Rside evaluations

	Address	Contents	*contents	**contents	***contents
j	0x600	4			
i	0x200	44			
p1	0x300	0x600	4		
p2	0x400	0x300	0x600	4	
p3	0x500	0x400	0x300	0x600	4

```
//continued from previous slide
p1 = &j;
***p3 = 4;
printf("*p1:%d i:%d\n",*p1, i);
```

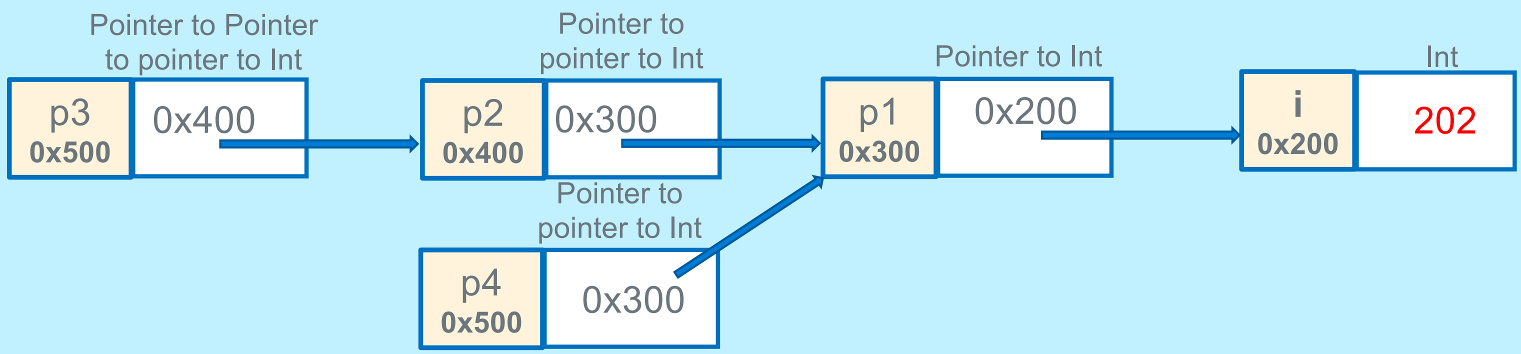
```
% ./a.out
j:43 i:44
j:132 i:44
*p1:4 i:44
```

Key: memory address or name | Memory contents

x

Pointers to Pointers to Pointers…. Lside Practice

```
int i = 42;
int *p1 = &i;
int **p2 = &p1;
int ***p3 = &p2;
p3 = &p2;
int **p4;
p1 = &i;
p2 = &p1;
p3 = &p2;
```




```
*p1 = 34; // +1 read lside

**p2 = 19; //+2 reads lside

***p3 = 101; //+3 reads lside

**p2 = *p1 + i;

p4 = *p3;

(**p4)++;

--(*p1);
```

destination addresses for data write when on Lside

	address	variable	*variable	**variable	***variable
i	0x200	0x200			
p1	0x300	0x300	0x200		
p2	0x400	0x400	0x300	0x200	
p3	0x500	0x500	0x400	0x300	0x200

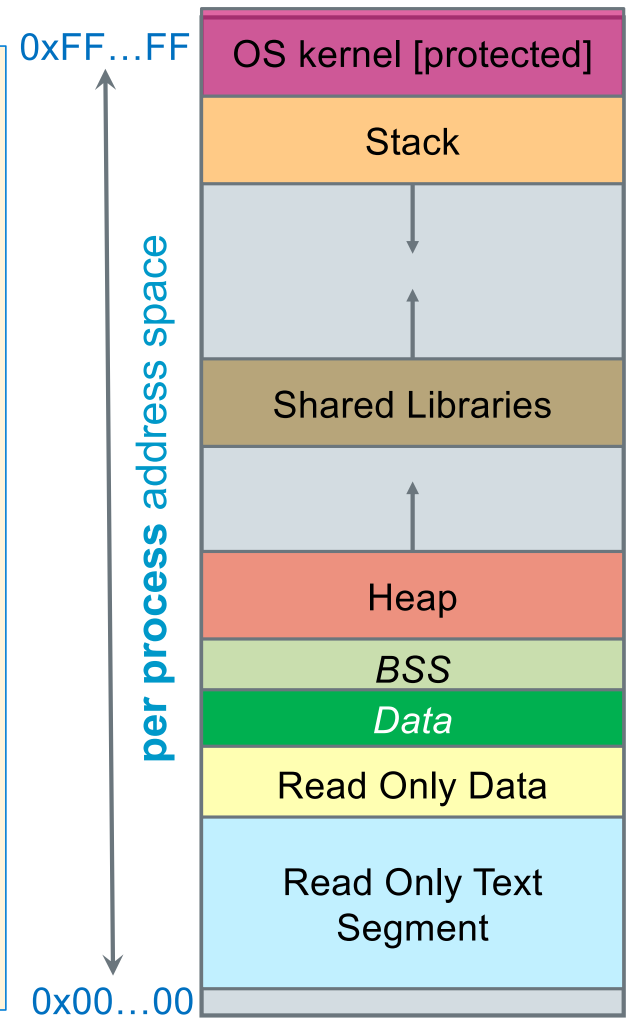
Key: | memory address or name | Memory contents |

(slide is best viewed is in pptx presentation mode)

x

Process Memory Under Linux

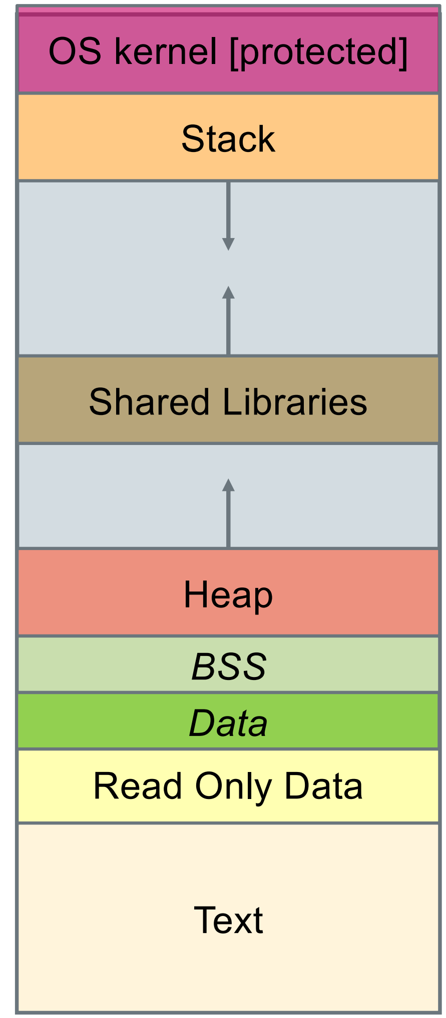
- When your program is running it has been loaded into memory and is called a process
- *Stack segment: Stores Local variables*
  - Allocated and freed at function call entry & exit
- *Data segment + BSS: Stores Global and static variables*
  - Allocated/freed when the process starts/exits
  - BSS - Static variables with an implicit initial value
  - Static Data - Initialized with an explicit initial value
- Heap segment: *Stores dynamically-allocated* variables
  - Allocated with a function call
  - Managed by the stdio library malloc() routines
- *Read Only Data: Stores immutable* Literals
- *Text*: Stores your code in machine language + libraries



x

The Heap Memory Segment

- Heap: "pool" of memory that is available to a program
  - Managed by C runtime library and linked to your code; **not managed by the OS**
- Heap memory is **dynamically** *"borrowed"* *or "allocated"* by calling a library function
- When heap memory is no longer needed, it is *"returned" or deallocated* *for* **reuse**
- Heap memory has a lifetime from allocation until it is deallocated
  - Lifetime is independent of the scope it is allocated in (it is like a static variable)
- If too much memory has already been allocated, the library will attempt to borrow additional memory from the OS and will fail, returning a NULL



x

Heap Dynamic Memory Allocation Library Functions

#include <stdlib.h>	args	Clears memory
void *malloc(…)	size_t size	no
void *calloc(…)	size_t nmemb, size_t memsize	yes
void *realloc(…)	void *ptr, size_size	no
void free(...)	void *ptr	no

- **void *** means these library functions return a pointer to generic (untyped) memory
  - Be careful with void * pointers and pointer math as void * points at untyped memory (not allowed in C, but allowed in gcc). The assignment to a typed pointer *"converts"* it from a void *
- **size_t is** an unsigned integer data type, the result of a sizeof() operator

```
int *ptr = malloc(sizeof(*ptr) * 100); // allocate an array of 100 ints
```

- **please read: % man 3 malloc**

x

Use of Malloc

`void *malloc(size_t size)`

- Returns a pointer to a **contiguous** block of **`size`** bytes **of uninitialized memory** from the heap
  - The block is **aligned to an 8-byte (arm32) or 16-byte (64-bit arm/intel) boundary**
  - returns **`NULL`** if allocation failed (also sets **`errno`**) **always CHECK for NULL RETURN!**
- Blocks returned on different calls to malloc() are not necessarily adjacent
- **`void *`** is implicitly cast into any pointer type on assignment to a pointer variable

```
#include <stdlib.h>                  // need this for malloc() etc
char *getbuf(int cnt)
{
    char *bufptr;
    /* ALWAYS CHECK THE RETURN VALUE FROM MALLOC!!!! */
    if ((bufptr = malloc(cnt * sizeof(*bufptr))) == NULL) {
        fprintf(stderr, "Unable to malloc memory");
        return NULL;
    }
    return bufptr;    // the calling function must free memory
}
```


x

Using and Freeing Heap Memory

- `void free(void *p)`
  - Deallocates the whole block pointed to by **`p`** to the pool of available memory
  - Freed memory is used in future allocation (expect the contents to change after freed)
  - Pointer **`p`** must be the same address as *originally returned* by one of the heap allocation routines **`malloc(),calloc(),realloc()`**
  - Pointer argument to free() is not changed by the call to free()
- Defensive programming: set the pointer to NULL after passing it to free()

```
#define COLCNT 1024
char *ptr, *endptr, *bufptr;

bufptr = getbuf(COLCNT);     // do not lose bufptr!, memory leak
ptr = bufptr;
endptr = ptr + COLCNT;
while (ptr < endptr)
       *ptr++ = 'a';         // fill each array element with 'a'
free(bufptr);                // returns memory to the heap
bufptr = NULL;               // set bufptr to NULL
```


X

Mis-Use of Free()

- Call **`free()`** only with only the same memory returned from the heap
  - It is NOT an error to pass **`free()`** a pointer to NULL
- Continuing to write to memory after you **`free() it`** is likely to corrupt the heap or return changed values
  - Later calls to heap routines (**`malloc()`**, **`realloc()`**, **`calloc()`**) may fail or seg fault

```
char *bytes = malloc(1024 * sizeof(*bytes));
char *ptr = "cse30";
          …
    /* some code */
    free(bytes + 5);        // not ok
    free(ptr);              /* not memory on the heap */
```

```
char *bytes = malloc(1024 * sizeof(*bytes));
…
    /* some code */
    free(bytes);
    strcpy(bytes, "cse30");   // INVALID! used after free
……
```


x

Heap Memory "Leaks"

- A memory leak is when you **allocate memory** on the heap, **but never free it**

```
void
leaky_memory (void)
{
    char *bytes = malloc(BLKSZ * sizeof(*bytes));
…
    /* code that never passes the pointer in bytes to anything */
    return;
}
```

- Your program is responsible for cleaning up any memory it allocates but no longer needs
  - If you keep allocating memory, you may run out of memory in the heap!
- Memory leaks may cause long running programs to fault when they exhaust OS memory limits
  - Make sure you free memory when you no longer need it
- Valgrind is a tool for finding memory leaks (not pre-installed in all linux distributions though!)

x

Dangling Pointers

- When a pointer points to a memory location that is no longer "valid"
- Really hard to debug as the use of the return pointers may not generate a seg fault

```
char *dangling_freed_heap(void)
{
    char *buff = malloc(BLKSZ * sizeof(*buff));
…
    free(buff);
    return buff;
}
```

- `dangling_freed_heap()` type code often causes the allocators (**`malloc()`**and friends) to **seg fault**
  - Because it corrupts data structures the heap code uses to manage the memory pool

x

strdup(): Allocate Space and Copy a String

```
char *strdup(char *s);
```

- **`strdup`** is a function that returns a **null-terminated**, heap-allocated string copy of the provided text
- Alternative: **`malloc`** and copy the string

```
char *str = strdup("Hello, world!");
str[0] = 'h';

free(str);
str = NULL;
```

Calloc()

```
void *calloc(size_t elementCnt, size_t elementSize)
```

`calloc()` variant of **`malloc()`** but zeros out every byte of memory before returning a pointer to it (so this has a runtime cost!)

- First parameter is the number of elements you would like to allocate space for
- Second parameter is the size of each element

```
// allocate 10-element array of pointers to char, zero filled
char **arr;
arr = calloc(10, sizeof(*arr));
if (arr == NULL)
  // handle the error
```

- Originally designed to allocate arrays but works for any memory allocation
  - **`calloc()`** multiplies the two parameters together for the total size
- **`calloc()`** is more expensive at runtime (uses both cpu and memory bandwidth) than **`malloc()`** because it must zero out memory it allocates at runtime
- Use `calloc()` only when you need the buffer to be zero filled prior to FIRST use

x

Realloc

```
void *realloc(void *ptr, size_t size);
```

- **realloc** function takes an existing allocation pointer and enlarges to a new requested size, It returns the new pointer (may be same or different address)
  - If a new buffer, ptr is no longer valid!
- realloc() only accepts pointers that were previously returned my malloc etc.
- Make sure to not lose original pointer if realloc() fails (newstr versus str)

```
char *str = strdup("Hello");
char *addition = " World!\n";
int len = strlen(str);
char *newstr;

if ((newstr = realloc(str, len + strlen(addition) + 1)) != NULL) {
    strcpy(newstr + len, addition);
    printf("%s", newstr);
    free(newstr);
} else         // realloc() failed so free orig buffer
    free(str);
```

Heap Allocation Routine Summary

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
char *strdup(char *s);
void free(void *ptr);
```

Heap **memory allocation** guarantee:

- `NULL` on failure, so check return value
- Memory is returned is contiguous
- it is not recycled unless you call free
- `realloc` preserves existing data
- `calloc` zero-initializes bytes, `malloc` and `realloc` do not

Undefined behavior occurs:

- If you overflow (i.e., you access beyond bytes allocated)
- If you use after `free`, or if `free` is called twice on a location
- If you `realloc`/`free` non-heap address

PA5: getopt() usage- parsing command line Arguments

```
int getopt(int argc, char *argv[], const char *optstring);  / / please see man 3 getopt
```

- Option string describes the option flags: either a letter or a letter followed by
  - Colon (:), the flag requires an argument and then (char ***optarg)** points at the argument

```
% ./extract -c 3 1 2 3
```

```
when processing -c optarg (char *) points at the 3
```

- **`Call getopt()`** in a loop; it returns the next option flag (a char in an int like getchar()):
  - Next command line flag
  - -1 if there are none left
  - '?' indicates flag is not one specified (error) or the flag is specified but has a missing argument
    - **optopt** contains the flag that was detected, but is the cause for the error
- When getopt finishes, **optind** contains the index to the next non-flag argument to process

```
% ./extract -c 3 1 2
```

```
optind = 3;  // int
outcols = argc – optind (# of args after the 3)
```

getopt() sample

- For this example, the options are

1. a single flag **x**
2. a flag **f** with a required argument to the flag

- Additional arguments are not options flags, but filenames to process

% ./a.out –f file.txt a b c

```
optind = 3;  // int
```

```
while ((opt = getopt(argc, argv, "xf:")) != -1) {
     switch (opt) {
     case 'x':
          xFlag = 1;
          printf("-x flag found\n");
          break;
     case 'f':
          datafile =  optarg;   // string (char *)
          printf("-f %s found\n", optarg);
          break;
     case '?':
          if (optopt == 'f')
               fprintf(stderr, "%s -f datafile is missing\n", argv[0]);
          /* fall through */
     default:
          error = 1; /* error = 0 above getopt(); have an error */
          break;
     }
}
if (error != 0) {
     fprintf(stderr, "Usage: %s [-x] -f datafile\n", argv[0]);
     return EXIT_FAILURE;
}
for (int i = optind; i < argc; i++)
      printf("argv[%d] is: %s\n", i, argv[i]);  // additional args
```


X

Struct Variable Definitions

- Variable definitions like any other data type:

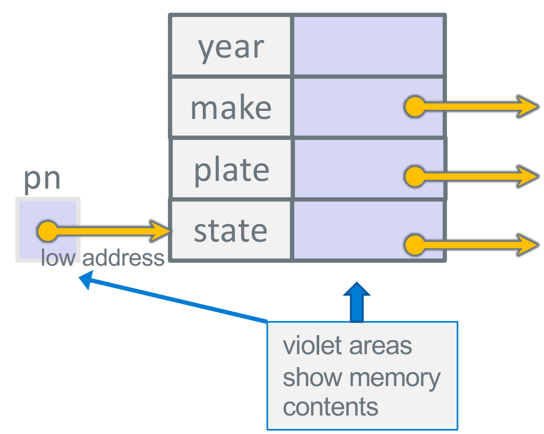
```
struct vehicle name1, *pn, ar[3];
```

type: "struct vehicle"
single variable instance
pointer
array

- Can combine struct and variable definition:
  - This syntax can be harder to read, though

```
struct vehicle {
  char *state;
  char *plate;
  char *make;
  int year;
} name1, *pn = &name1, ar[3];
```

```
struct vehicle {
  char *state;
  char *plate;
  char *make;
  int year;
};
struct vehicle name1;
struct vehicle *pn;
struct vehicle ar[3];
pn = &name1;
```



x

Accessing members of a struct

- Like arrays, struct variables are aggregated contiguous objects in memory
- the . structure operator which *"selects"* the requested field or member

```
struct date {// defining struct type
    int month;
    int day; // members date struct
};
```

```
struct date bday; // struct instance
bday.month = 1;
bday.day = 24;

// shorter initializer syntax
struct date new_years_eve = {12, 31};
struct date final = {.day= 24, .month= 1};
```

day	24
month	1

- Now create a *pointer* to a struct

```
struct date *ptr = &bday;
```

- Two options to reference a member via a struct pointer (. is higher precedence than *):
- Use * and . operators: `(*ptr).month = 11;`
- Use -> operator for shorthand: `ptr->month = 11;`

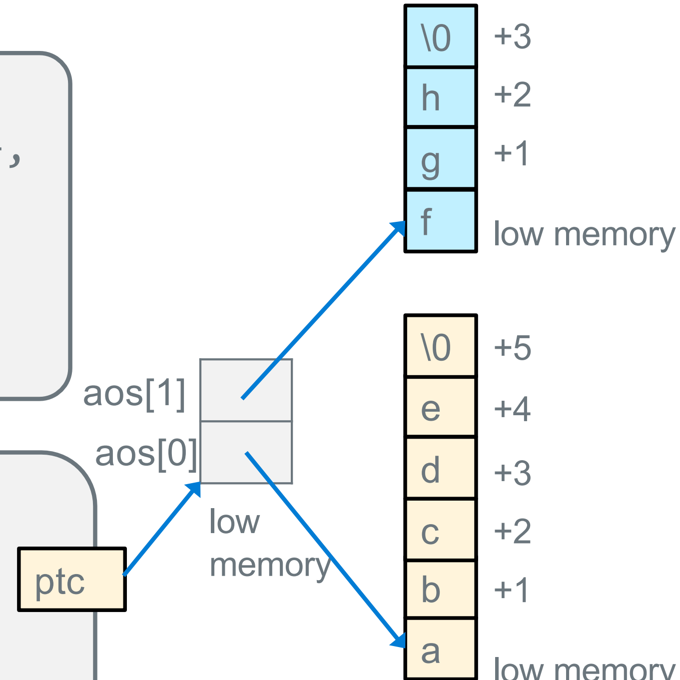
More to come….

Extra Slides

Pointer Array to Mutable Strings

- Make an array of pointers to mutable strings requires using a cast to an array (char [ ])
- Add a NULL sentinel at the end to indicate the end of the array

```
char *aos[] = {
   (char []) {"abcde"},
   (char []) {"fgh"},
   (char *)  {NULL}
};
char **ptc = aos;
```




```
printf("%c\n", *(*(aos + 1) + 1));

while (*ptc != NULL) {
    printf("%s\n", *ptc);     // prints string

    for (int j = 0; *(*ptc + j); j++)
        putchar(*(*ptc + j)); // char in string

    putchar('\n');
    ptc++;
}
```

```
%./a.out
g
abcde
abcde
fgh
fgh
```


x