

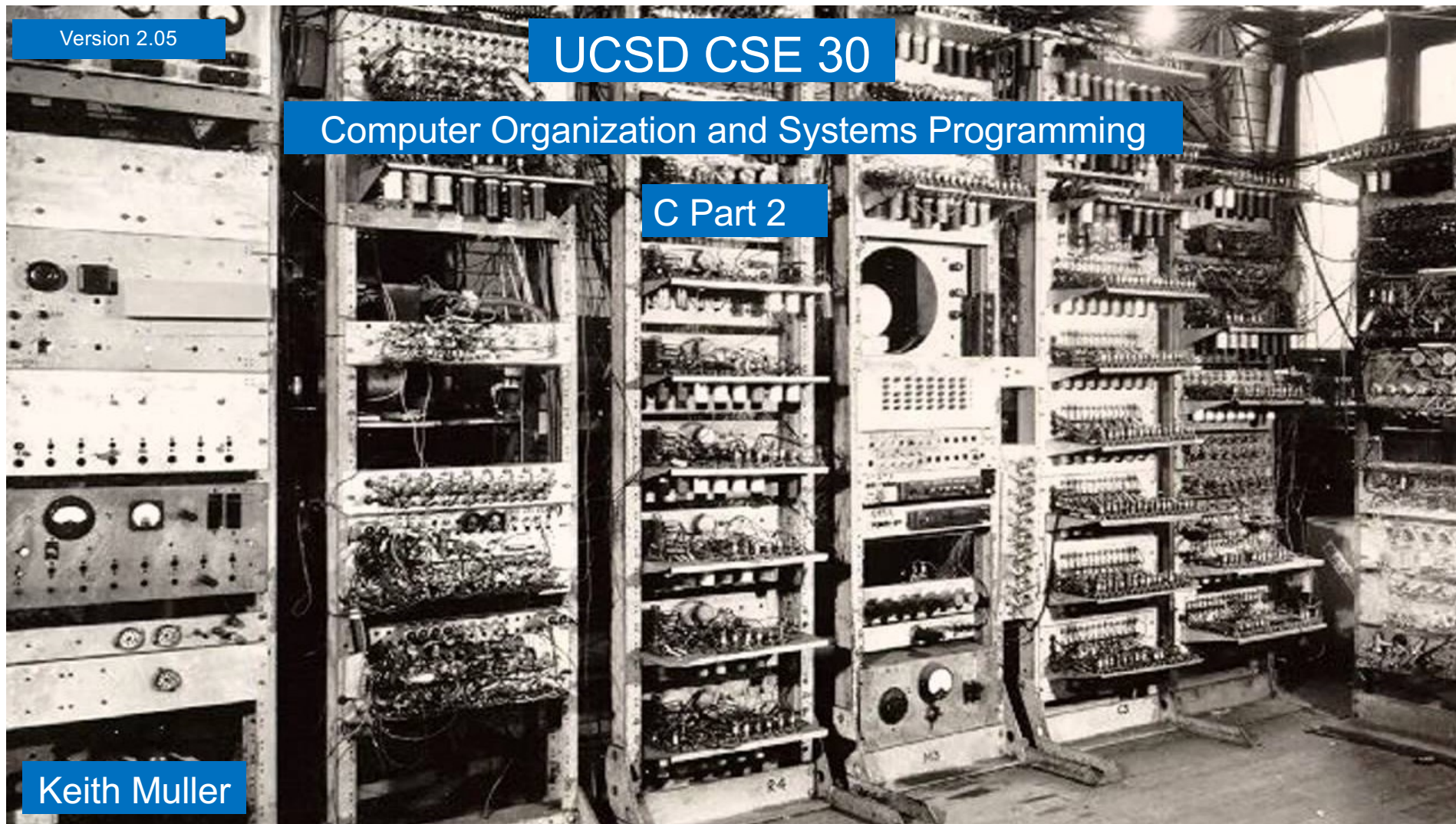
Version 2.05

UCSD CSE 30

Computer Organization and Systems Programming

C Part 2

Keith Muller



Review: Binary Numbering

- Binary is base 2
 - *adjective*: being in a state of one of two **mutually exclusive** conditions such as **on** or off, **true** or **false**, **molten** or **frozen**, **presence** or **absence** of a signal
 - From Late Latin *bīnārius* (“consisting of two”)
- **Two** symbols:
0 1
- Numbers in C that start with **0b** are binary
- Example: What is **0b110** in base 10?
 - $0b110 = 110_2 = (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) = 6_{10}$
- A **bit** is a single binary digit
- A **byte** is an 8-bit value



powers of two

$$\text{Unsigned binary Number} = \sum_{i=0}^{n-1} b_i \times 2^i = b_{n-1}2^{N-1} + b_{n-2}2^{N-2} + \dots + b_12^1 + b_02^0$$

Review: Hexadecimal Numbering

- hexadecimal is base 16
 - From “hexa” (Ancient Greek ἑξά-) \Rightarrow six
 - and from “decem” (Latin) \Rightarrow ten

- Sixteen** symbols

0 1 2 3 4 5 6 7 8 9 a b c d e f



- Numbers in C that start with **0x** are hexadecimal numbers
 - $16_{10} = 0x10_{16}$
- Example: What is **0xa5** in base 10?
 - $0xa5 = a5_{16} = (10 \times 16^1) + (5 \times 16^0) = 165_{10}$
- Hexadecimal numbers are **very commonly used** in programming to express binary values
 - Imagine the difficulty in correctly expressing a 64-bit binary value in your code

$$\text{Unsigned Hex Number} = \sum_{i=0}^{n-1} b_i \times 16^i = b_{n-1}16^{n-1} + b_{n-2}16^{n-2} + \dots + b_116^1 + b_016^0$$

Binary <---> Hexadecimal Equivalences

- **Hex → Binary:** $16^1 = 2^4$ 1 digit hex = 4 digits binary
 1. Replace hex digits with binary digits
 2. Drop **leading zeros**
 - Example: 0x2d to binary
 - 0x2 is 0b0010, 0xd is 0b1101
 - Drop two leading zeros, answer is 0b101101
- **Binary → Hex:** $2^4 = 16^1$
 1. **Pad** with enough **leading zeros** until number of digits is a multiple of 4
 2. **Replace** each **group of 4** with the **HEX equivalent**
 - Example: 0b101101
 - **Pad on the left** to: 0b 0010 1101
 - Replace to get: 0x2d

Number Base Overview (as written in C)

- Decimal is base 10 and Hexadecimal is base 16,
- **Hex digits** have 16 values 0 - 9 a - f (written in C as 0x0 – 0xf)
- No standard prefix in C for binary (most use **hex**)
 - gcc (compiler) allows **0b** prefix **others might not**

| | | | | | | | | |
|---------------|--------|--------|--------|--------|--------|--------|--------|--------|
| Hex digit | 0x0 | 0x1 | 0x2 | 0x3 | 0x4 | 0x5 | 0x6 | 0x7 |
| Decimal value | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Binary value | 0b0000 | 0b0001 | 0b0010 | 0b0011 | 0b0100 | 0b0101 | 0b0110 | 0b0111 |

| | | | | | | | | |
|---------------|--------|--------|--------|--------|--------|--------|--------|--------|
| Hex digit | 0x8 | 0x9 | 0xa | 0xb | 0xc | 0xd | 0xe | 0xf |
| Decimal value | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Binary value | 0b1000 | 0b1001 | 0b1010 | 0b1011 | 0b1100 | 0b1101 | 0b1110 | 0b1111 |

Hex to Binary (group 4 bits per digit from the right)

- Each Hex digit is 4 bits in base 2 $16^1 = 2^4$

0x f a 5 3

1111 1010 0101 0011

0b1111101001010011

↑ binary start with a 0b in C

Binary to Hex (group 4 bits per digit from the right)

- 4 binary bits is one Hex digit $2^4 = 16^1$

0b 0110 1010 0011 1111
 └─┬─┘ └─┬─┘ └─┬─┘ └─┬─┘
 6 a 3 f

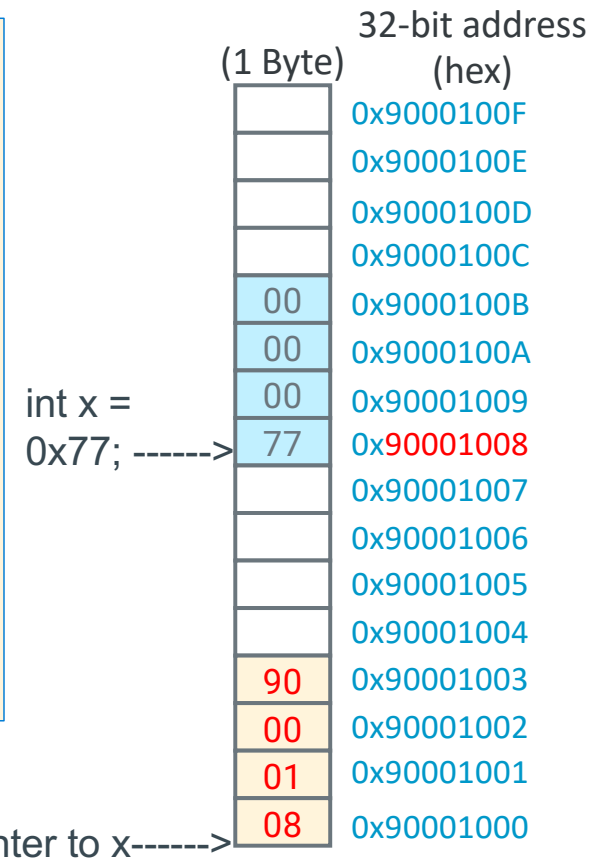
0x6a3f

hex start with 0x in C



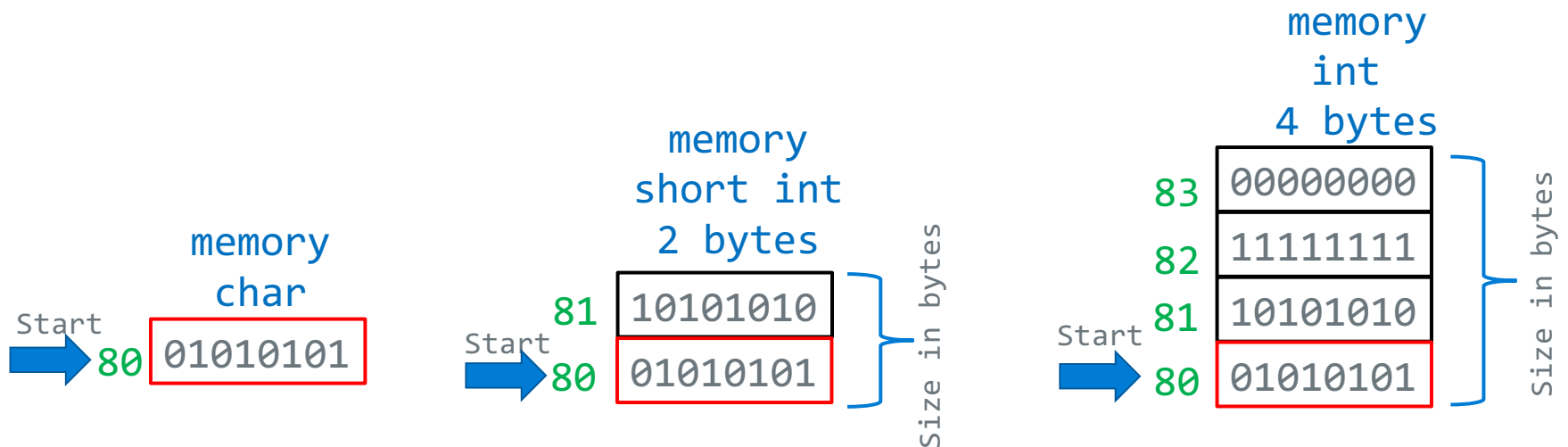
Address and Pointers

- An **address** refers to a location in memory, the **lowest** or **first byte** in a **contiguous sequence of bytes**
- A **pointer** is a **variable** whose **contents** (or value) can be properly used as an **address**
 - The **value in a pointer** *should* be a **valid address allocated to the process** by the **operating system**
- The **variable x** is at **memory address 0x90001008**
- The **variable pt** is at **memory location 0x90001000**
- The **contents** of **pt** is the **address of x 0x90001008**



Variables in Memory: Size and Address

- The number of contiguous bytes a variable uses is based on the *type* of the variable
 - Different variable types require different numbers of contiguous bytes
- **Variable names** map to a starting address in memory
- **Example Below:** Variables all starting at address 0x80, each box is a byte



Variables: Size

- Integer types

- `char`, `int`

- Floating Point

- `float`, `double`

- Modifiers for each base type

- `short` [int]
- `long` [int, double]
- `signed` [char, int]
- `unsigned` [char, int]
- `const`: variable read only

- char type

- One byte in a byte addressable memory
- **Signed** vs **Unsigned** Char implementations
- **Be careful** `char` is unsigned on arm and signed on other HW like intel

| C Data Type | AArch-32 contiguous Bytes | AArch-64 contiguous Bytes | printf specification |
|----------------------------------|---------------------------------|---------------------------------|-------------------------|
| <code>char</code> (arm unsigned) | 1 | 1 | %c |
| <code>short int</code> | 2 | 2 | %hd |
| <code>unsigned short int</code> | 2 | 2 | %hu |
| <code>int</code> | 4 | 4 | %d / %i |
| <code>unsigned int</code> | 4 | 4 | %u |
| <code>long int</code> | 4 | 8 | %ld |
| <code>long long int</code> | 8 | 8 | %lld |
| <code>float</code> | 4 | 4 | %f |
| <code>double</code> | 8 | 8 | %lf |
| <code>long double</code> | 8 | 16 | %Lf |
| <code>pointer *</code> | 4 | 8 | %p |

size of a pointer is the word size

sizeof(): Variable Size (number of bytes) Operator

```
#include <stddef.h>
/* size_t type may vary by system but is always unsigned */
```

sizeof() operator returns a value of type **size_t**:

the number of bytes used to store a variable or variable type

```
size_t size = sizeof(variable_type);
```

or

```
size_t size = sizeof(variable_name); // preferred!
```

- The argument to sizeof() is often an expression:

```
size = sizeof(int * 10);
```

- reads as:

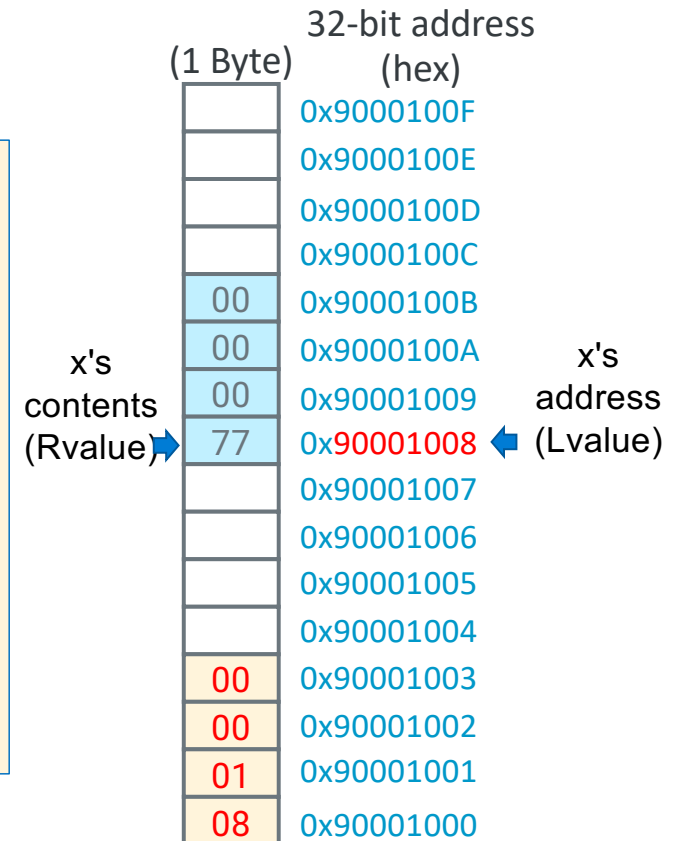
- number of bytes required to store **10 integers (an array of [10])**

Memory Addresses & Memory Content

Variable names in a C statement evaluation

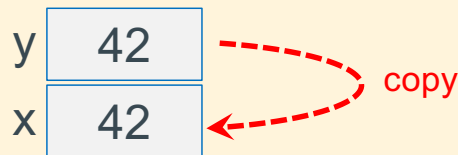
```
x = x + 1;    // Lvalue = Rvalue
```

- **Lvalue:** when on the left side (Lside or Left value) of the = sign
 - address where it is stored in memory – a constant
 - **Address assigned to a variable cannot be changed at runtime**
 - Does not require a memory read
 - **Lside Must evaluate to an address**
- **Rvalue:** when on the right side (Rside or Right value) of an = sign
 - contents or value stored in the variable (at its memory address)
 - requires a memory read to obtain contents



Memory Addresses & Memory Content

`y = 42;` One memory write required
`x = y;` One memory read required
 // Lvalue = Rvalue



- **x** on left side (**Lside**) of the assignment operator = evaluates to:
 - **Address** of the memory assigned to the **x** – this is x's **Lvalue**
- **y** on right side (**Rside**) of the assignment operator = evaluates to:
 - **Contents** of the memory assigned to the variable **y** (type determines length – number of bytes) - this is y's **Rvalue**
- So, `x = y;` is:

Read memory at **y** (**Rvalue**); write it to memory at **x's** address (**Lvalue**)

Introduction: Address Operator: &

- Unary **address operator (&)** produces the **address** of where an **identifier** is in memory
 - Print assigned address to **g**
- **Example** this might print:
value of g is: 42
address of g is: 0x71a0a0
(the address will vary)
- **Tip:** printf() format specifier to display an address/pointer (in hex) is "%p"

```
int main(void)
{
    int g = 42;

    printf("value of g is: %d\n", g);
    printf("address of g is: %p\n", &g);
    return EXIT_SUCCESS;
}
```


Introduction: Address Operator: &

- Requirement: **identifier must have a Lvalue**
 - Cannot be used with **constants** (e.g., 12) or **expressions** (e.g., x + y)
 - Example: **&12** does not have an *Lvalue*,
 - so, **&12** is not a legal expression
- How can I get an **address for use on the Rside**?
 - **&var** (any variable identifier or name)
 - **function_name** (name of a **function**, not func());
 - **&func_name** is equivalent
 - **array_name** (name of the **array** like array_name[5]);
 - **&array_name** is equivalent

Pointer Variables

- In C, there is a *variable type* for **storing an address**: a *pointer*
 - **Contents** of a pointer is an unsigned (positive numbers) memory address

```
type *name; // defines a pointer; name contains address of a variable of type
```

- A *pointer* is defined by placing a *star* (or *asterisk*) (*) before the identifier (name)
- You also must specify the *type of variable* to which the pointer points
- **Pointers are typed!** Why?
 - The compiler needs to know the *size* (sizeof()) of the data **you are pointing at** (number of consecutive bytes to access) to use (dereference) the pointer
- When the **Rside of a variable** contains a **memory address**, (it **evaluates** to an **address**) the variable is called a **pointer variable**

Pointer Variables - 2

- A pointer cannot 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
- Pointer variables all use the **same amount of memory** no matter what they point at

```
int *iptr;  
char *cptr;  
  
printf("iptr(%u) cptr(%u)\n", sizeof(iptr), sizeof(cptr));
```

- Above prints on a 32-raspberry pi

```
% ./example  
iptr(4) cptr(4)
```

Defining Pointer Variables

- Assigning a value to a pointer:

```
int *p = &i;  /* p points at i (assign address i to p) */
```

- Is the same as writing the following definition and assignment statements

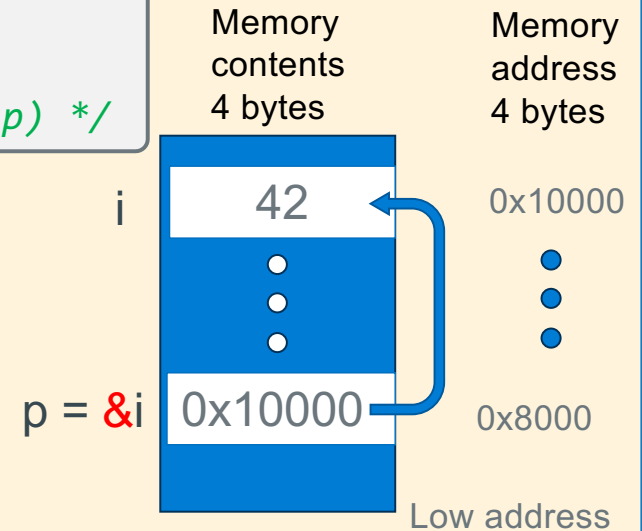
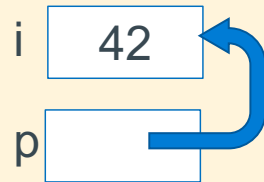
```
int *p;      /* p is defined (not initialized) */  
p = &i;      /* p points at i (assign address of i to p) */
```

- The ***** is part of the definition of **p** and is not part of the variable name
 - The name of the variable is simply **p**, not ***p**
- C mostly ignores whitespace, so these three definitions are equivalent

```
int  *p = &i;    /* Style A */  
int *  p = &i;    /* Style B */  
int*  p = &i;    /* Style C */
```

Using Pointer Variables and the Address Operator & - 1

```
int i = 42;  
int *p;  /* p contains the address of an integer */  
p = &i;  /* p "points at" i (assign address of i to p) */
```



- **Warning:** be careful when defining multiple pointers on the same line:

`int *p1, p2;` is not the same as: `int *p1, *p2;`

Some find this clearer instead:

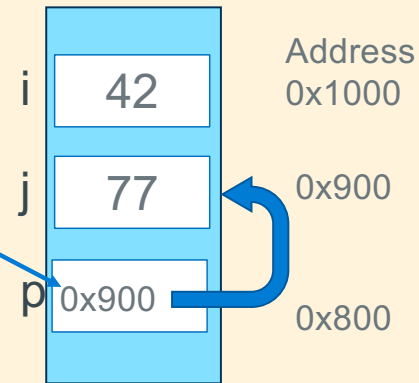
```
int *p1;  
int *p2;
```

Using Pointer Variables and the Address Operator & - 2

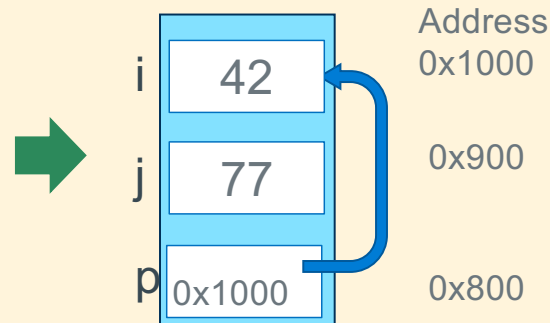
- As with any variable, you can change a pointers value (contents)

`p = &j;` */* p now points at j */*

See that p's value
(contents) is the
address of i



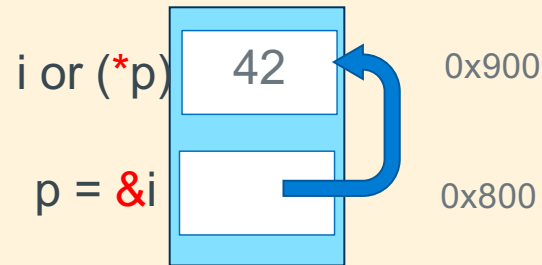
`p = &i;` */* p now points at i */*



Indirection (or dereference) Operator: *

- The **indirection operator** (*) or the *dereference operator to a variable* is the **inverse** of the *address operator* (&)
- **address operator** (&) can be thought of as:

"get the address of this box"



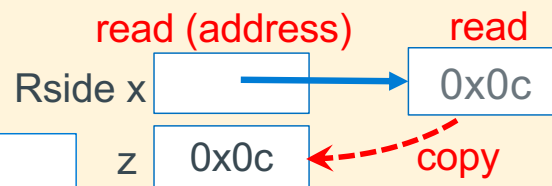
- **indirection operator** (*) can be thought of as:
- "follow the arrow to the next box and get its contents"*
- **Indirection operator causes an additional read to occur**, when on either the Rside or Lside of a statement

Rside Indirection (or dereference) Operator: *

- Performs the following steps when the * is on the Rside:
 1. read the contents of the variable to get an address
 2. read and return the contents at that address
 - (requires two reads of memory on the Rside)

```
z = *x; // copy the contents of memory pointed at by x to z
```

Two reads here
(1) read to get an address
(2) read the address to get the value



Rside Indirection (or dereference) Operator: *

*Contents of **p** is the address of **i***
(*p points at i*)

```
int i = 42;  
int *p;
```

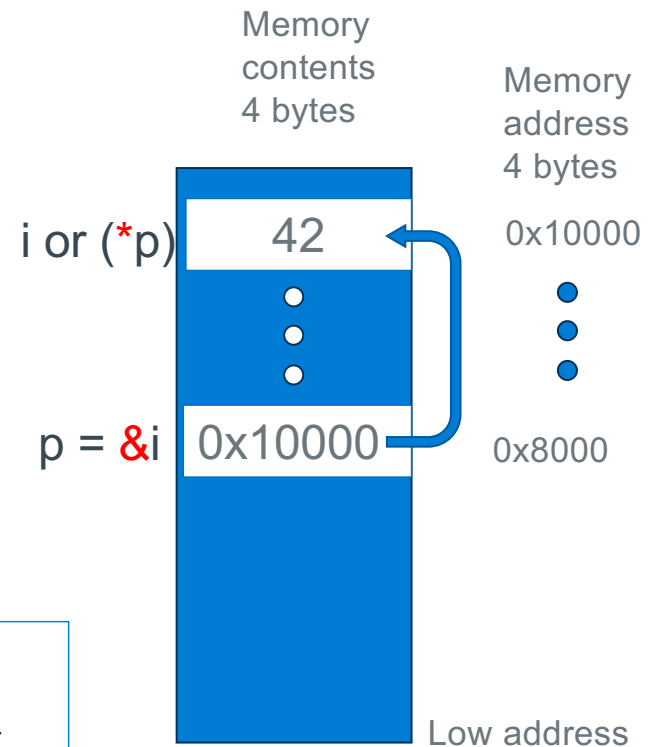
```
p = &i;
```

No reads here

```
printf("*p is %d\n", *p);
```

```
% ./a.out  
*p is 42
```

Two reads here
(1) read to get an address
(2) read the address to get the value

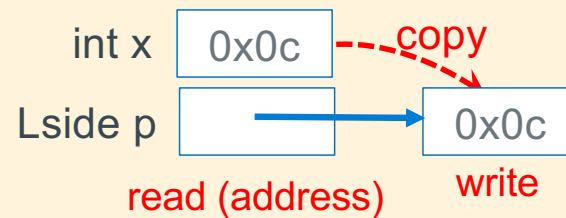


Lside Indirection Operator

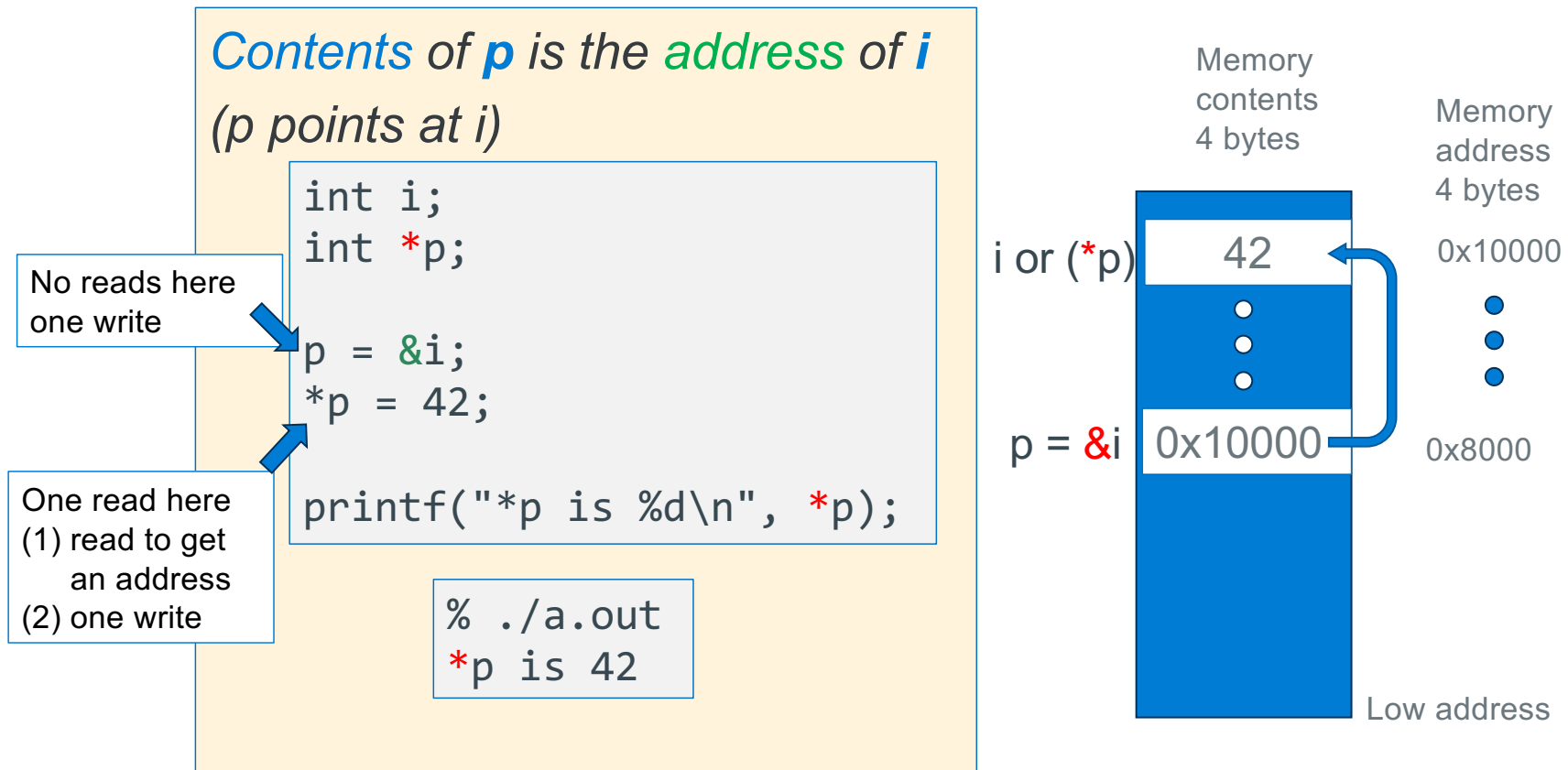
Performs the following steps when the ***** is on the Lside:

1. **read** the **contents** of the **variable** to get **an address**
2. **write** the evaluation of the Rside expression to that address
 - (requires **one read of memory and one write of memory on the Lside**)

```
*p = x; // copy the value of x to the memory pointed at by p
```

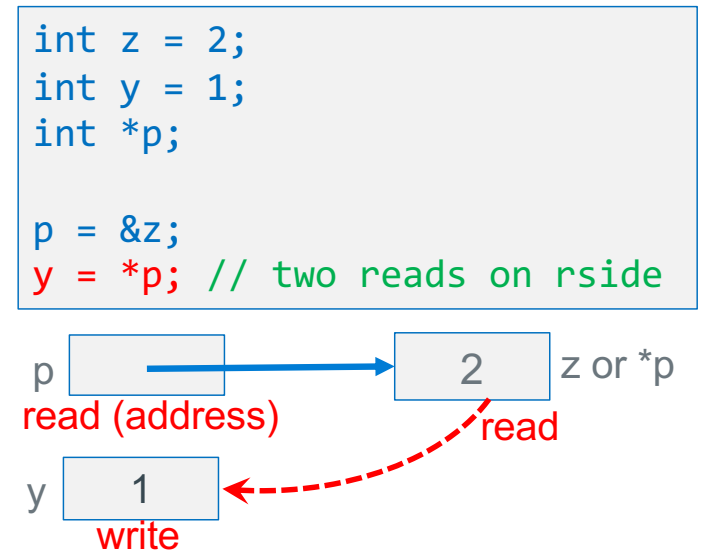
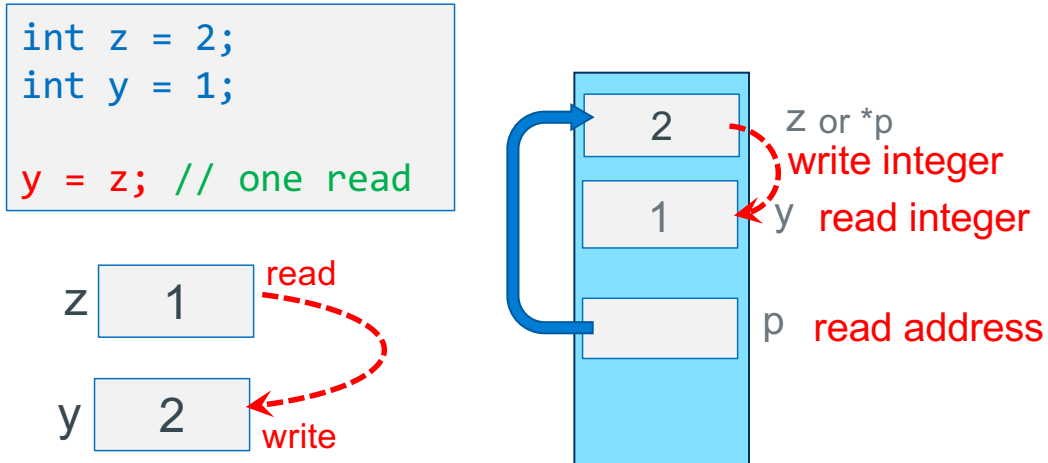


Left Side Indirection (or dereference) Operator: *



Each use of a * operator results in one additional read: Rside

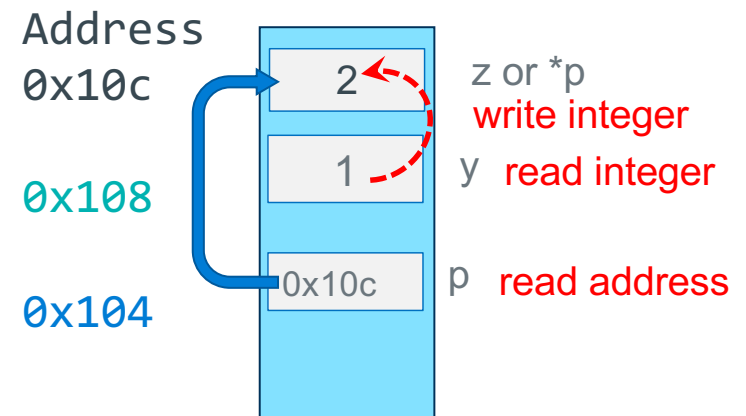
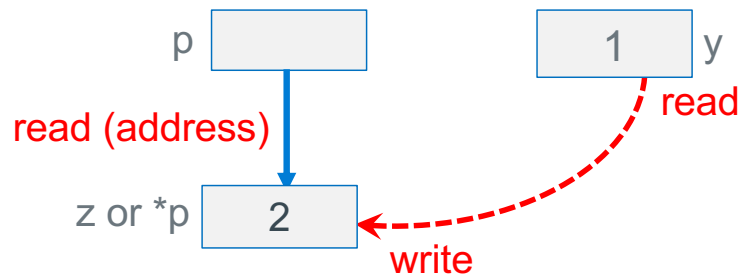
RULE: Each * when used as a dereference operator in a **statement** (either **Lside** or **Rside**) it causes an additional read to be performed



Aside: `y = *(&z);` // same as `y = z`

Each use of a * operator results in one additional read: Lside

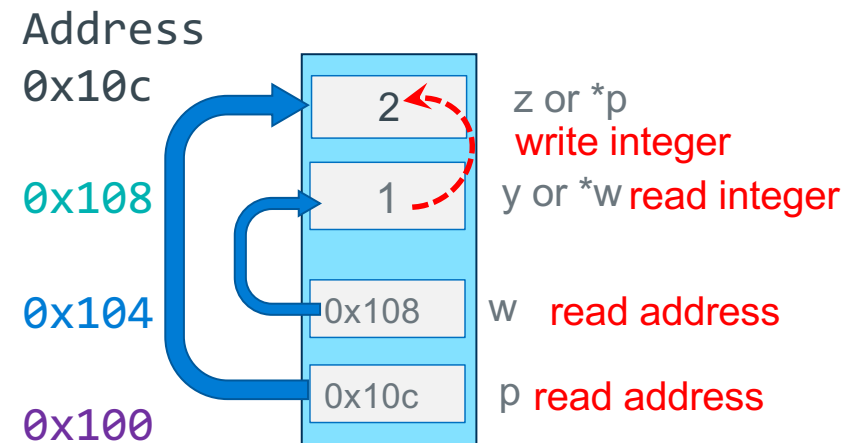
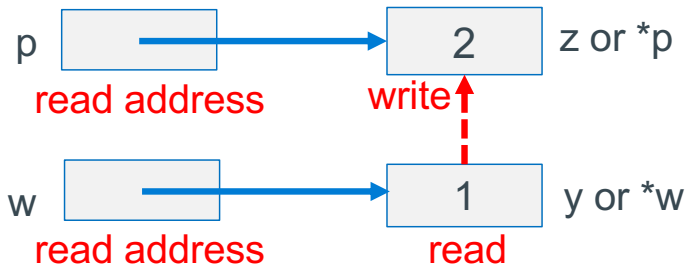
```
int z = 2;  
int y = 1;  
int *p;  
  
p = &z;  
*p = y;    // one read on lside
```



Each use of a * operator results in one additional read : both sides

```
int z = 2;  
int y = 1;  
int *w;  
int *p;
```

```
p = &z;  
w = &y;  
*p = *w;
```

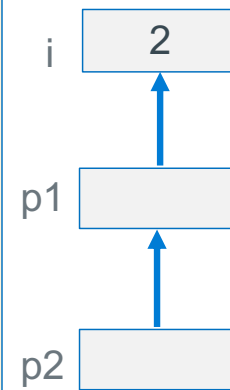


Pointer to Pointers (Double Indirection)

- Define a pointer to a pointer (p2 below)

```
int i = 2;  
int *p1;  
int **p2; // pointer to a pointer to an int  
  
p1 = &i;  
p2 = &p1;  
printf("%d\n", (**p2) * (**p2));
```

- C allows any number of pointer indirections
 - more than two levels is very uncommon in real applications as it reduces readability and generates a lot of memory reads
- RULE (important):** number of ***** in the variable definition tells you how many **reads** it takes to get to the **base type**
 $\text{\#reads to base type} = \text{number of } * \text{ (in the definition)} + 1$
- Example:
`int **p2;` // requires 3 reads to get to the int

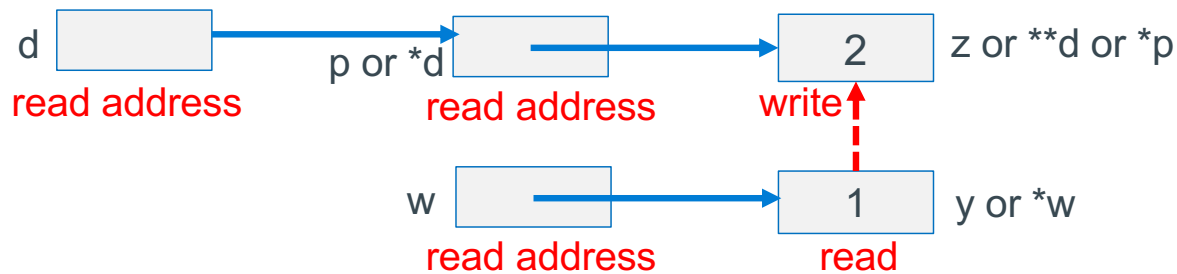
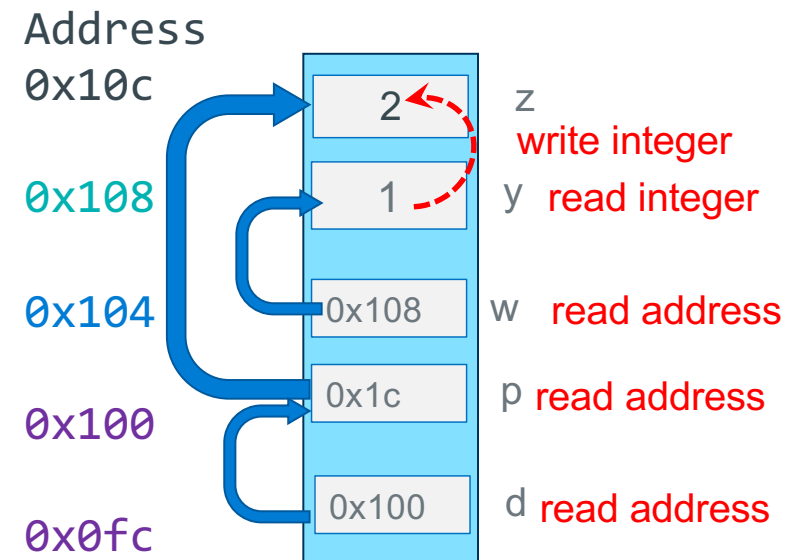


Double Indirection: Lside

```

int z = 2;
int y = 1;
int *w;
int *p;
int **d;

p = &z;
w = &y;
d = &p;
**d = *w;
    
```



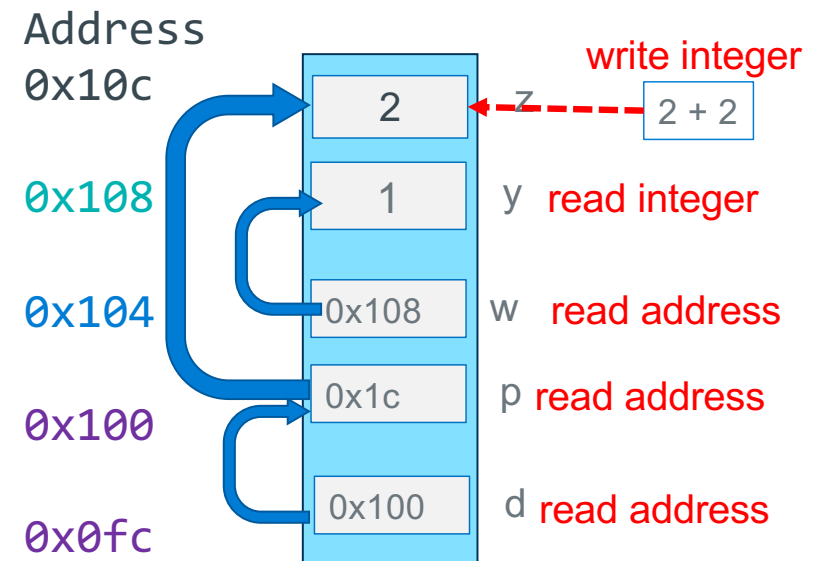
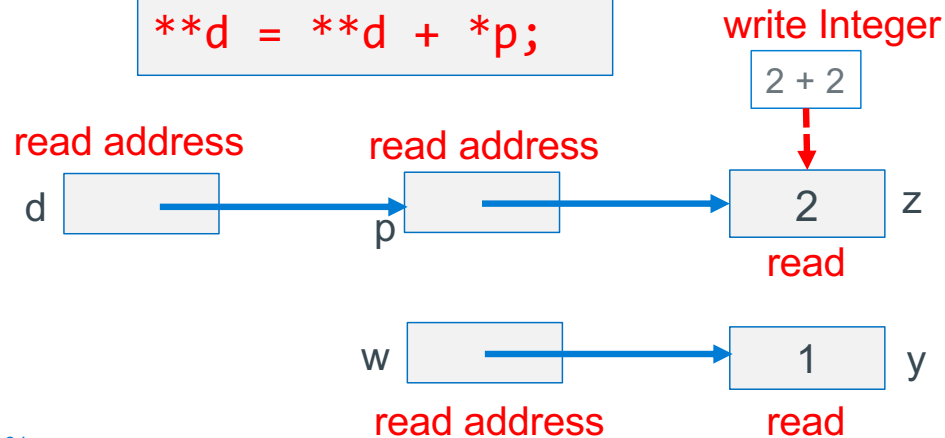
Double Indirection: Rside

```

int z = 2;
int y = 1;
int *w;
int *p;
int **d;

p = &z;
w = &y;
d = &p;

**d = **d + *p;
    
```



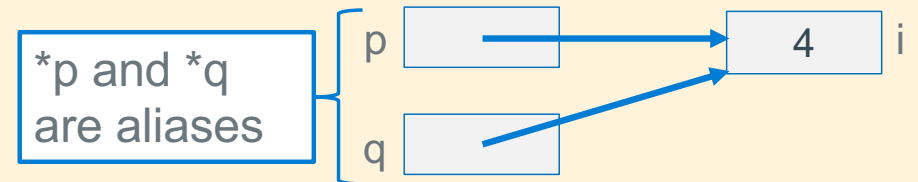
Important Observation
`**d` on Lside is two reads
`**d` on Rside is three reads

What is Aliasing?

- **Two or more** variables are **aliases** of each other when they all reference the same memory (so different names, same memory location)
- **Example:** When one pointer is copied to another pointer it *creates an alias*
- **Side effect:** Changing one variables value (content) changes the value for other variables
 - Multiple variables all read and write the same memory location
 - Aliases occur either by **accident** (coding errors) or **deliberate** (careful: readability)

```
int i = 5;
int *p;
int *q;

p = &i;
q = p;    // *p & *q now aliases
*q = 4;   // changes i and *p
```



Result *p, *q and i all have the value of 4

Defining Arrays

Definition: `type name[count]`

- **"Compound"** data type where each value in an array is an element of `type`
- Allocates **name** with a *fixed* `count` array elements of type `type`
- Allocates $(\text{count} * \text{sizeof}(\text{type}))$ bytes of *contiguous memory*
- Common usage is to specify a compile-time constant for `count`

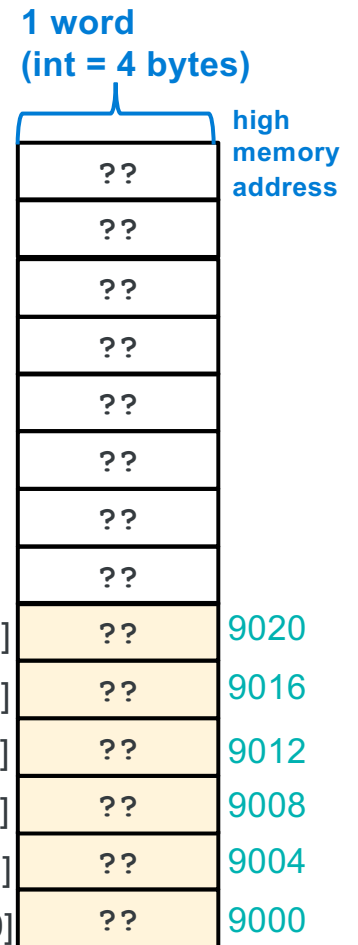
```
#define BSZ 6  
int b[BSZ];
```

BSZ is a macro replaced by the C preprocessor

- Array **names are constants** and **cannot be assigned** (the name cannot appear on the Lside by itself)

```
int a [BSZ];  
a = b;          // invalid does not copy the array  
                // must copy arrays element by element
```

```
int b[6];
```



x

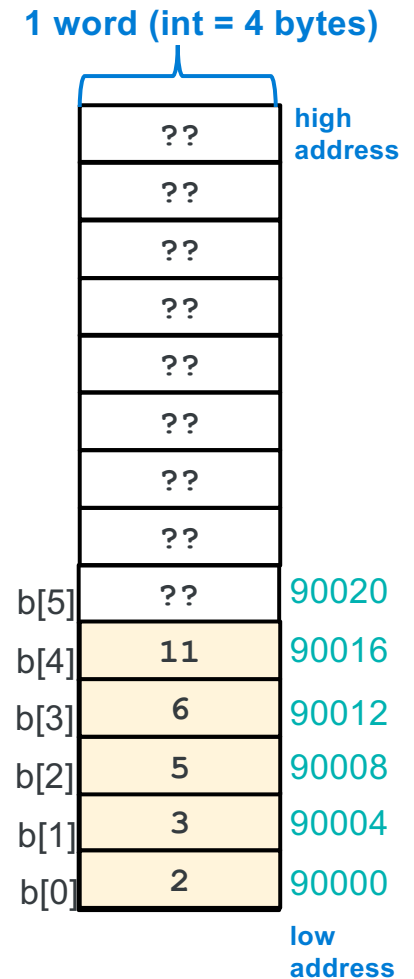
Array Initialization

- Initialization: `type name[count] = {val0,...,valN};`
 - `{ }` (*optional*) initialization list can only be used at **time of definition**
 - If no `count` supplied, `count` is determined by compiler using the number of array initializers
- `no initialization values given; then elements are initialized to 0`
- `int block[20] = {};` //only works with constant size arrays
 - defines an **array of 20 integers** each element filled with zeros
 - Performance comment: do not zero automatic arrays unless really needed!
- When a `count` is given:
 - extra initialization values are **ignored**
 - missing initialization values are set to **zero**

```
int block[5] = {2, 3, 5, 6, 11, 13};
```

not needed and if used **may** truncate initialization list

6 initialization values given, **only 5 are used**



X

Accessing Arrays Using Indexing

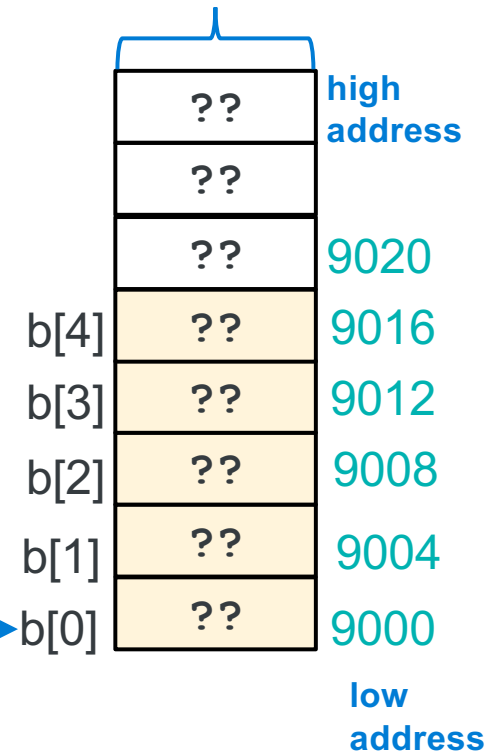
- **name** [**index**] selects the **index** element of the array
 - index **should be** unsigned
 - Elements range from: 0 to count – 1 (int x[count];)
- **name** [**index**] can be used as an **assignment target** or as a **value in an expression**

```
int a[2] = {1, 2};  
a[0] = a[1];
```
- **Array name** (by itself with no []) on the **Rside** evaluates to the address of the first element of the array

```
int b[5];  
int *p = b;
```

p 9000

1 word
(int = 4 bytes)



How many elements are in an array?

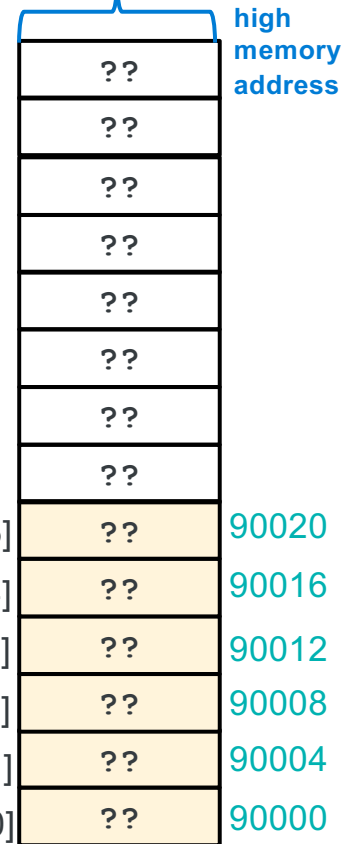
- The number of elements of space allocated to an array (called **element count**) and indirectly the total size in bytes of an array is not stored anywhere!!!!!!
- An **array name** is just the **address of the first element in a block of contiguous memory**
 - So, an array does not know its own size!

```
#define SZ 6
int block[SZ];      // you specify the array has SZ elements
int indx;           // use when SZ is defined

for (indx = 0; indx < SZ; indx++)
    block[indx] = 0;
```

```
int b[6];
```

1 word
(int = 4 bytes)



Determining Element Count: compile time calculation

- Programmatically determining the element count in a compiler calculated array
`sizeof(array) / sizeof(of just one element in the array)`
- `sizeof(array)` only works when used in the SAME **scope** where the array variable was defined

```
#include <stddef.h>
int main()
{
    int block[] =
        {2, 3, 5, 6, 11, 13};    // automatic: compiler calculates array size

    int cnt = (int)(sizeof(block) / sizeof(block[0]));    // in this case cnt = 6

    for (int indx = 0; indx < cnt; indx++)
        block[indx] = 0;
```

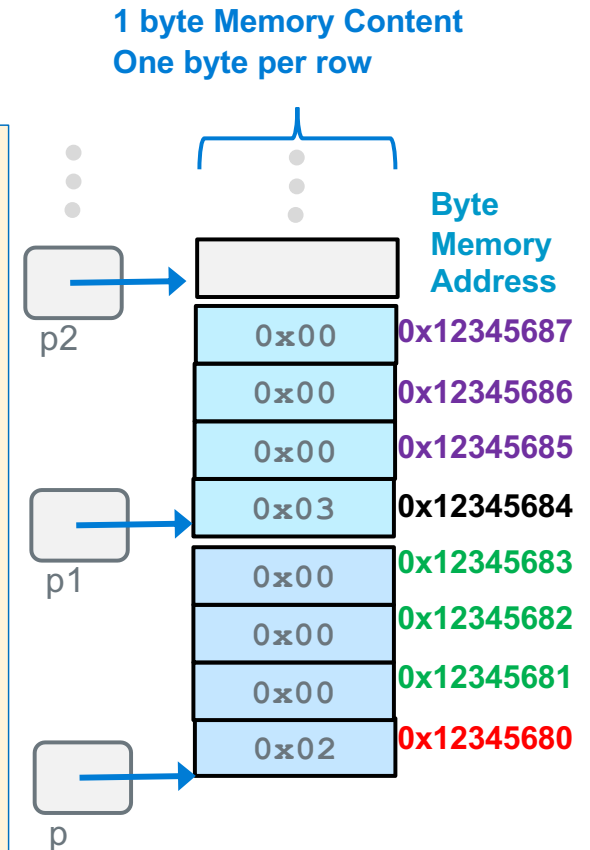
Pointers and Arrays - 1

- A few slides back we stated: **Array name** (by itself) on the Rside evaluates to the **address of the first element of the array**

```
int buf[] = {2, 3, 5, 6, 11};
```

- Array indexing syntax (`[]`) an operator that performs *pointer arithmetic*
- buf** and **&buf[0]** on the **Rside** are **equivalent**, **both evaluate** to the address of the first array element

```
int *p = buf;           // or int *p = &buf[0];  
int *p1 = &buf[1];  
int *p2 = &buf[2];  
int *p3 = &buf[3];
```



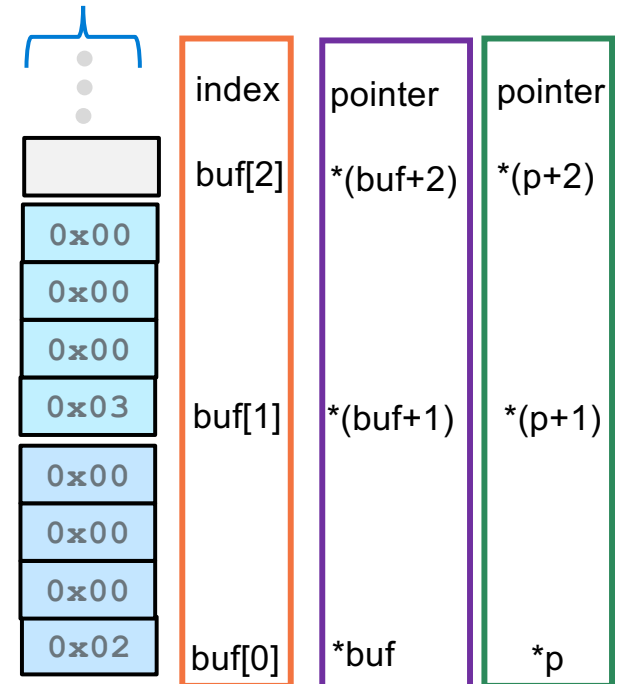
Pointers and Arrays - 2

- When `p` is a pointer, the actual evaluation of the address:
 - `(p+1)` depends on the base type the pointer `p` points at
- `(p+1)` adds `1 x sizeof(what p points at)` bytes to `p`
 - `++p` is equivalent to `p = p + 1`
- Using pointer arithmetic to find array elements:
 - Address of the second element `&buf[1]` is `(buf + 1)`
 - It can be referenced as `*(buf + 1)` or `buf[1]`

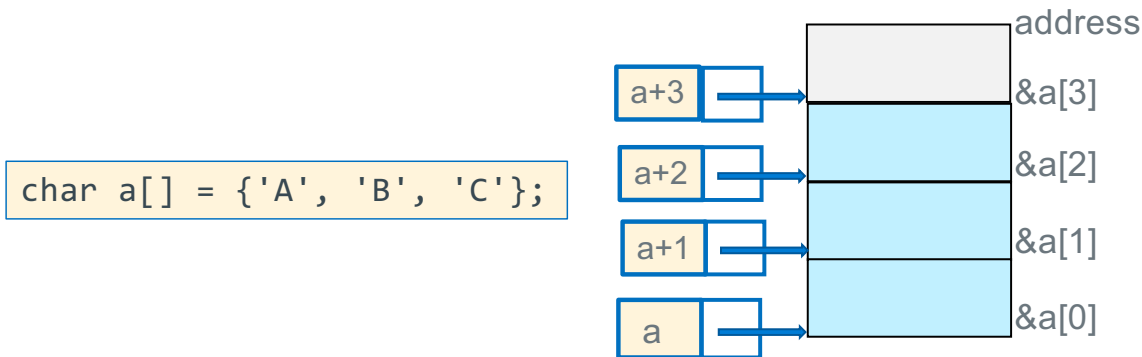
```
int buf[] = {2, 3, 5, 6, 11};
int *p;
p = buf;

*p = *p + 10; // {12, 3, 5, 6, 11}
*(p + 1) = *(buf + 1) + 10; // {12, 13, 5, 6, 11}
```

1 byte Memory Content
One byte per row



Pointer Arithmetic In Use – C's Performance Focus



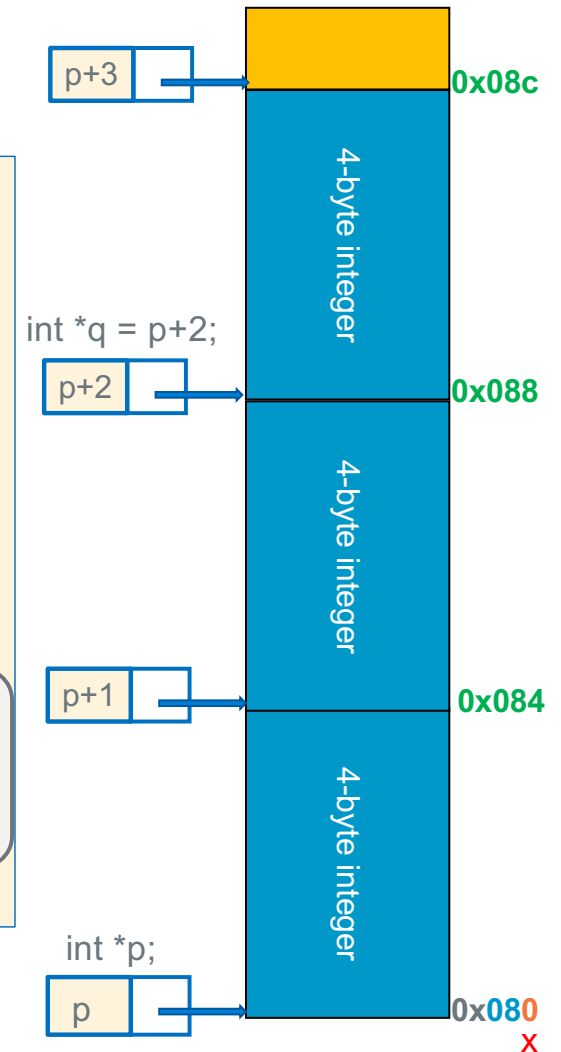
- **Alert!**: C performance focus does not perform any array “bounds checking”
- **Performance by Design**: bound checking slows down execution of a properly written program
- **Example**: array **a** of length **i**, C does not verify that **a[j]** or ***(a + j)** is valid (does not check: $0 \leq j < i$)
 - C simply “*translates*” and accesses the memory specified from: **a[j]** to be ***(a + j)** which may be *outside the bounds* of the array
 - OS only “*faults*” for an incorrect access to memory (read-only or not assigned to your process)
 - It does not fault for out of bound indexes or out of scope
- **Lack of bound checking** is a common source of errors and bugs and is a common criticism of C

Pointer Arithmetic

- You cannot add two pointers (*what is the reason?*)
- A pointer *q* can be subtracted from another pointer *p* when the pointers are the same type – **best done only within arrays!**
- The value of $(p - q)$ is the number of **elements between** the two pointers
 - Using memory address arithmetic (*p* and *q* are both **byte addresses**):

distance in elements = $(p - q) / \text{sizeof}(*p)$

$(p + 3) - p = 3 = (0x08c - 0x080) / 4 = 3$



Pointer Comparisons

- Pointers (**same type**) can be compared with the comparison operators:

<, <=, ==, !=, >=, >

```
int numb[] = {9, 8, 1, 9, 5};
int *end;
int *a;
end = numb + (int) (sizeof(numb)/sizeof(*numb));
a = numb;
while (a < end) // compares two pointers (address)
    /* rest of code including doing an a++ */
```

- Invalid, Undefined, or **risky** pointer arithmetic (some examples)
 - Add, multiply, divide on two pointers
 - Subtract two pointers of different types or pointing at different arrays
 - Compare two pointers of different types
 - Subtract a pointer from an integer

Using Pointers to Traverse an array

```
int x[] = {0xd4c3b2a1, 0xd4c3b200, 0x12345684};
int cnt = (int)(sizeof(x) / sizeof(*x));

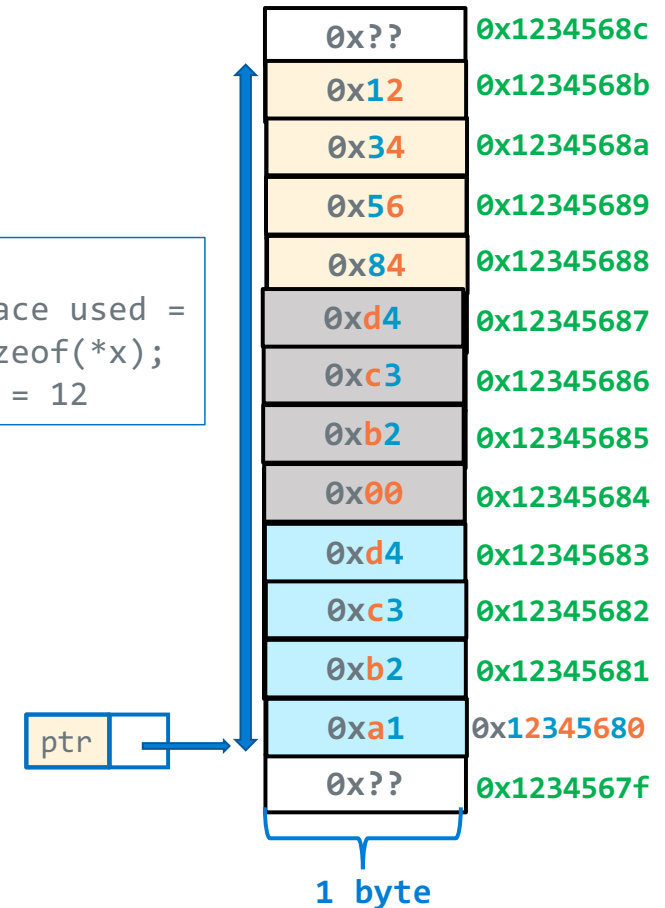
for (int j = 0; j < cnt; j++)
    printf("%#x\n", x[j]);
}
```

```
cnt = 3;
actual space used =
cnt * sizeof(*x);
= 12
```

```
int x[] = {0xd4c3b2a1, 0xd4c3b200, 0x12345684};
int cnt = (int)(sizeof(x) / sizeof(*x));
int *ptr = x;           // or &x[0]

for (int j = 0; j < cnt; j++)
    printf("%#x\n", *(ptr + j));
}
```

Brute force translation to pointers



Fast Ways to Traverse an Array: Use a Limit Pointer

```
int x[] = {0xd4c3b2a1, 0xd4c3b200, 0x12345684};
int cnt = (int)(sizeof(x) / sizeof(*x));
```

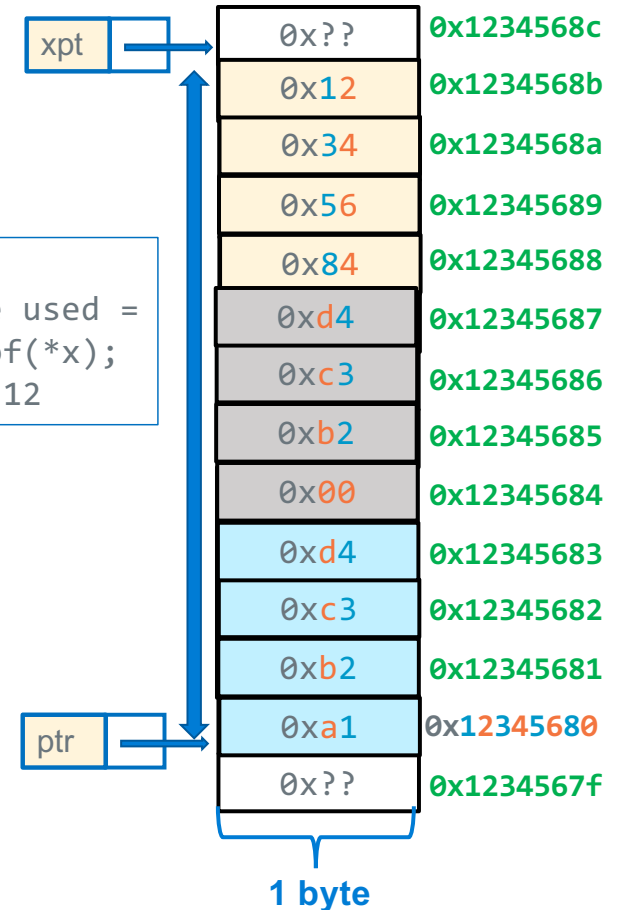
```
int *ptr;
int *xptr;
ptr = x; //or &x[0]
xptr = ptr + cnt;
```

xptr is a **loop limit pointer**
it points **1 element past**
the end of the array

```
while (ptr < xptr) {
    printf("%#x\n", *ptr);
    ptr++;
}
```

```
% ./a.out
0xd4c3b2a1
0xd4c3b200
0x12345684
```

cnt = 3;
actual space used =
cnt * sizeof(*x);
= 12



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 |

| common | With Parentheses | Meaning |
|--------|------------------|--|
| *p++ | *(p++) | (1)The Rvalue is the object that p points at (2)increment pointer p to next element ++ is higher than * |
| (*p)++ | | (1)Rvalue is the object that p points at (2)increment the object |
| *++p | *(++p) | (1)Increment pointer p first to the next element (2)Rvalue is the object that the incremented pointer points at |
| ++*p | ++(*p) | Rvalue is the incremented value of the object that p points at |

Example of a hard-to-understand pointer statement

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

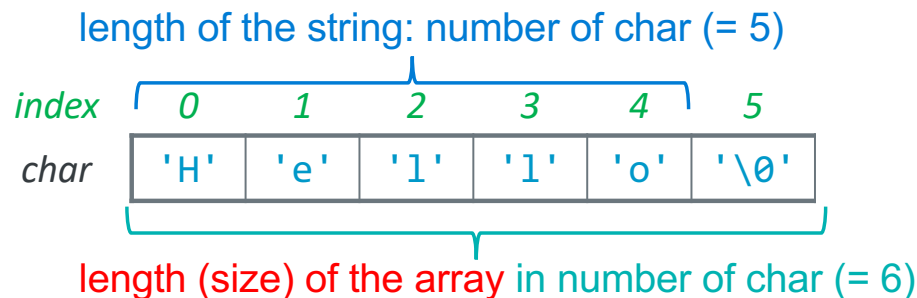
```
x = 1 + (*ptr++)++; // yuck!!
```

| 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 |

```
/* 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
```

C Strings - 1

- C does not have a **dedicated type** for strings
- Strings are an **array of characters** terminated by a **sentinel termination character**
- `'\0'` is the **Null termination character**; has the **value of zero** (do not confuse with `'0'`)
- An **array of chars** contains **a string only when** it is terminated by a `'\0'`
- **Length of a string** is the **number of characters** in it, not including the `'\0'`
- Strings in C are **not** objects
 - **No embedded information about them**, you **just have a name** and a memory **location**
 - You **cannot use + or +=** to concatenate strings in C
 - For example, you must **calculate string length** using code at runtime looking for the sentinel



C Strings - 2

- First '`\0`' encountered from the start of the string always indicates the end of a string
- The '`\0`' **does not have to be** in the **last element in the space allocated to the array**
 - But String length is always **less than the size of the array** it is contained in
- In the example below, the array `buf` contains two strings (but only `cat` is seen as the string)
 - One string starts at `&(buf[0])` is `"cat"` with a string length of 3
 - The other string starts at `&(b[4])` is `"o"` with a string length of 1
 - `"o"` has two bytes: `'o'` and `\0`

string length: number of char (= 3)

string length: number of char (= 1)

| | | | | | | |
|-------|------|------|------|------|------|------|
| index | 0 | 1 | 2 | 3 | 4 | 5 |
| buf | 'c' | 'a' | 't' | '\0' | 'o' | '\0' |
| | 0x63 | 0x61 | 0x74 | 0x00 | 0x6f | 0x00 |

length (size) of the array in number of char (= 6)

Defining Strings: Initialization

- When you combine the automatic length definition for arrays with double quote("") **initialization**
 - Compiler automatically adds the null terminator '\0' for you

```
char a[4] = {'c', 'a', 't', '\0'};  
char b[] = "cat";  
char c[] = {'c', 'a', 't', '\0', 'a', 'b'};  
char empty[] = "";  
// compiler calculates size, adds '\0'  
// array size 6, first string length 3  
// empty string - contains '\0'  
// string length = 0
```

Background: Different Ways to Pass Parameters

- **Call-by-reference (or pass by reference)**

- Parameter in the called function is an alias (references the same memory location) for the supplied argument
- Modifying the parameter modifies the calling argument

Call-by-value (or pass by value) (C)

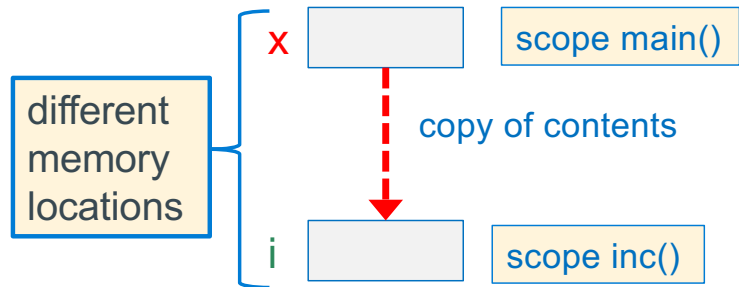
- What **Called** Function Does
 - Passed Parameters are used like local variables
 - Modifying the passed parameter in the function is allowed just like a local variable
 - So, writing to the parameter, only changes the copy
- The return value from a function in C is **by value**

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

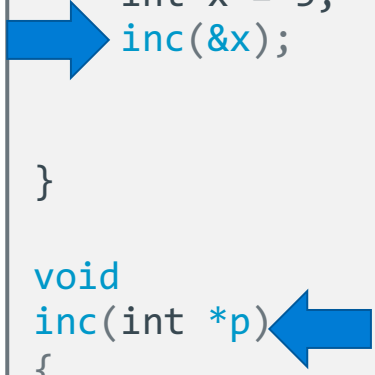


- when `inc(x)` is called, a copy of `x` is made to another memory location
 - `inc()` cannot change the variable `x` since `inc()` does not have the address of `x`, it is local to `main()` so, 5 is printed
- The `inc()` function is free to change its copy of the argument (just like any local variable)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

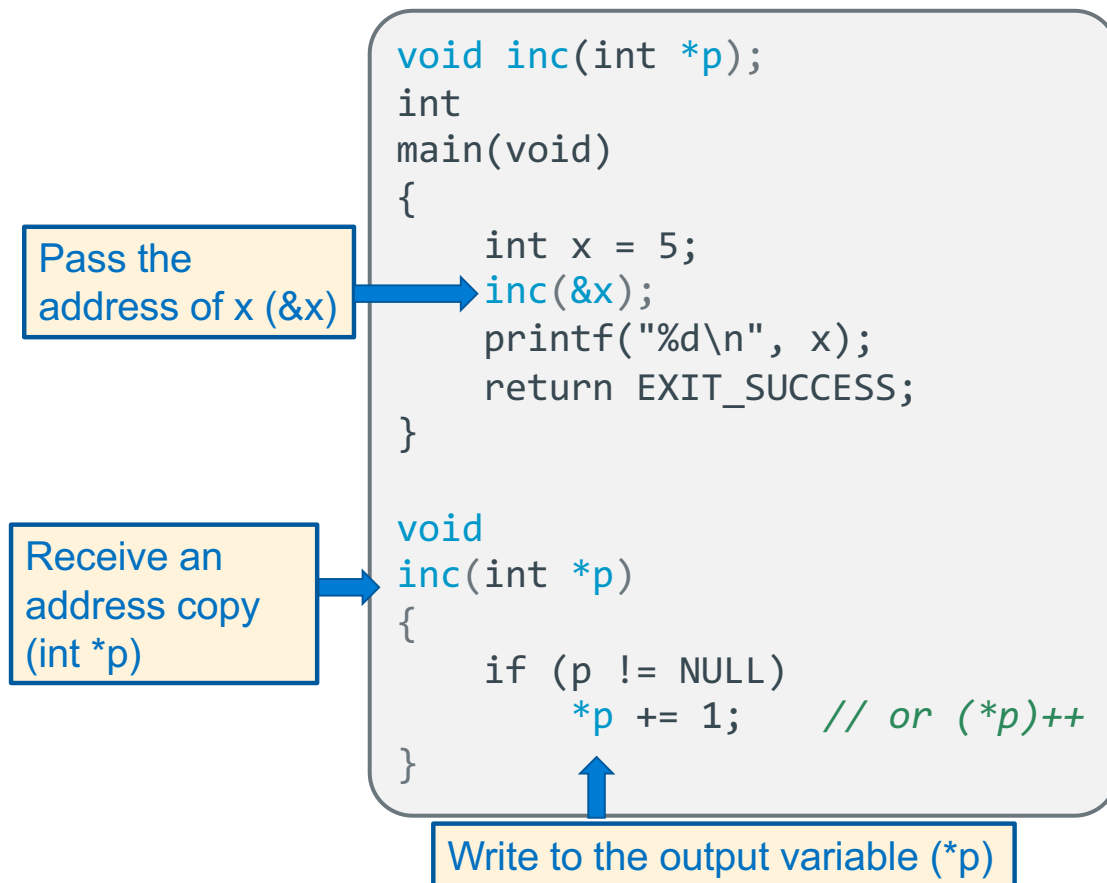
Output Parameters (Mimics Call by Reference)

- 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 (add *)
 - 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 *p);  
int  
main(void)  
{  
    int x = 5;  
    inc(&x);  
}  
  
void  
inc(int *p)  
{  
  
}  
}
```

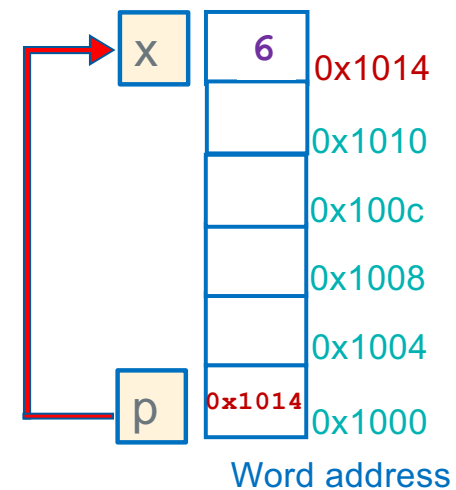


Example Using Output Parameters



At the Call to inc() in main()

1. Allocate space for p
2. Copy x's address into p



With a pointer to X,

inc() can change x in main()
this is called a side effect

p just like any other local variable

Array Parameters: Call-By-Value or Call-By-Reference?

- `Type []` array parameter is automatically “promoted” to a pointer of type `Type *`, and a copy of the *pointer* is *passed by value*

```
int main(void)
{
    int numbers[] = {9, 8, 1, 9, 5};

    passa(numbers);
    printf("numbers size:%lu\n", sizeof(numbers)); // 20
    return EXIT_SUCCESS;
}
```

```
void passa(int a[])
{
    printf("a size:%lu\n", sizeof(a)); // 4
    return;
}
```

IMPORTANT:

See the size difference 20 in `main()` in `passa()` is 4 bytes (size of a pointer)

- Call-by-value pointer (callee can change the pointer parameter to point to something else!)
- Acts like call-by-reference (called function can change the contents caller's array)

Arrays As Parameters: What is the size of the array?

- It's tricky to use arrays as parameters, as **they are passed as pointers to the start of the array**
 - In C, Arrays do not know their own size and at runtime there is no “bounds” checking on indexes

```
int sumAll(int a[]);  
  
int main(void)  
{  
    int numb[] = {9, 8, 1, 9, 5};  
    int sum = sumAll(numb);  
  
    return EXIT_SUCCESS;  
}  
  
int sumAll(int a[])  
{  
    int i, sum = 0;  
    int sz = (int) (sizeof(a)/sizeof(*a));  
    for (i = 0; i < sz; i++) // this does not work  
        sum += a[i];  
}
```

the name is the address, so this is passing a pointer to the start of the array

“inside” the body of sumAll(), the question is: how big is that array? all I have is a POINTER to the first element.....
sz is a 1 on 32 bit arm

Arrays As Parameters, Approach 1: Pass the size

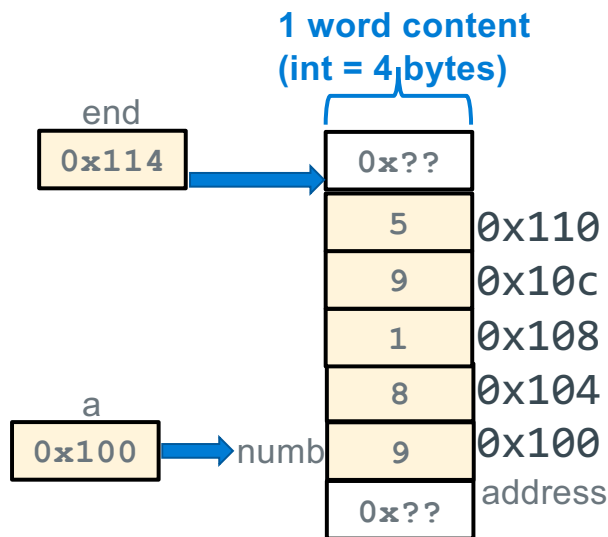
Two ways to pass array size

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

```
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;
}
```



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

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

same as:
sum = sum + *a;
a++;

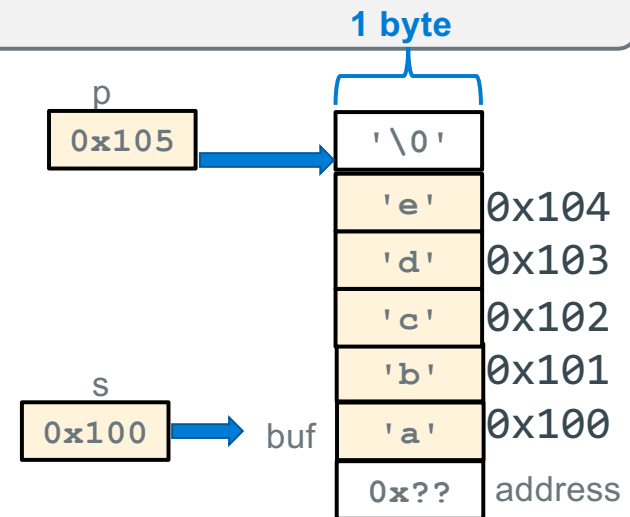
Arrays As Parameters, Approach 2: Use a sentinel element

- 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); // returns number of chars in string, not counting \0
int main(void)
{
    char buf[] = {'a', 'b', 'c', 'd', 'e', '\0'}; // or buf[] = "abcde";

    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);
}
```



Reference: Some String Routines in libc (#include <string.h>)

| Function | Description |
|---|---|
| <code>strlen(<i>str</i>)</code> | returns the # of chars in a C string (before null-terminating character). |
| <code>strcmp(<i>str1</i>, <i>str2</i>)</code> , <code>strncmp(<i>str1</i>, <i>str2</i>, <i>n</i>)</code> | compares two strings; returns 0 if identical, <0 if <i>str1</i> comes before <i>str2</i> in alphabet, >0 if <i>str1</i> comes after <i>str2</i> in alphabet. <i>strncmp</i> stops comparing after at most <i>n</i> characters. |
| <code>strchr(<i>str</i>, <i>ch</i>)</code> <code>strrchr(<i>str</i>, <i>ch</i>)</code> | character search: returns a pointer to the first occurrence of <i>ch</i> in <i>str</i> , or NULL if <i>ch</i> was not found in <i>str</i> . <code>strrchr</code> find the last occurrence. |
| <code>strstr(<i>haystack</i>, <i>needle</i>)</code> | string search: returns a pointer to the start of the first occurrence of <i>needle</i> in <i>haystack</i> , or NULL if <i>needle</i> was not found in <i>haystack</i> . |
| <code>strcpy(<i>dst</i>, <i>src</i>)</code> , <code>strncpy(<i>dst</i>, <i>src</i>, <i>n</i>)</code> | copies characters in <i>src</i> to <i>dst</i> , including null-terminating character. Assumes enough space in <i>dst</i> . Strings must not overlap. <i>strncpy</i> stops after at most <i>n</i> chars, and <u>does not</u> add null-terminating char. |
| <code>strcat(<i>dst</i>, <i>src</i>)</code> , <code>strncat(<i>dst</i>, <i>src</i>, <i>n</i>)</code> | concatenate <i>src</i> onto the end of <i>dst</i> . <i>strncat</i> stops concatenating after at most <i>n</i> characters. <u>Always</u> adds a null-terminating character. |
| <code>strspn(<i>str</i>, <i>accept</i>)</code> , <code>strcspn(<i>str</i>, <i>reject</i>)</code> | <i>strspn</i> returns the length of the initial part of <i>str</i> which contains <u>only</u> characters in <i>accept</i> . <i>strcspn</i> returns the length of the initial part of <i>str</i> which does <u>not</u> contain any characters in <i>reject</i> . |

Do not overuse strlen()

- C string library function `strlen()` calculates string length **at runtime**
- **Do not overuse `strlen()`, as it walks the array each time called**

```
int count_e(char *s) //  $O(n^2)$  !!!  
{  
    int count = 0;  
    if (s == NULL)  
        return 0;  
    for (int j = 0; j < strlen(s); j++) {  
        if (s[j] == 'e')  
            count++;  
    }  
    return count;  
}
```



```
int count_e(char *s) //  $O(n)$  !!!  
{  
    int count = 0;  
    if (s == NULL)  
        return 0;  
    while (*s) {  
        if (*s++ == 'e')  
            count++;  
    }  
    return count;  
}
```

The NULL Constant and Pointers

- **NULL is a constant** that **evaluates to zero (0)**
- You **assign a pointer variable to contain NULL** to **indicate that the pointer does not point at anything**
- A **pointer variable** with a **value of NULL** is called a “**NULL pointer**” (invalid address!)
- Memory location 0 (address is 0) is not a valid memory address in any C program
- Dereferencing NULL at runtime will cause a program fault (segmentation fault)!

```
p = NULL;  
i = *p;           /* segmentation fault! */  
*(int *)900000 = 25; /* cast 900000 to a pointer */  
                  /* if writeable address space, it works */  
                  /* that memory location just changed */
```

Using the NULL Pointer

- Many functions return NULL to indicate an error has occurred

```
/* these are all equivalent */  
int *p = NULL;  
int *p = (int *)0;    // cast 0 to a pointer type  
int *p = (void *)0;   // automatically gets converted to the correct type
```

- NULL is considered “false” when used in a Boolean context
 - Remember: false expressions** in C are defined to be zero or NULL
- The following two are equivalent (the second one is preferred for readability):

```
if (p) ...  
if (p != NULL) ...
```

Simple String IO - Reading

| Task | Example Function Calls |
|---------------|---|
| Read a string | <pre>#include <stdio.h> char *strptr; char myStr[BFSZ]; strptr = fgets(myStr, BFSZ, stdin);</pre> <div>must pass the size of the array so fgets() knows how much space there is</div> |

`char *fgets(char array[], int size, FILE *stream)`

- `char *` is a pointer (address) to an array of char
- reads in at most **one less than size** characters from **stream** and stores them into **array**
- Reading stops after an **EOF** or a newline '\n'
 - If a newline ('\n') is read, it is stored into the buffer
 - **A terminating null byte ('\0') is always stored after the last character in the buffer**

| | | | | | | | | | | | | | | | | |
|---|---|---|---|--|---|---|--|---|---|---|---|---|---|---|----|----|
| t | h | i | s | | i | s | | a | s | t | r | i | n | g | \n | \0 |
|---|---|---|---|--|---|---|--|---|---|---|---|---|---|---|----|----|

- Returns a **NULL at end of file** (or a read failure), otherwise a pointer to array (pointers later...)
- See `man 3 fgets`

Pointer returns from a function call

```
char *next(char *ptr)
{
    if (ptr == NULL)
        return NULL;

    while ((*ptr != '\0') && (*ptr != ','))
        ptr++;

    if (*ptr == ',')
        return ++ptr;
    return NULL;
}
```

```
#include <stdlib.h>
#include <stdio.h>
#define BUFSZ 512
char *next(char *);

int main()
{
    char buf[BUFSZ];
    char *ptr;

    while (fgets(buf, BUFSZ, stdin) != NULL) {
        printf("buf: %s\n", buf);

        if ((ptr = next(buf)) != NULL)
            printf("after: %s\n", ptr);
        else
            printf("no comma found\n");
    }
    return EXIT_SUCCESS;
}
```

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:
 - Address of a **passed parameter copy** as the caller may or will deallocate it after the call
 - 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
}
```


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

| | | | | | | |
|--------------|-----|-----|-----|-----|-----|------|
| <i>index</i> | 0 | 1 | 2 | 3 | 4 | 5 |
| <i>char</i> | 'H' | 'e' | 'l' | 'l' | 'o' | '\0' |

```
// strncpy adds a length limit on copy
char str1[6];
int cnt = (int)(sizeof(str1) / sizeof(str1[0]));

strncpy(str1, "hello", cnt); // \0 copied
strncpy(str1, "hello", cnt - 1); // \0 not copied
```

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

    while ((*s0++ = *s1++) && --len) //watch short circuit
        here
        ;
    return str;
}
```

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

String Literals, Mutable and Immutable arrays - 1

- `mess1` is a **mutable** array (type is `char []`) with enough space to hold the string + `'\0'`

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

`mess1[]` Hello World\0

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

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

`mess2` → Hello World\0 ← read only string literal

- `mess3` is a **pointer** to a mutable array

```
char *mess3 = (char []) {"Hello World"}; // mutable string  
*(mess3 + 1) = '\0'; // ok
```

using the cast `(char [])`
makes it mutable

`mess3` → Hello World\0 ← mutable string

2D Arrays

- Generic (uniform) 2D array format:

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

- allocates a single, contiguous 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

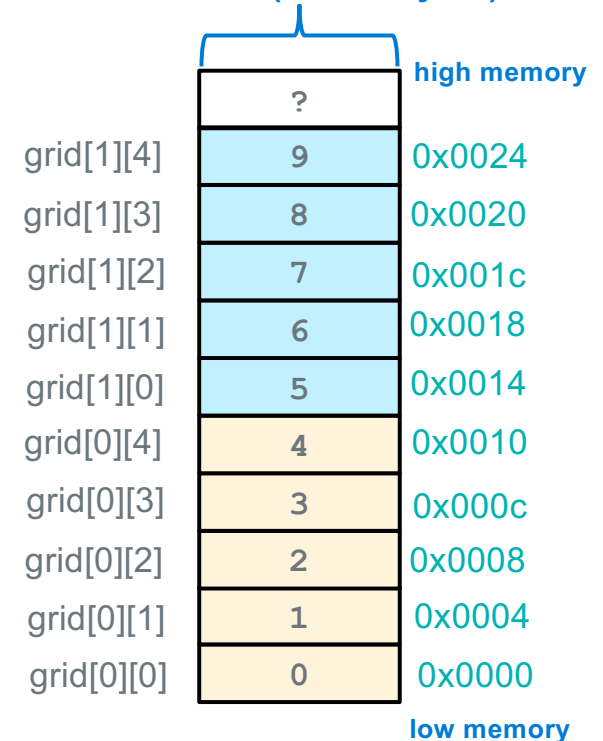
// Must specify row length, compiler counts rows

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

| | | | | |
|--------|--------|--------|--------|--------|
| [1][0] | [1][1] | [1][2] | [1][3] | [1][4] |
| [0][0] | [0][1] | [0][2] | [0][3] | [0][4] |

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

1 word (int = 4 bytes)



2D Array of Char (where elements may contain strings)

- 2D array of chars (where rows may include strings)
- Each row has the same fixed number of memory allocated
- All the rows are the same length regardless of the actual string length)
- The column size must be large enough for the longest string

high memory char aos2d[3][22] = {"my", "two dimensional", "char array"};

| | | | | | | | | | | | | | | | | | | | | | | |
|----------|---|---|------|---|---|---|---|---|---|---|------|---|---|---|---|--|---|---|---|---|---|------|
| aos2d[2] | c | h | a | r | | a | r | r | a | y | '\0' | | | | | | | | | | | |
| aos2d[1] | t | w | o | | d | i | m | e | n | s | i | o | n | a | l | | a | r | r | a | y | '\0' |
| aos2d[0] | m | y | '\0' | | | | | | | | | | | | | | | | | | | |

low memory

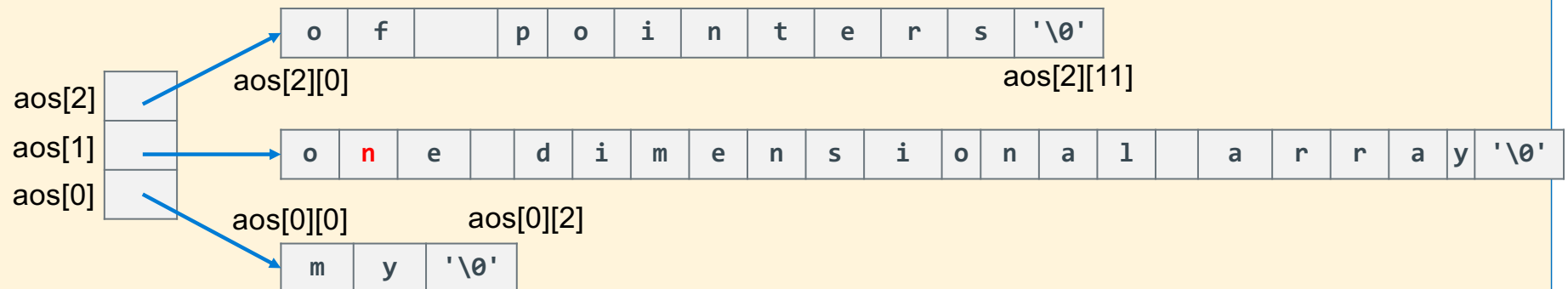
```
#define ROWS 3
char aos[ROWS][22] = { "my", "two dimensional", "char array"};
char (*ptc)[22] = aos; // ptr points at a row of 22 chars

for (int i = 0; i < ROWS; i++)
    printf("%s\n", *(ptc + i));
```

high memory

Pointer Array to Strings (This is NOT a 2D array)

- 2D char arrays are an inefficient way to store strings (wastes memory) unless all the strings are similar lengths, so 2D char arrays are *rarely used* with string elements
- **An array of pointers** is common for strings as "rows" can vary in length
- `char *aos[3];`



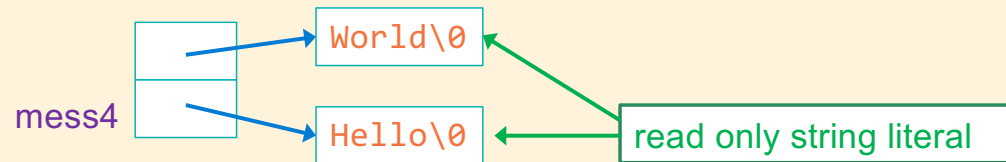
- `aos` is an array of pointers; each pointer points at a character array (also a string here)
- **Not a 2D array**, but any char can be accessed as if it was in a 2D array of chars
 - When I was learning, this was the most confusing syntax aspects of C

String Literals, Mutable and Immutable arrays - 2

- `mess4` is an array of pointers to immutable arrays

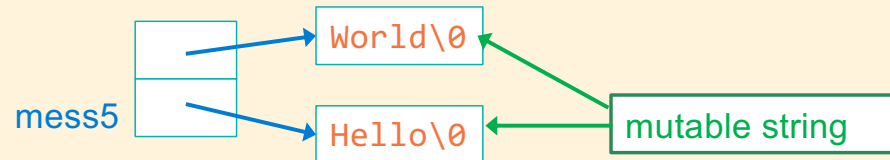
```
char *mess4[] = {"Hello", "World"}; // immutable string  
*(*mess4 + 1) = '\0'; // bus error
```

Bus error: writing
read only memory
Seg fault: writing
unallocated memory



- `mess5` is an array of pointers to mutable arrays

```
char *mess5[] = { (char []){"Hello"}, (char []){"World"}};  
*(*mess5 + 1) = '\0'; // OK!
```

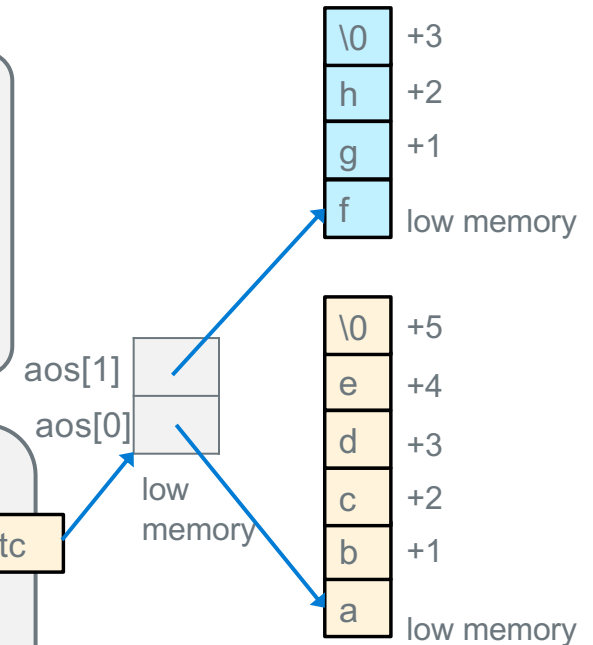


Pointer Array to Mutable Strings

- 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++;  
}
```

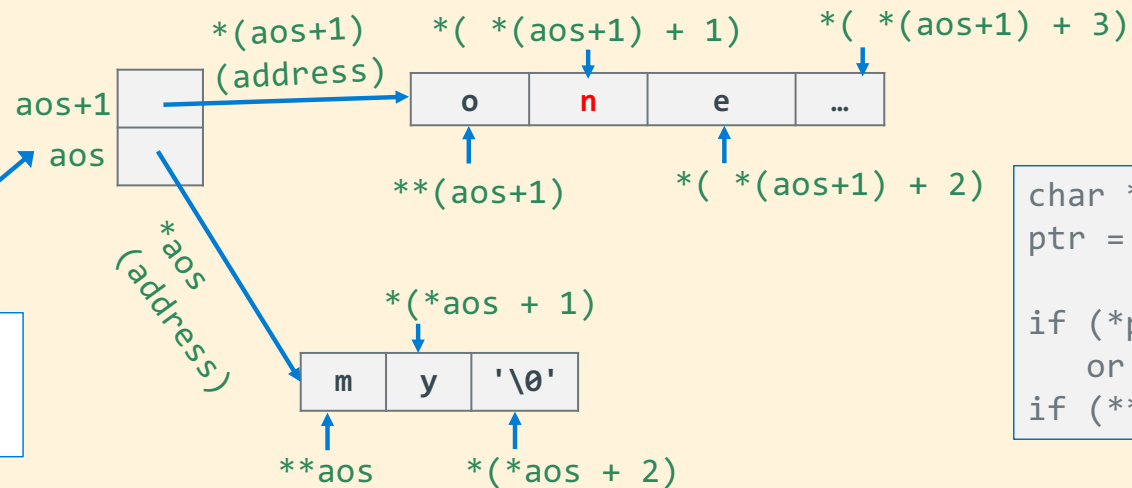
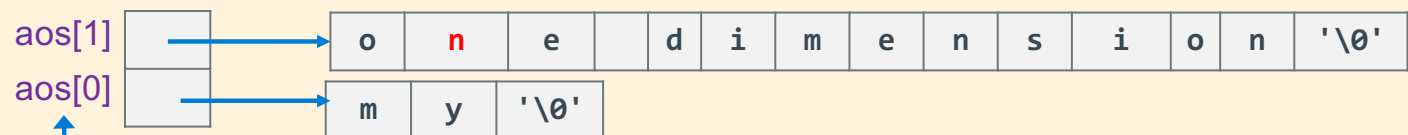


```
%./a.out  
g  
abcde  
abcde  
fgh  
fgh
```


Pointer Array to Strings

How to access: `aos[1][1]` is `*(*(aos + 1) + 1)` which contains 'n'
its address is `(*(aos + 1) + 1)`

aos+2 is not shown due to space limits on the slide

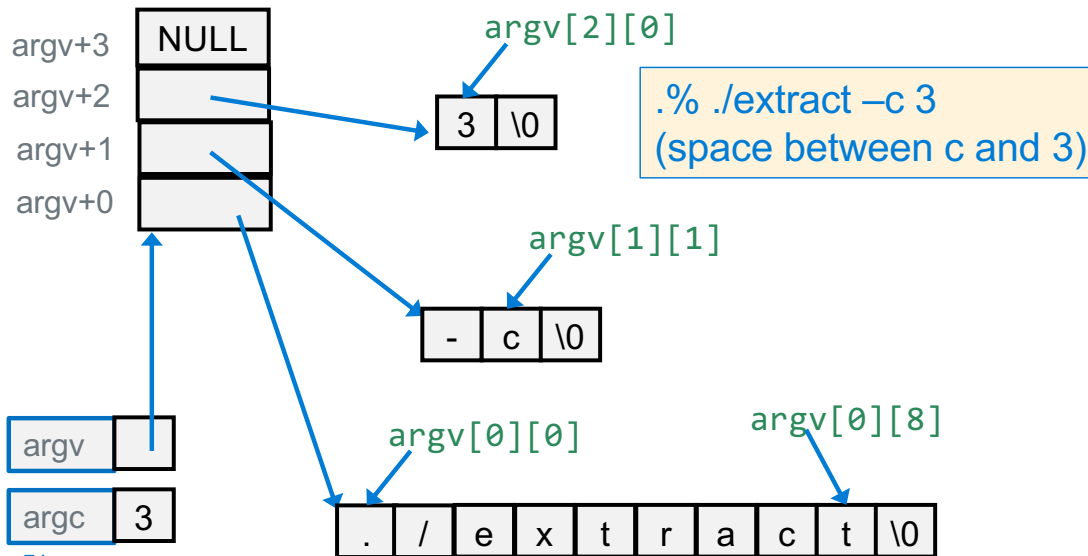


```
char *ptr;  
ptr = *aos;  
  
if (*ptr == ',')  
    or  
if (**aos) == ','
```

Notice that the first elements address is the array name

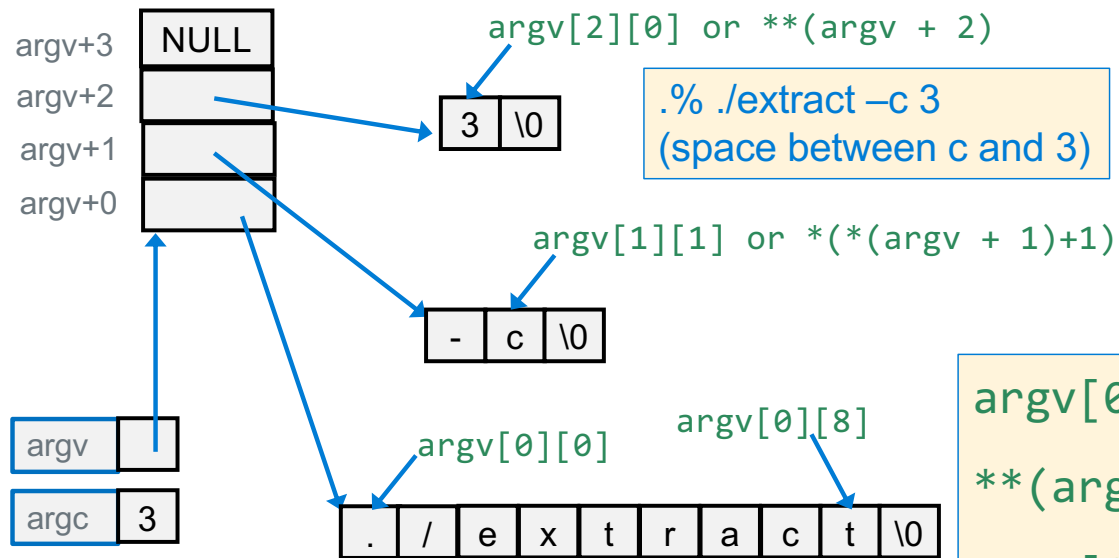
main() Command line arguments: argc, argv

- Arguments are passed to main() as a pointer to an array of pointers (****argv** or ***argv[]**)
Conceptually: % *argv[0] *argv[1] *argv[2]
- argc** is the number of VALID elements (they point at something)
- *argv** (**argv[0]**) is **usually** is the **name** of the executable file (% **./vim** file.c)
- *(argv + argc)** always contains a NULL (0) sentinel
- *argv[]** (or ****argv**) elements point at **mutable strings!**



```
printf("%s\n", *(argv+0));  
printf("%s\n", *(argv+1));  
printf("%s\n", *(argv+2));
```

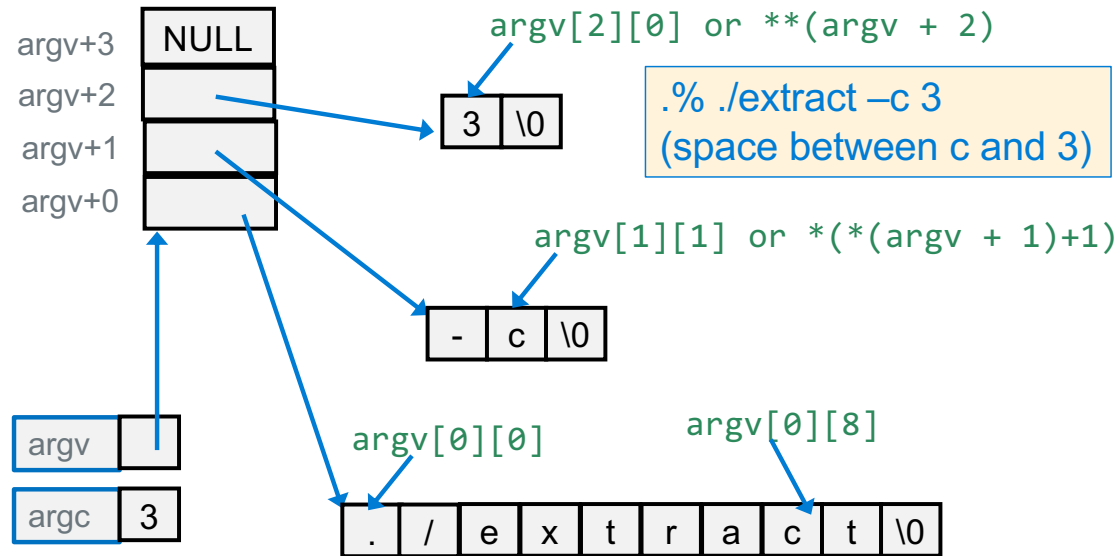
main() Command line arguments: argc, argv



`argv[0][0]` equiv to `** (argv+0)`
`** (argv+0)` equiv `**argv`
`argv[0][8]` equiv `* (*argv + 8)`

```
char *pt = *argv;  
*pt equiv to **argv  
*(pt+8) equiv to * (*argv + 8)
```

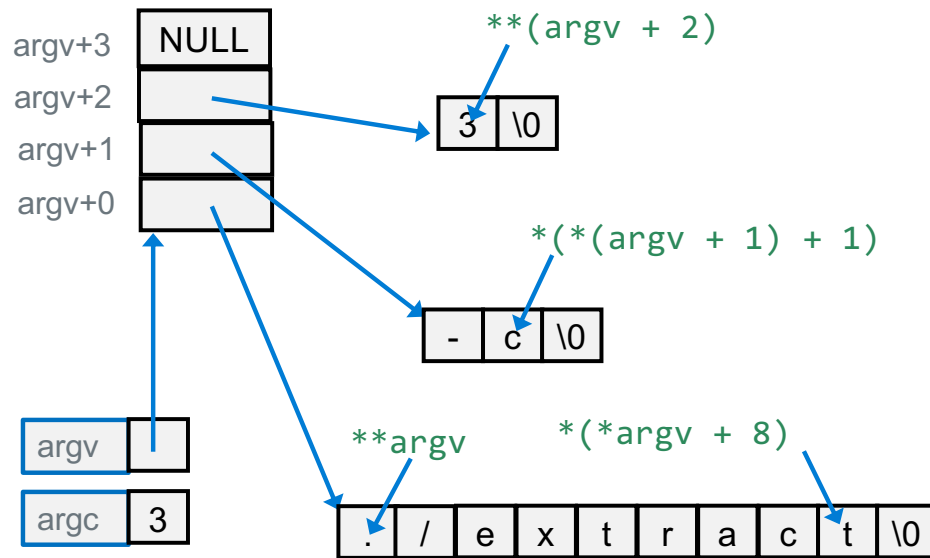
Printing argv char at a time



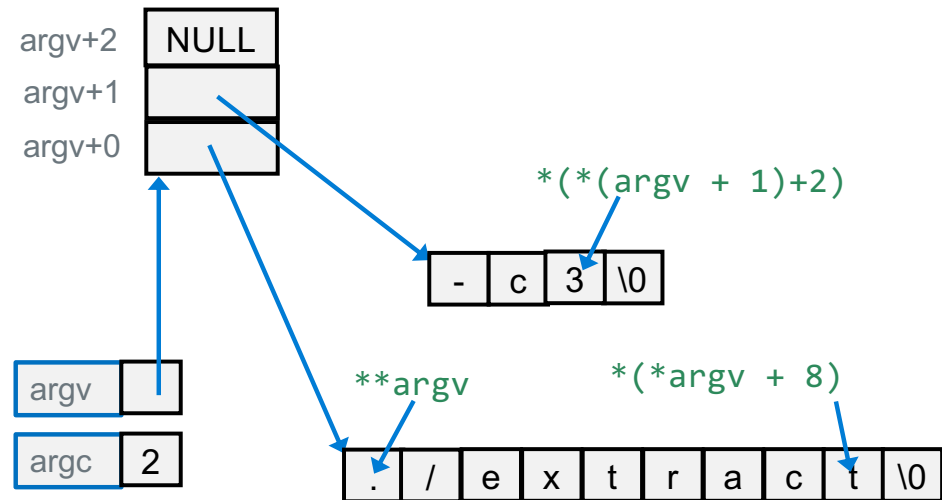
```
int main(int argc, char *argv[])
{
    for (int i = 0; argv[i] != NULL; i++) {
        for (int j = 0; argv[i][j] != '\0'; j++)
            putchar(argv[i][j]);
        putchar('\n');
    }
    return EXIT_SUCCESS;
}
```

```
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)

Pointers to Functions (Function Pointers)

- Similar in concept to an array name, a **function name ends up being the address of the first instruction in a function**

- A function pointer variable contains the address of a function

- Generic format: `returnType (*name) (type1, ..., typeN)`

- Looks like a function prototype with extra * in front of name
- Why are parentheses around (*name) needed?

`returnType *name (type1, ..., typeN) //wrong`

- Above says name is a function returning a pointer to returnType

- Using the function:

`(*name) (arg1, ..., argN)`

`name (arg1, ..., argN)`

Calls the pointed-to function with the given arguments and returns the return value

Pointers to Function Example

```
int main(void)
{
    int array[] = {4, 8, 15, 16, 23, 42};
    int cnt = sizeof(array)/sizeof(array[0]);

    print_array(array, cnt);

    array_update(add1, array, cnt);
    print_array(array, cnt);

    array_update(sqr, array, cnt);
    print_array(array, cnt);

    return EXIT_SUCCESS;
}
```

```
// array_update(f,a,len) (*f)() to each array element
void array_update(int (*f)(int), int *a, int len)
{
    for (int *p = a; p < a + len; ++p)
        *p = f(*p);
}
```

```
void print_array(int *a, int cnt)
{
    int *endp = a + cnt;
    for (int *p = a; p < endp; ++p)
        printf("%d ", *p);
    printf("\n");
}
```

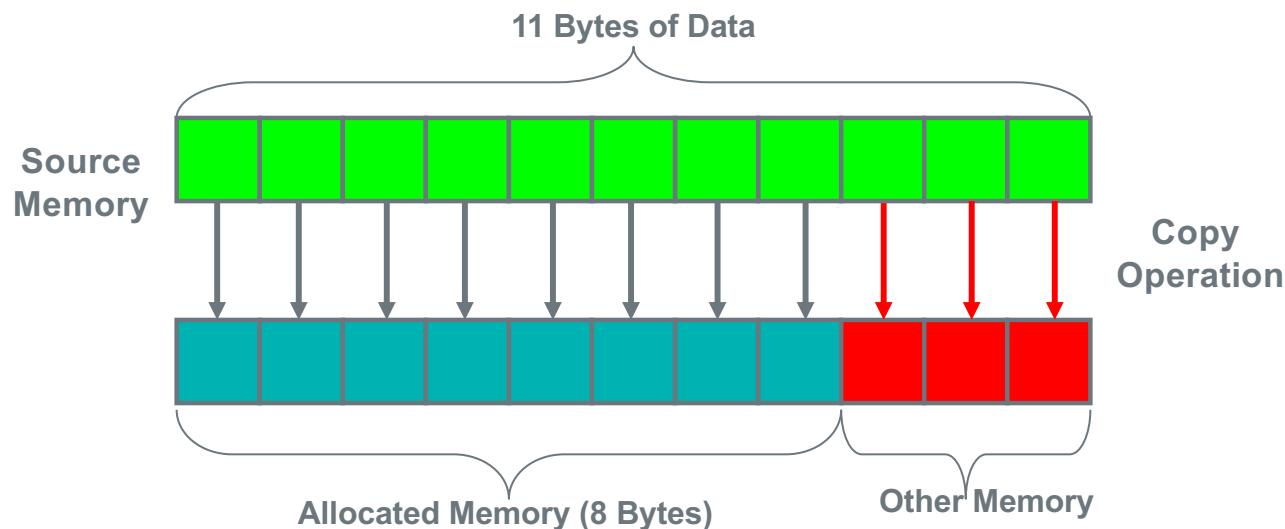
```
int add1(int i)
{
    return i + 1;
}

int sqr(int i)
{
    return i * i;
}
```

```
./a.out
4 8 15 16 23 42
5 9 16 17 24 43
25 81 256 289 576 1849
```

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**



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

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

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

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

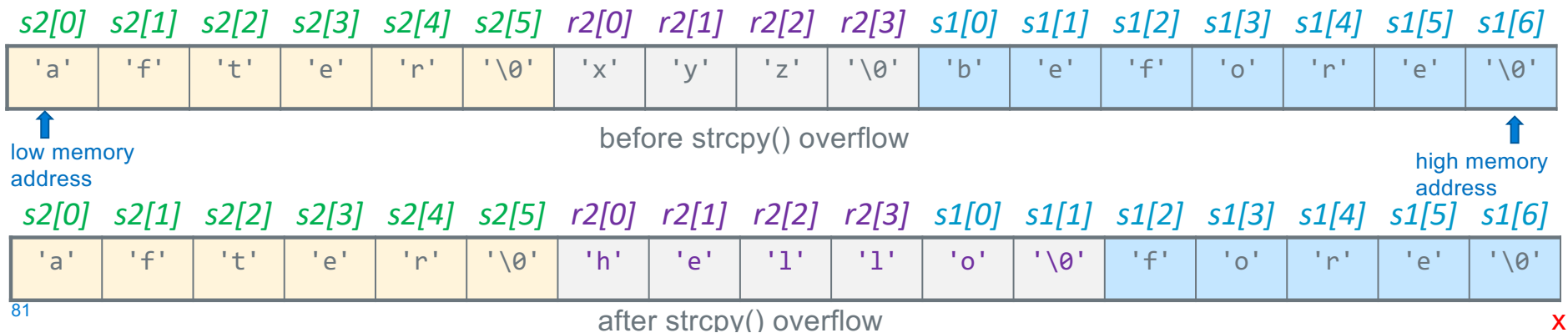
    printf("\ns2:%s\nr2: %s\nr2:%s\n",s2,r2,s1);
    return EXIT_SUCCESS;
}
```

these are mutable
arrays, not literals

compile on pi-cluster with
gcc test.c

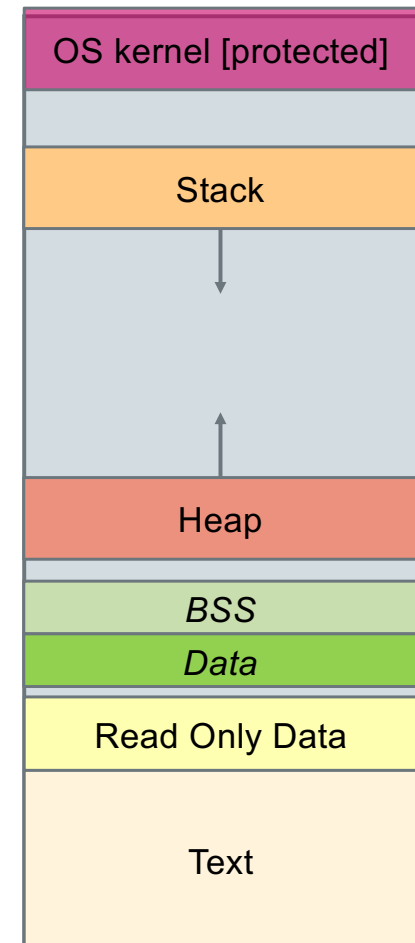
```
./a.out
s2: after
r2: xyz
s1: before

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



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



Heap Dynamic Memory Allocation Library Functions

| <code>#include <stdlib.h></code> | args | Clears memory |
|--|--|---------------|
| <code>void *malloc(...)</code> | <code>size_t size</code> | no |
| <code>void *calloc(...)</code> | <code>size_t nmem, size_t memsize</code> | yes |
| <code>void *realloc(...)</code> | <code>void *ptr, size_size</code> | no |
| <code>void free(...)</code> | <code>void *ptr</code> | 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**

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
}
```

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
}
```

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!, NULL check not shown
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
```

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 deallocates the memory */  
    return;  
}
```

- **Best practice:** free up memory **you allocated** when you no longer need it
 - 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**
- **Valgrind** is a tool for finding memory leaks (not pre-installed in all linux distributions though!)

Valgrind – Finding Buffer Overflows and Memory leaks

```
1 #define SZ 50
2 #include <stdlib.h>
3 int main(void)
4 {
5     char *buf;
6     if ((buf = malloc(SZ * sizeof(*buf))) == NULL)
7         return EXIT_FAILURE;
8     *(buf + SZ) = 'A';
9     return EXIT_SUCCESS;
10 }
```

```
% valgrind -q --leak-check=full --leak-resolution=med -s ./valgexample
==651== Invalid write of size 1
==651==    at 0x10444: main (valg.c:8)
==651== Address 0x49d305a is 0 bytes after a block of size 50 alloc'd
==651==    at 0x484A760: malloc (vg_replace_malloc.c:381)
==651==    by 0x1041B: main (valg.c:6)
==651==
==651== 50 bytes in 1 blocks are definitely lost in loss record 1 of 1
==651==    at 0x484A760: malloc (vg_replace_malloc.c:381)
==651==    by 0x1041B: main (valg.c:6)
==651==
==651== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Writing outside of allocated buffer space

Memory not freed

More Dangling Pointers: Reusing "freed" memory

- 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

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 = '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
```

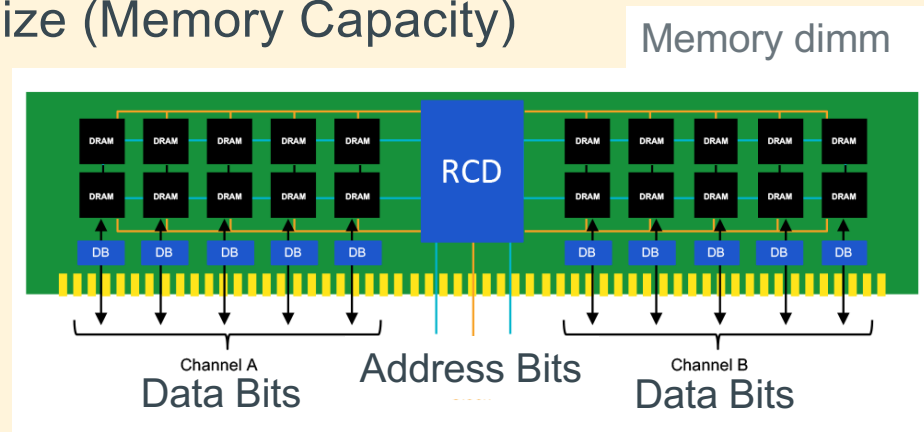
- Originally designed to allocate arrays but works for any memory allocation
 - calloc() multiplies the two parameters together for the total size
- calloc() is more expensive at runtime (uses both cpu and memory bandwidth) than malloc() because it must zero out memory it allocates at runtime
- Use calloc()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

Extra Slides

-

Memory Size

- Since memory addresses are implemented in hardware using binary
 - The **Size (number of byte sized cells)** of Memory is specified in **powers of 2**
- Memory size/capacity in **bytes** is specified by the “**Number of bits**” in an address
 - 32 bits of address = $2^{32} = 4,294,967,296$
 - Address Range is 0 to $2^{32} - 1$ (unsigned)
- Shorthand notation for address size (Memory Capacity)
 - KB = 2^{10} (K=1024) kilobyte
 - MB = 2^{20} megabyte
 - GB = 2^{30} gigabyte
 - TB = 2^{40} terabyte
 - PB = 2^{50} petabyte



Fixed size types in C (later addition to C)

- Sometimes programs need to be written for a particular range of integers or for a particular size of storage, regardless of what machine the program runs on
- In the file `<stdint.h>` the following fixed size types are defined for use in these situations:

| Signed Data types | Unsigned Data types | Exact Size |
|----------------------|-----------------------|-------------------|
| <code>int8_t</code> | <code>uint8_t</code> | 8 bits (1 byte) |
| <code>int16_t</code> | <code>uint16_t</code> | 16 bits (2 bytes) |
| <code>int32_t</code> | <code>uint32_t</code> | 32 bits (4 bytes) |
| <code>int64_t</code> | <code>uint64_t</code> | 64 bits (8 bytes) |

Defining Strings: Initialization Equivalents

- Following definitions create **equivalent** 4-character arrays
 - These are all strings as they all include a null ('\0') terminator

```
char a[4] = {'c', 'a', 't', '\0'};
char b[4] = {'c', 'a', 't', 0};
char c[4] = {'c', 'a', 't'};           // missing initial value defaults to 0
char d[4] = { 99, 97, 116, 0};         // 99 = 'c', 97 = 'a', 116 = 't'
char e[4] = "cat";
char f[4] = "cat\0";                   // literal has 5 chars; array f string
                                        // length is 3
```

Pointer Practice

`int *ptr;` Declares a variable, `ptr`, which is a pointer to (it contains the address of) an `int` in memory

`int x = 5;`
`int y = 2;` Declares two variables, `x` and `y`, that contain `ints`, and *initializes* them to 5 and 2, respectively


`ptr = &x;` Sets `ptr` to contain the address of `x` ("`ptr` points to `x`")


`y = 1 + *ptr;` Sets `y` to "1 plus the value stored at the address held by `ptr`. Because `ptr` points to `x`, this is equivalent to `y = 1 + x;`
"Dereference `ptr`"


`x = *(&y);` Sets `x = y`; The `*` and `&` cancel each other. get the address of `y` and then get the contents pointed by that address

`ptr`

`x` write
`y` write

`ptr`  `x`
`y` write

`ptr`  `x` read
`y` write

`ptr`  `x` write
`y` read

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)

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);
```

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

- 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

| | | | | | | |
|--------------|-----|-----|-----|-----|-----|------|
| <i>index</i> | 0 | 1 | 2 | 3 | 4 | 5 |
| <i>char</i> | 'H' | 'e' | 'l' | 'l' | 'o' | '\0' |

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

```
char *strcpy(char *s0, char *s1)  
{  
    char *str = s0;  
  
    if ((s0 == NULL) || (s1 == NULL))  
        return NULL;  
    while (*s0++ = *s1++)  
        ;  
    return str; // address of dest string  
}
```