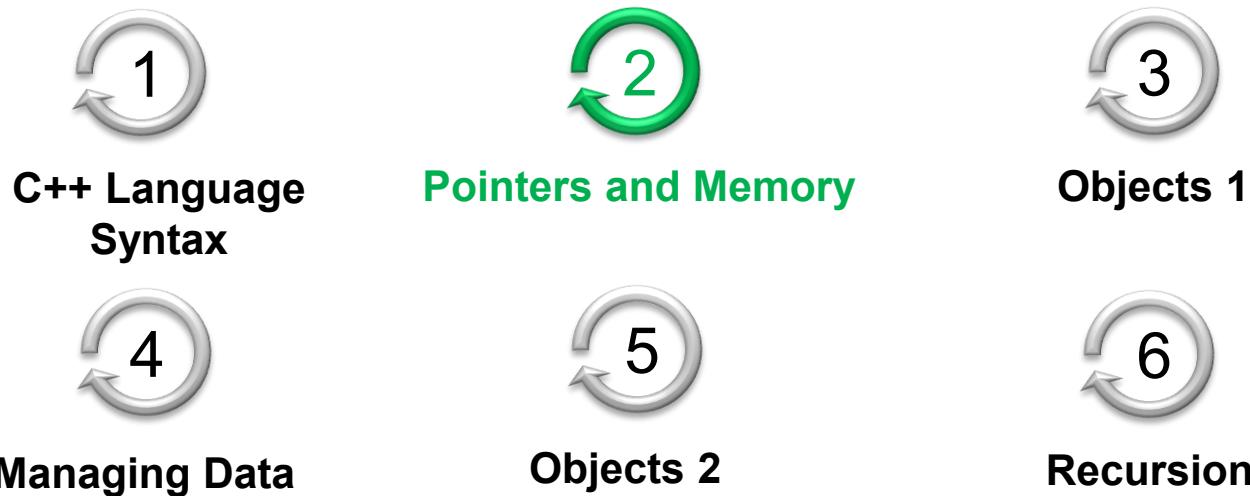
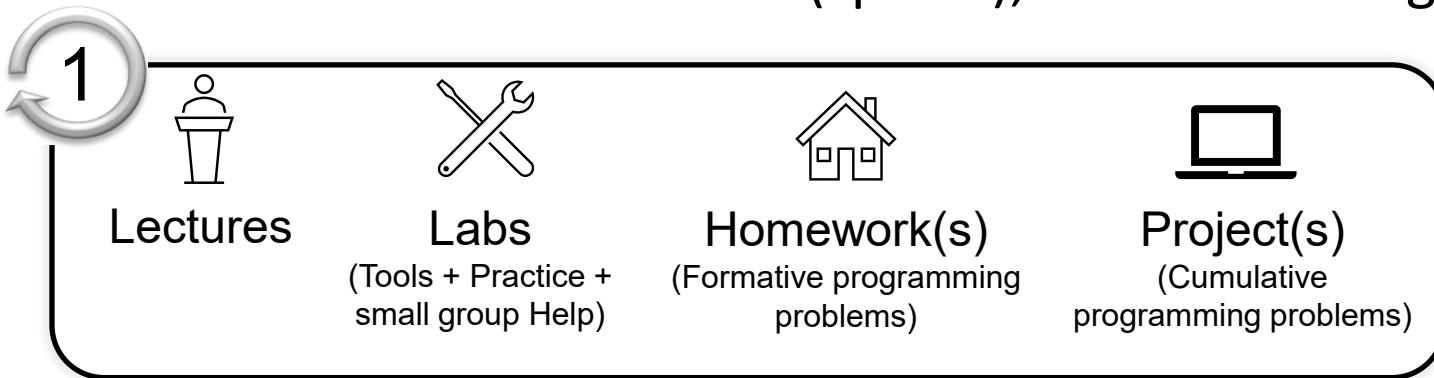


CS103 Unit 2a – Pointers and Pass by Reference

Unit 2 – Pointers and Memory

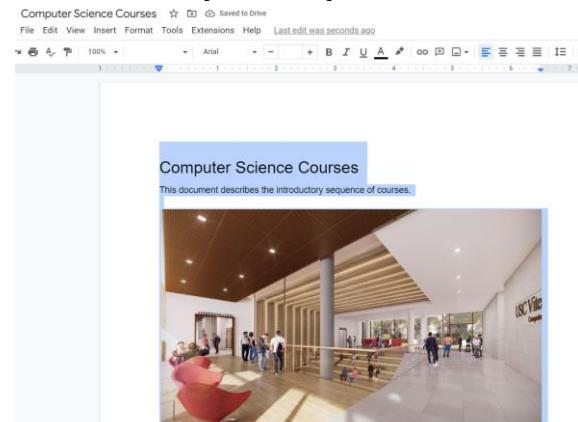
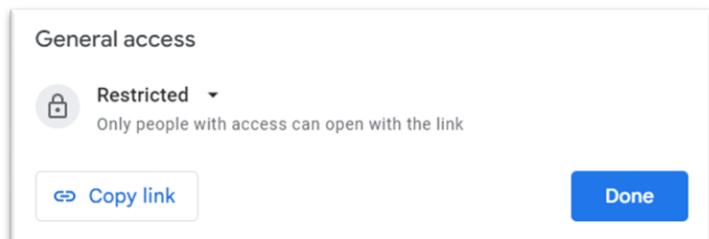
- The course is broken into 6 units (spirals), each consisting of:



INTRODUCTION TO POINTERS

Recall: Pass-by-Reference Pros/Cons

- **Scenario:** You write a paper and include a lot of **LARGE** images. You need to send it to your teammates. You can
 - As a Google doc and simply e-mail the URL or
 - Attach the document/file in the e-mail or
- What are the pros of each approach?
- Google Doc
 - Less info to send (send link, not all data)
 - Reference to original (i.e. if original changes, you'll see it)
- Email Attachment
 - Can treat the copy as a scratch copy and modify freely



Use Pointers when...

- We need **pass-by-reference** (as opposed to pass-by-value), either to:
 - **Change a variable (or variables) local to one function in some other function**
 - Analogy: a Google-doc link with "Can Edit" permission
 - **Avoid making needless copies** of data which wastes time
 - Analogy: A Google-doc link with "Can View" permission (think large arrays)
- We need to perform **dynamic memory allocation**
- We need to **access a specific location** in the computer (i.e. hardware devices) [Not covered in this class, but EE 109/CS 356]
 - Useful for embedded systems and device programming

All of these will be explained in the following slides.



Pointer Analogy

- Imagine a set of lockers or safe deposit boxes each with a number (just like memory locations have an address)
- There are some boxes with gold jewelry and others that do not contain gold but simply hold a piece of paper with another box number written on it (i.e. a pointer to another box)
- What is stored in one box might be:
 - [Box 7]:** Gold (i.e. data / something valuable like an int, double, etc.)
 - [Box 9]:** The number of another box which contains gold (i.e. box 9 holds a **pointer-to** some other data)
 - [Box 16]:** The number of another box which contains a number of a box containing gold (i.e. box 16 holds a **pointer-to a pointer-to** data)



Each box has a **number to identify it** (i.e. an **address**) and a **value inside of it**. So do variables in memory.

0 ₈	1	2 ₁₅	3	4	5 ₃
6 ₁₁	7	8 ₄	9 ₇	10 ₃	11
12	13 ₁	14	15	16 ₉	17 ₃

- The **value** of (i.e. what is in) one box might be the **address** of (pointer-to) another box.
- By changing the **number** in a box (i.e. the value of a pointer), we can have one location refer to many different **locations**, in succession.



Pointer Analogy

- But what if rather than gold or other obviously valuable objects, the "valuable objects" were simply slips of papers with numbers.
 - Would you be able to **distinguish** whether a box is **storing data or storing a pointer**?
 - And if it is storing a pointer, would you know whether it is pointing at just **1 data element or an array of data elements**?
- No! This is why we need:
 - Pointer **types** (e.g. `int*` or `char*`) to tell us that what's in this variable is a pointer as well as what kind of data we'll find when we follow (dereference) the pointer (e.g. `int` or `char`).
 - To remember context on our own (as the programmer)



0 ₈	1 ₇	2 ₁₅	3 ₉	4 ₁₅	5 ₃
6 ₁₁	7 ₁₂	8 ₄	9 ₇	10 ₃	11 ₄
12 ₆	13 ₁	14 ₈	15 ₁	16 ₉	17 ₃

Pointers

- Pointers are variables that store the **address** of some other variable in memory
- More abstractly, pointers are **references** to other "things" which can be:
 - **data** (i.e. ints, chars, doubles) or
 - **other pointers**
- The concept of a pointer is very common and used in many places in everyday life
 - Phone numbers or mailing addresses are references or "pointers" to your physical phone or location
 - Excel workbook has cell names we can use to reference the data (=A1 means get data in A1)
 - URLs (www.usc.edu is a pointer to a physical HTML file on some server) and can be used in any other page to "point to" USC's website

Address	Memory Data
73a8	ptr1 00000000
73ac	000073b8 ptr2 00000000
73b0	000073bc int a 108
73b4	double z -2.25
73b8	
73bc	
73c0	

Prerequisites: Data Sizes, Computer Memory

POINTER BASICS

Steps To Using Pointers

- **Variable**

- You can't use pointers without something to point to (create a variable of some type, **T var**)
 - Note: We use **T** as a placeholder for ANY type



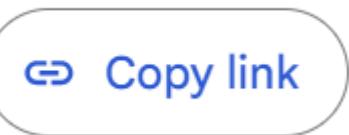
- **Pointer**

- Declare a pointer variable or argument (declare a variable of type **T* pvar**)



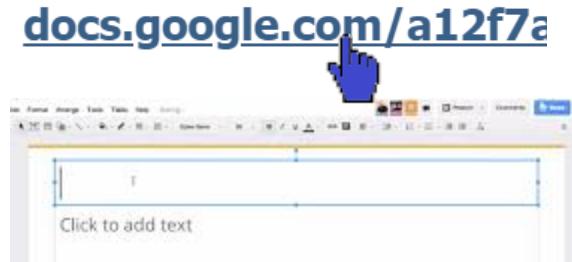
- **Link**

- Generate the pointer/link to the variable using **&** operator (**&var**)



- **Dereference (Use)**

- Follow the link to view or edit the variable using the ***** operator (***pvar**)



C++ Pointer Operators

- 2 operators used to manipulate pointers (i.e. addresses) in C/C++: **&** (address-of op) and ***** (dereference op)
 - **&<variable>** evaluates to the "address-of" **<variable>**
 - *Essentially, you get a pointer to a variable by writing &variable*
 - ***<pointer>** evaluates to the **data** pointed to by **<pointer>** (data at the address given by **<pointer>**)
 - **&** and ***** are **inverse operations**

- We say **&** returns the address/reference/link of some value while

- ***** dereferences the address and returns the value

- **&variable => address/pointer**

- ***pointer => variable value**

- ***(&variable) => variable**

& = get a link

 Unit0_FirstDayOverview.pdf

 Unit1a_CPP_Expr.pdf

 Unit1b_CPP_ProgramFlow_v2.pdf

Preview

Open with

Share

Get link

*** = follow the link**

docs.google.com/a12f7a

Generating a Pointer

- When a variable is declared, memory is allocated for it. Its **starting location** in memory is its address.

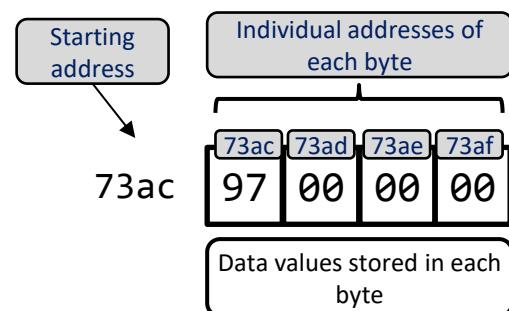
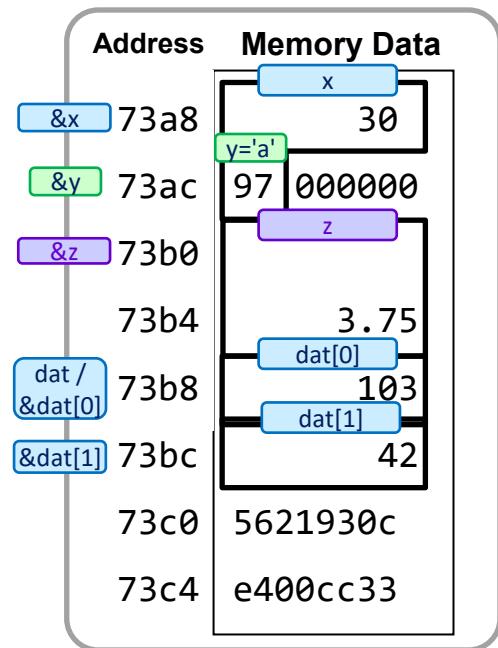
- `int x = 30;`
 - `char y = 'a';`
 - `double z = 3.75;`
 - `int dat[2] = {103,42};`

variable

- To generate a pointer, use the `&` operator to get the address of a variable in C/C++
(Tip: Read '`&x`' as 'address of x')

- `&x => _____`
- `&y => _____`
- `&z => _____`
- `&dat[1] = _____`
- `dat => _____`

link



- Great, but what should we do with these pointers and where should we store them?

Pointer Variables and their Declaration

- Data variable declarations:

- `int x = 30;`
- `char y = 'a';`
- `double z = 3.75;`
- `int dat[2] = {103,42};`

variable

- We can now declare **pointer variables** that don't store data but the **addresses of data**

- To declare a pointer, include a * after the type [e.g. `int*`], which is read "pointer to (an) int(s)". That variable can then store pointers to (addresses of) the given type (e.g. `int`)

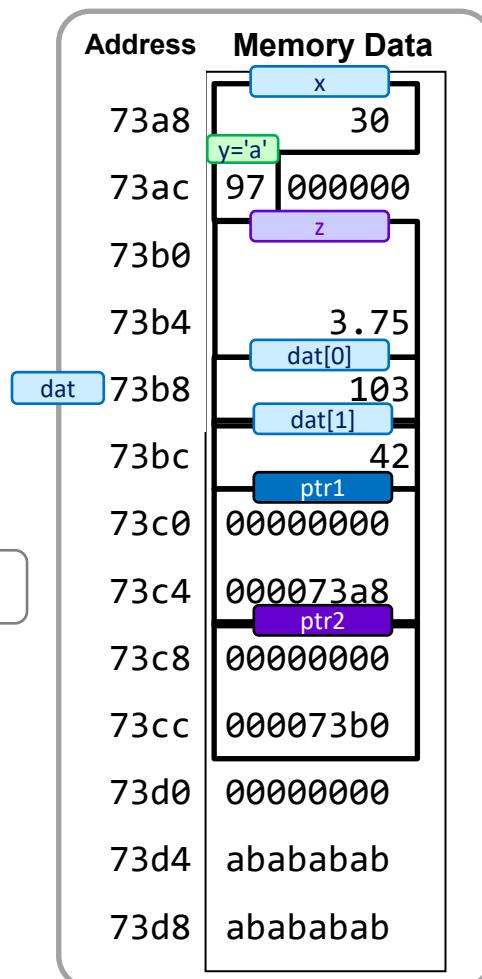
- `int *ptr1 = &x; // ptr1 = 0x73a8`
- `double* ptr2 = &z; // ptr2 = 0x73b0`
- `int* ptr3 = &y; // Error! Type mismatch.`

pointer

link

Notes:

1. Pointers should ONLY store the addresses of variables of its declared type (`int*` pointers should only point at ints, not chars)
2. Best to immediately initialize a pointer with the address of some variable, rather than leave it uninitialized.
3. Where the * is in the declaration (i.e. next to the type or variable name) does not matter [e.g. `int* ptr1 ..or.. int *ptr1`].



Dereferencing Pointer Variables

- Data variable declarations:

- `int x = 30;`
- `char y = 'a';`
- `double z = 3.75;`
- `int dat[2] = {103,42};`

variable

- We can declare pointer variables that store addresses of other variables

- `int *ptr1 = &x; // ptr1 = 0x73a8`
- `double* ptr2 = &z; // ptr2 = 0x73b0`

pointer

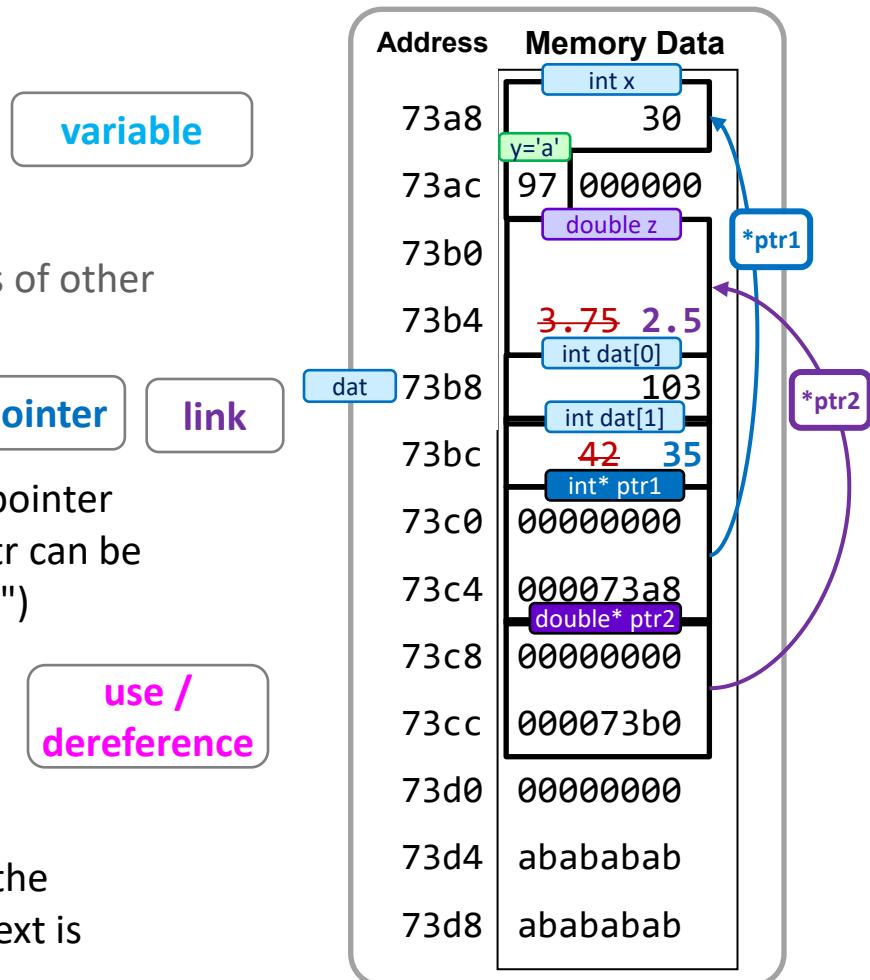
link

- We can access the data whose address is stored in a pointer variable by **dereferencing it** using the `*` operator. `*ptr` can be read as, "get/set **the data at the address stored in ptr**")

- `dat[1] = *ptr1 + 5; // dat[1] = 35`
- `*ptr2 = *ptr2 - 1.25; // z = 2.5`

use /
dereference

- It may be confusing but notice the `*` appears both in the declaration and in the dereference expression. Context is important to distinguish. More on the next slide...



Cutting through the Syntax

- * after a **type** = declare/allocate a pointer variable
- * in an **expression/assignment** = dereference

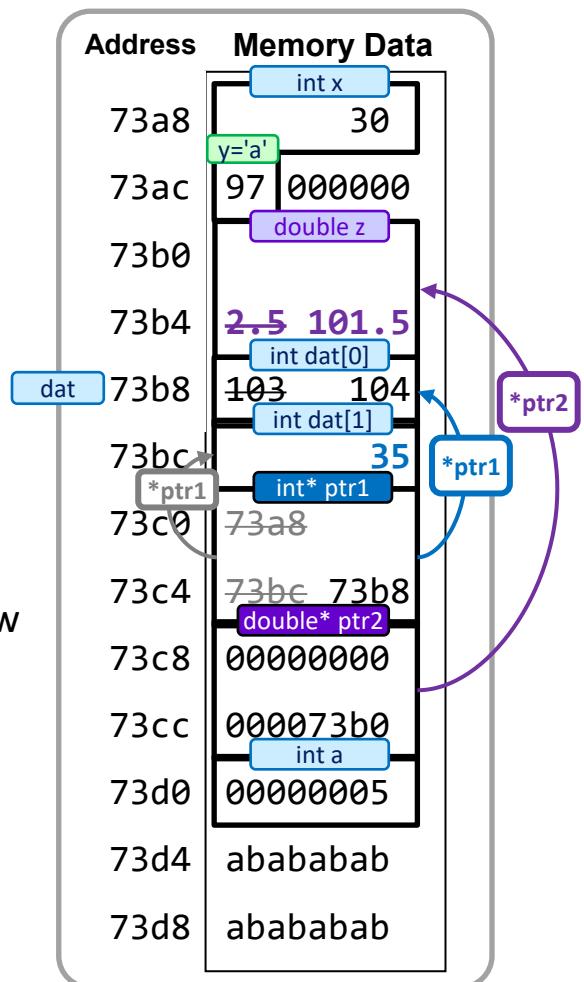
	Declaring a pointer	De-referencing a pointer
<code>char *p</code>	✓	
<code>x = *p + 1</code>		✓
<code>int* ptr</code>	✓	
<code>*ptr = 5</code>		✓
<code>(*ptr)++</code>		✓
<code>char* p1[10];</code>	✓	

Helpful tip to understand syntax: We declare a pointer as:

- `int *ptr` because when we dereference it as `*ptr`, we get an `int`
- `char *p` is a declaration of a pointer and thus, `*p` yields a `char`

Assigning to Pointer Variables

- Data variable declarations:
 - `int x = 30;`
 - `char y = 'a';`
 - `double z = 3.75;`
 - `int dat[2] = {103,42};`
- Declaring pointer variables and setting them with addresses (using &):
 - `int *ptr1 = &x; // ptr1 = 0x73a8`
 - `double* ptr2 = &z; // ptr2 = 0x73b0`
- Dereferencing pointer variables (using *) to get data pointed to:
 - `dat[1] = *ptr1 + 5; // dat[1] = 35`
 - `*ptr2 = *ptr2 - 1.25; // z = 2.5`
- We can change what variable the pointer references by assigning a new address to it and dereference the pointer as many times as we like
 - `ptr1 = &dat[1];`
`int a = *ptr1 % 10; // a = 5 after exec.`
 - `ptr1 = dat; // why is & not needed?`
 - `*ptr1 += 1; // dat[0] = 104`
 - `*ptr2 = *ptr1 - *ptr2;`



Skill: Drawing Data Diagrams

- Though painful, it is helpful to draw out relevant data diagrams, especially when pointers are involved
 - Draw a table with a column for each variable (or just a box for each variable)
 - Label the column header with the variable name and a **made-up address**. Use whatever number is easiest for your address: @100, @500, etc.)
 - Annotate the variable values as you evaluate each line of code
- DS Malik Textbook: Chapter 13, Question 6

```
int x, y;
```

variable

```
int *p = &x;
```

pointer

link

```
int* q = &y;
```

```
x = 35; y = 46;
```

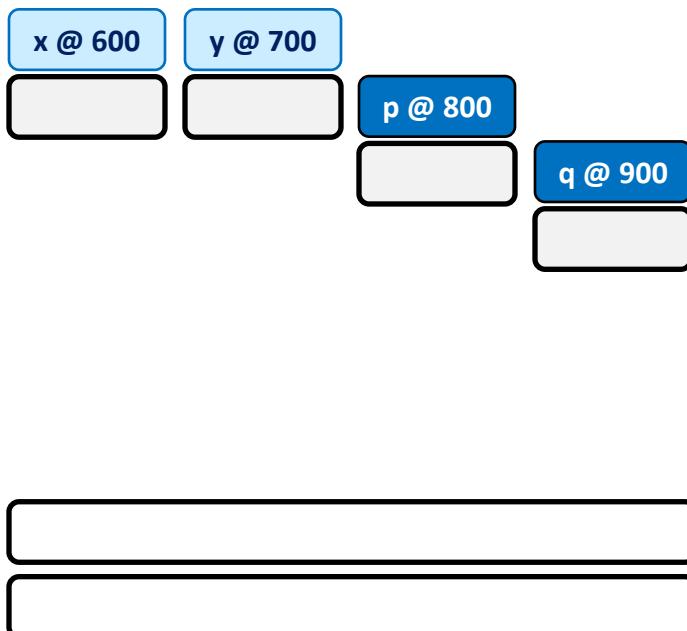
```
p = q;
```

```
*p = 78;
```

use / dereference

```
cout << x << " " << y << endl;
```

```
cout << *p << " " << *q << endl;
```



Skill: Drawing Data Diagrams (Sol)

- Though painful, it is helpful to draw out relevant data diagrams, especially when pointers are involved
 - Draw a table with a column for each variable (or just a box for each variable)
 - Label the column header with the variable name and a **made-up address**. Use whatever number is easiest for your address: @100, @500, etc.)
 - Annotate the variable values as you evaluate each line of code
- DS Malik Textbook: Chapter 13, Question 6

```
int x, y;
```

variable

```
int* p = &x;
```

pointer

```
int* q = &y;
```

link

```
x = 35; y = 46;
```

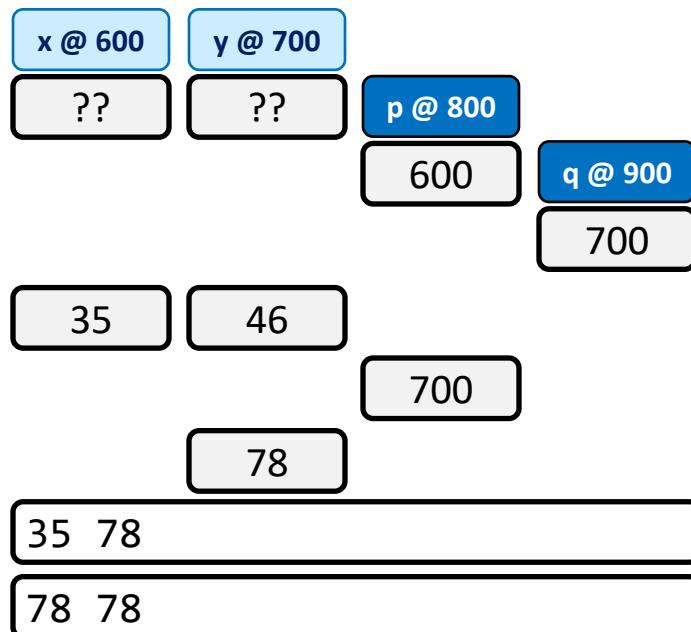
```
p = q;
```

```
*p = 78;
```

use / dereference

```
cout << x << " " << y << endl;
```

```
cout << *p << " " << *q << endl;
```



Pointer Summary

- To summarize:

- We can **declare pointer variables** to store addresses (not data) using the syntax **T*** where T is some type (e.g. **int *p**)
- We can **get the address** of some variable using the **&** operator (e.g. **&x, &y**)
 - Most often, this would then be assigned to a pointer variable (e.g. **p = &x**)
- We can **dereference a pointer** (i.e. follow a pointer) to get the data from the address it stores by using the ***** operator (e.g. **cout << *p << endl**)
- We can **change the address** the pointer stores to have it reference some other variable (e.g. **p = &z**)

- But why do we need them?

- Can't we just access x, y, and z directly?

```
int main(int argc, char *argv[])
{
    int x = 103;
    char y = 'a';
    int z = 42;

    int* p = &x;
    char *q = &y;

    *p = 42;
    cout << *p << endl;

    p = &z;

    cout << *p << endl;

    return 0;
}
```

Prerequisites: Pointer Basics

PASS BY REFERENCE

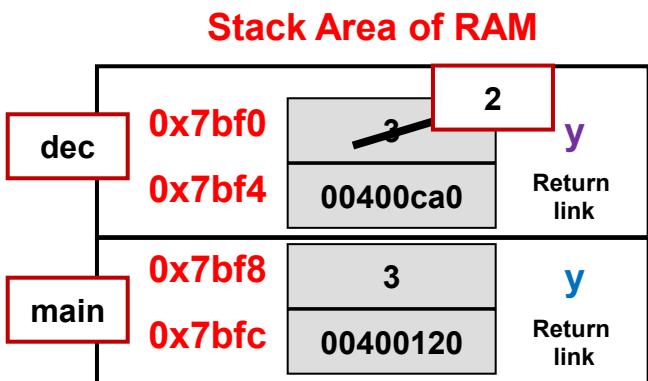
Recall: Pass-by-Value

- Each function has its own memory on the the **system stack** where all data related to the function is stored including:
 - **Local variables**
 - **Arguments to the function**
 - Return link (where to return) to the calling code
- When parameters are passed, a copy is made of the argument from the caller's area of the stack to a new location in the callee's area of the stack (aka **pass-by-value**)
 - This prevents one function from modifying the variables of another
- But what if we want a function to modify the data from another?
- We can use pointers!!! (aka **pass-by-reference**)

```
// Prototype
void dec(int);

int main()
{
    int y = 3;
    dec(y);
    cout << y << endl; // prints __
    return 0;
}

void dec(int y)
{
    y--;
}
```



Pass-by-(Pointer) Reference

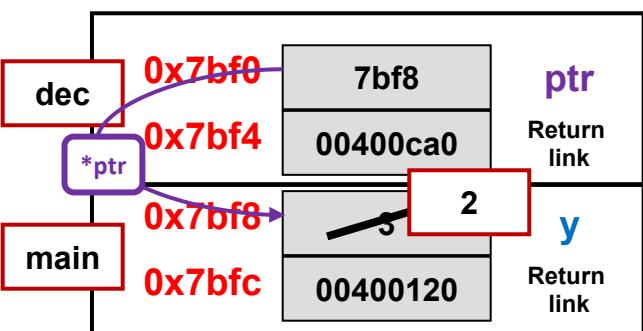
- We can now pass a pointer to a local variable from the caller function as an argument to the callee function.
- The pointer argument lives on the stack of the callee function but can be used (by dereferencing it) to access the local variable from the caller and modify its data.
- When the callee finishes and returns, the pointer argument dies, but the caller will now see the updated value of its local variable.
- Can you follow the syntax of the code to the right?

```
// Prototype
void dec(int);

int main() // caller
{
    int y = 3;
    dec(&y);
    cout << y << endl; // prints 2
    return 0;
}

void dec(int* ptr) // callee
{
    *ptr = *ptr - 1; // or (*ptr)--;
}
```

Stack Area of RAM

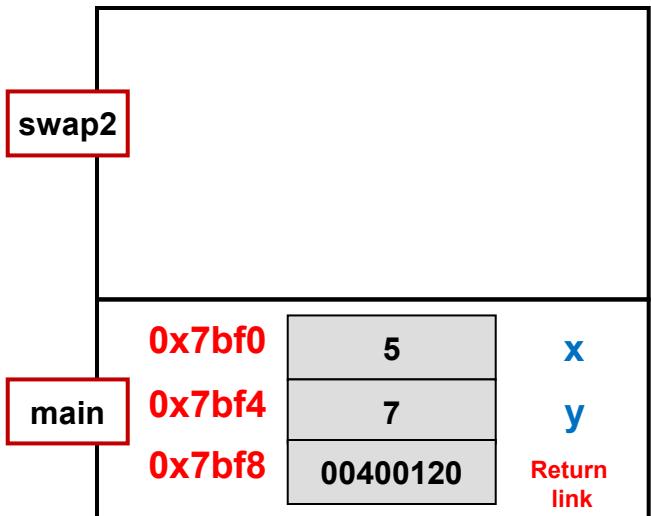


Swap Two Variables – (PB-Value Blank)

- Classic example of issues with local variables:
 - Write a function to swap two variables
- Pass-by-value doesn't work
 - Copy is made of x,y from main and passed to x,y of swapit
 - Swap is performed on the copies

Stack Area of RAM

Can you make a memory diagram of what is on the stack for swap2()?



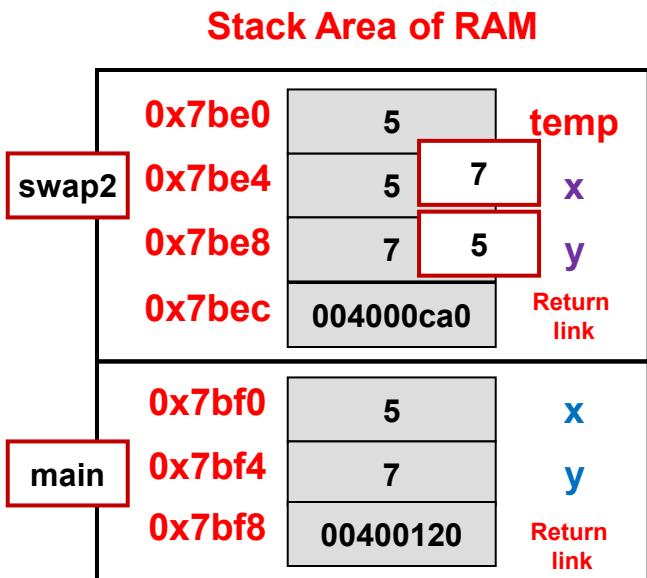
```
#include <iostream>
using namespace std;
void swap2(int x, int y);

int main()
{
    int x=5,y=7;
    swap2(x,y);
    cout << " x=" << x;
    cout << " y=" << y << endl;
}

void swap2(int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

Swap Two Variables – (PB-Value)

- Classic example of issues with local variables:
 - Write a function to swap two variables
- Pass-by-value doesn't work
 - Copy is made of x,y from main and passed to x,y of swapit
 - Swap is performed on the copies



```
#include <iostream>
using namespace std;
void swap2(int x, int y);

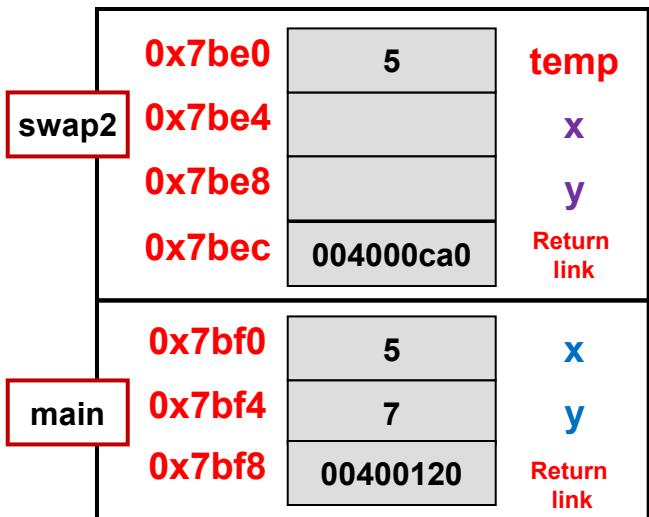
int main()
{
    int x=5,y=7;
    swap2(x,y);
    cout << " x=" << x;
    cout << " y=" << y << endl;
}

void swap2(int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

Swap Two Variables – (PB-Ref Blank)

- Classic example of issues with local variables:
 - Write a function to swap two variables
- Pass-by-reference (pointers) does work
 - Addresses of the actual x,y variables in main are passed
 - Use those address to change those physical memory locations

Can you fill in the values for x and y in swap2()?



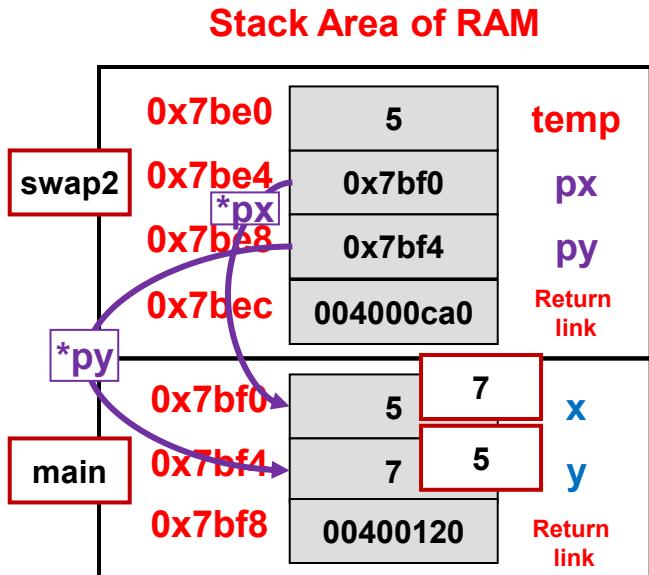
```
#include <iostream>
using namespace std;
void swap2(int* x, int* y);

int main()
{
    int x=5,y=7;
    swap2(&x, &y);
    cout << " x=" << x;
    cout << " y=" << y << endl;
}

void swap2(int* x, int* y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

Swap Two Variables – (PB-Ref)

- Classic example of issues with local variables:
 - Write a function to swap two variables
- Pass-by-reference (pointers) does work
 - Addresses of the actual x,y variables in main are passed
 - Use those address to change those physical memory locations



```
#include <iostream>
using namespace std;
void swap2(int* x, int* y);

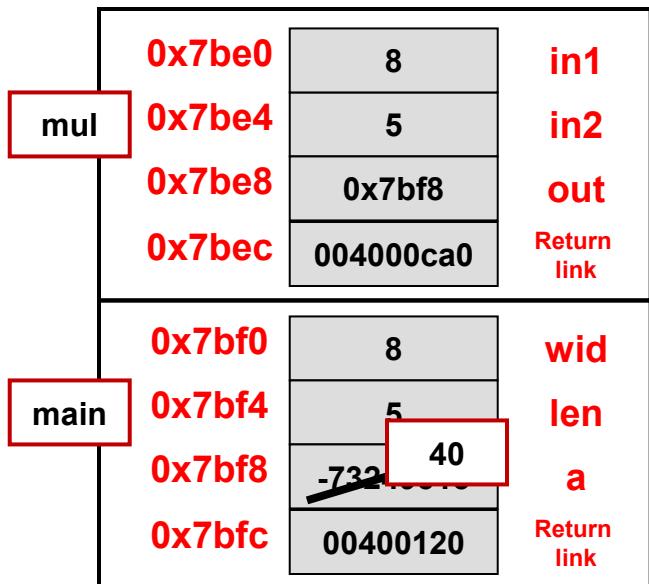
int main()
{
    int x=5,y=7;
    swap2(&x, &y);
    cout << " x=" << x;
    cout << " y=" << y << endl;
}

void swap2(int* px, int* py)
{
    int temp = *px;
    *px = *py;
    *py = temp;
}
```

Correct Usage of Pointers

- Commonly functions will take some inputs and produce some outputs
 - We'll use a simple 'multiply' function for now even though we can easily compute this without a function
 - We could use the return value but let's practice with pointers and say mul() must return void
- Can use a pointer to have a function modify the variable of another

Stack Area of RAM



```
// Computes the product of in1 & in2
int mul1(int in1, int in2);
void mul2(int in1, int in2, int* out);

int main()
{
    int wid = 8, len = 5, a;
    mul2(wid, len, _____);
    cout << "Ans. is " << a << endl;
    return 0;
}

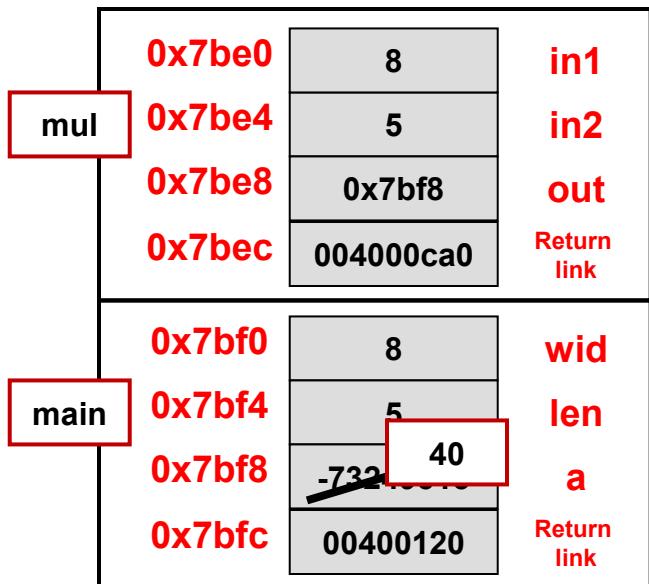
int mul1(int in1, int in2)
{
    return in1 * in2;
}

void mul2(int in1, int in2, _____)
{
    _____ = in1 * in2;
}
```

Correct Usage of Pointers

- Commonly functions will take some inputs and produce some outputs
 - We'll use a simple 'multiply' function for now even though we can easily compute this without a function
 - We could use the return value but let's practice with pointers and say mul() must return void
- Can use a pointer to have a function modify the variable of another

Stack Area of RAM



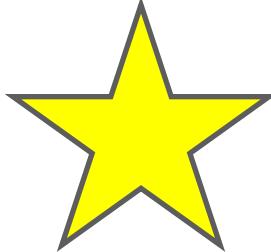
```
// Computes the product of in1 & in2
int mul1(int in1, int in2);
void mul2(int in1, int in2, int* out);

int main()
{
    int wid = 8, len = 5, a;
    mul2(wid, len, &a);
    cout << "Ans. is " << a << endl;
    return 0;
}

int mul1(int in1, int in2)
{
    return in1 * in2;
}

void mul2(int in1, int in2, int* out)
{
    *out = in1 * in2;
}
```

Pass-by-Reference Template



- To modify a type **T** variable named **var**:
 - Set the function to take a **T* varptr**
 - Pass **&var** in the caller function to create and send a pointer to the function.
 - In the calling function, dereference the pointer and assign:
***varptr = value**

```
// here T = double
void f1(double* pvar)
{ *pvar = 3.9; }

int main() {
    double var;
    f1(&var);
    cout << var << endl;
    return 0;
}
```

```
// here T = int*
void f2(int dat[], int len, int** pptr)
{
    int maxidx = 0;
    // loop to find the index of max
    *pptr = &dat[maxidx];
}

int main() {
    int dat[10] = { ... };
    int* ptr;
    f2(dat, 10, &ptr);
    cout << "Max: " << *ptr << endl;
    return 0;
}
```

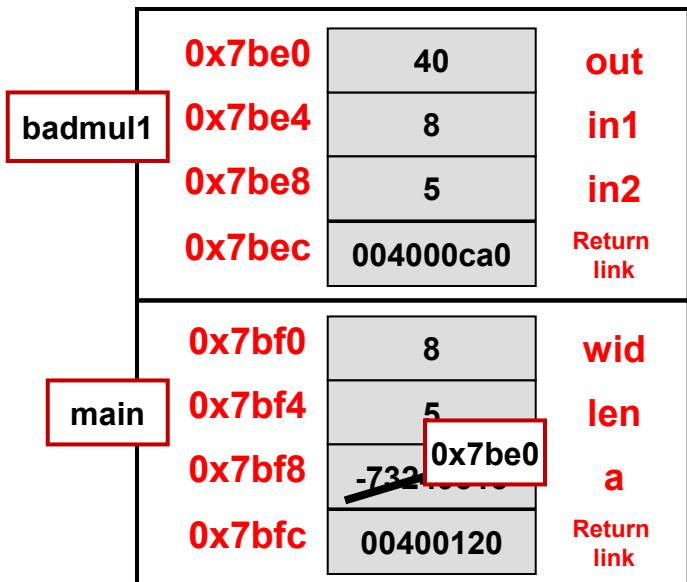
700 720

800

Misuse of Pointers

- Make sure you don't return a pointer to a dead variable
- You might get lucky and find that old value still there, but likely you won't

Stack Area of RAM



```
// Computes the product of in1 & in2
int* badmul1(int in1, int in2);

int main()
{
    int wid = 8, len = 5;
    int *a = badmul1(wid, len);
    cout << "Ans. is " << *a << endl;
    return 0;
}

// Bad! Returns a pointer to a var.
// that will go out of scope
int* badmul1(int in1, int in2)
{
    int out = in1 * in2;
    return &out;
}
```

Prerequisites: Pointer Basics, Data Sizes

POINTER ARITHMETIC AND ARRAYS

Review Questions

- The size of an 'int' is how many bytes?
– _____
- The size of a 'double' is how many bytes?
– _____
- T/F: The elements of an array are stored contiguously in memory
– _____
- In an array of integers, if `dat[0]` lived at address `0x200`, `dat[1]` would live at...?

Big Idea: Array Names \leftrightarrow Pointers

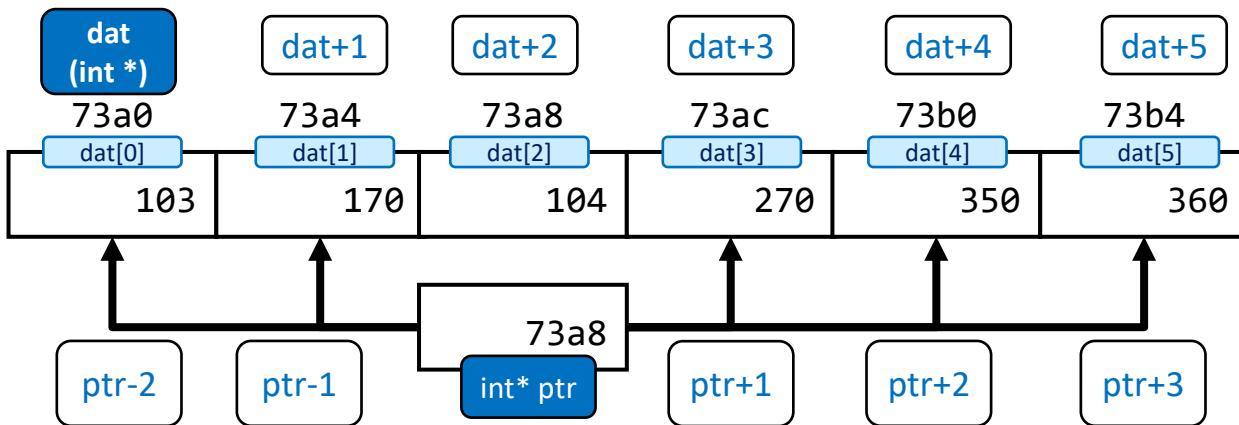
- **Big idea:** Array names and pointers are interchangeable
 - An array name is a pointer and a pointer can be used as an array name!
- Why? Because an array name by itself evaluates to:
 - _____
 - An array name is simply a pointer to the 0th element of that data type (i.e. an `int*`).
 - Given the declaration `int dat[10]`, `dat` is an _____ (type)
 - Given the declaration `char str[6]`, `str` is a _____ (type)
 - A pointer (i.e. `int* ptr;`) can be used as an array name once you point it to some location (see example below)

```
int dat[5] = {10,11,12,13,14};  
*dat = 1; // array name as ptr: same as dat[0] = 1;  
int *p = dat; // array name as ptr: same as int* p = &dat[0]  
p[1] = p[2]-8; // ptr as array name: same as dat[1]=dat[2]-8
```
- This is possible through **pointer arithmetic**.

Pointer Arithmetic

- Logical Progression: Pointers are variables storing addresses => addresses are just numbers => we can perform arithmetic on numbers => we should be able to perform arithmetic on pointers!
- We can perform addition or subtraction on pointer variables (i.e. addresses) just like any other variable. This is known as **pointer arithmetic**.
- **Important Difference:** The number added/subtracted is implicitly scaled (multiplied) by the size of the **type** pointed to, ensuring the resulting address points to a valid data item

```
int dat[6];  
int* ptr = &dat[2];  
ptr1 += <offset>;
```



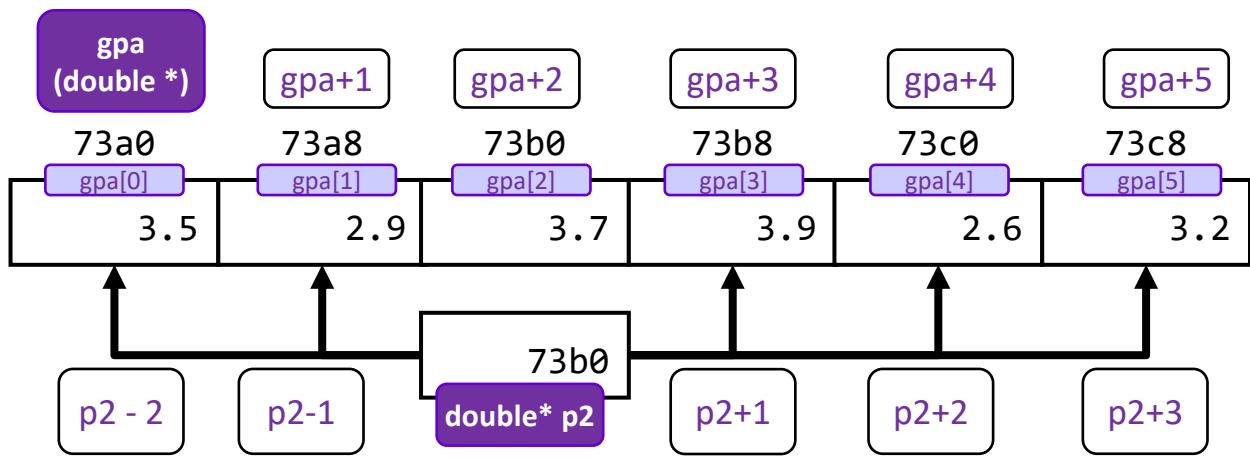
Pointer Arithmetic

- Pointer arithmetic implicitly scales the added value based on the type of pointer
 - For an `int*`, adding `+2` really adds `+2 * sizeof(int) = +2*4 = 8` so that the pointer will point 2 integers away
 - For a `double*`, adding `+2` really adds `+2 * sizeof(double) = +2*8 = 16` so that the pointer will point 2 doubles away

```
double gpa[6];
```

```
double* p2 = gpa+2
```

```
p2 += <offset>;
```



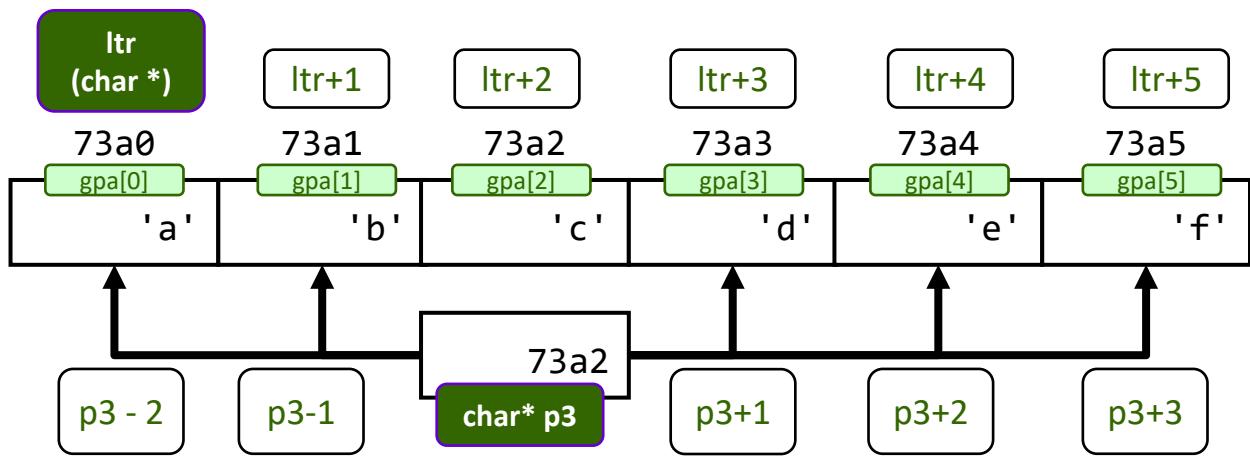
Pointer Arithmetic

- Pointer arithmetic implicitly scales the added value based on the type of pointer
 - For a `char*`, adding `+2` really adds `+2 * sizeof(char) = +2*1 = 2` so that the pointer will point 2 chars away

```
char ltr[6];
```

```
char* p3 = ltr+2
```

```
p3 += <offset>;
```



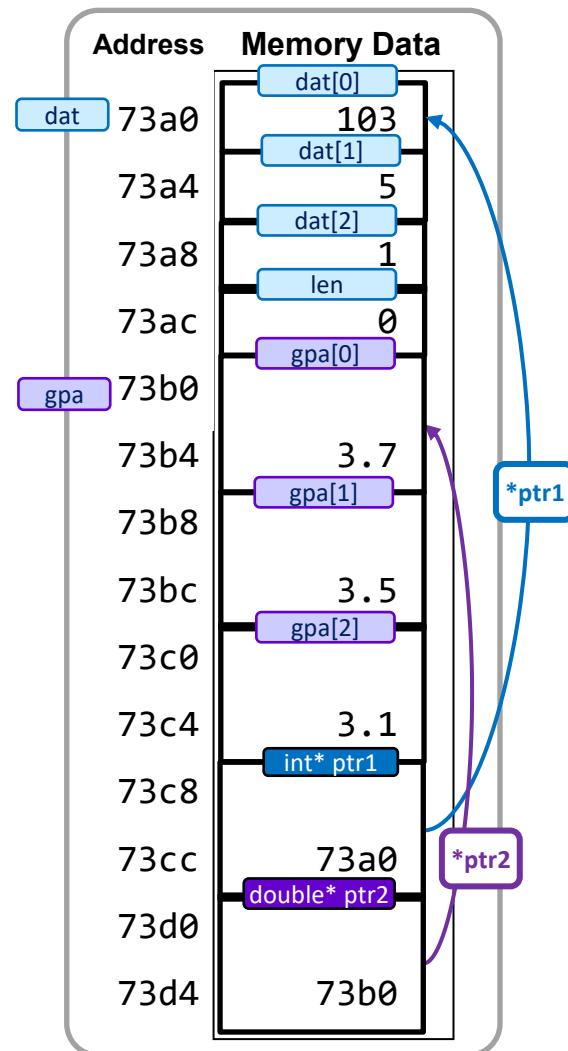
Pointer Arithmetic Examples

- The number added/subtracted to the pointer is implicitly scaled (multiplied) by the size of the type pointed to, ensuring the resulting address points to a valid data item

```

int dat[] = {103, 5, 1}
int len=0;
double gpa[3] = {3.7, 3.5, 3.1};
int *ptr1 = dat;
*ptr1 = 104;
ptr1 = ptr1 + 2; // addr. inc. by __ (2*sizeof(int))
(*ptr1)++; // increment the dereferenced value
ptr1--; // addr. dec. by __ (1*sizeof(int))
double *ptr2 = gpa;
ptr2 += 2; // ptr2 addr. + __ (2*sizeof dbl)
*ptr2++ = 4.0; // set dereferenced value to 4.0 then
                // increment addr. by __ (1*sizeof(double))
// *ptr2 = 2.9; What if??

```



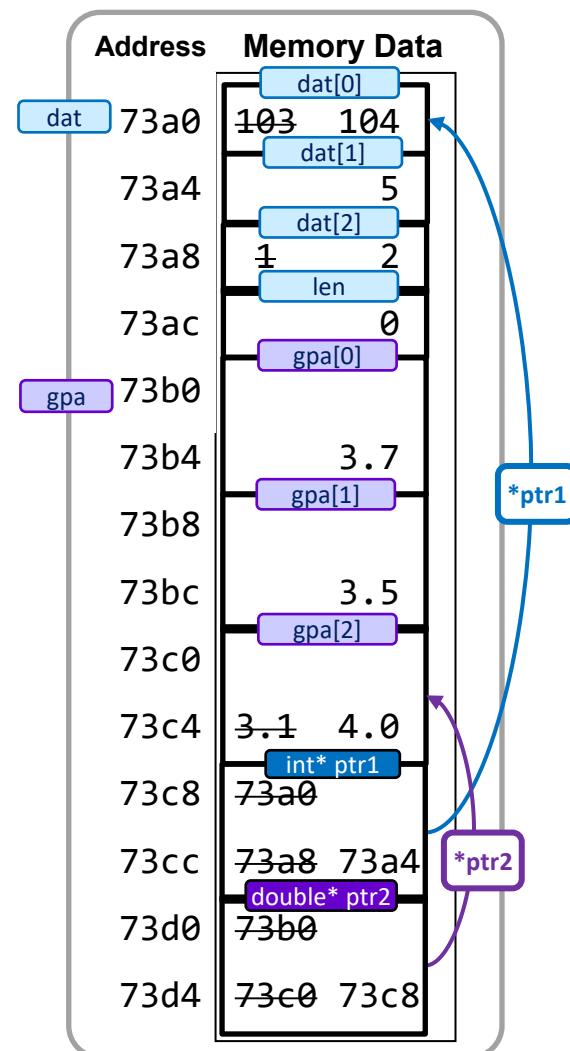
Pointer Arithmetic Examples

- The number added/subtracted to the pointer is implicitly scaled (multiplied) by the size of the type pointed to, ensuring the resulting address points to a valid data item

```

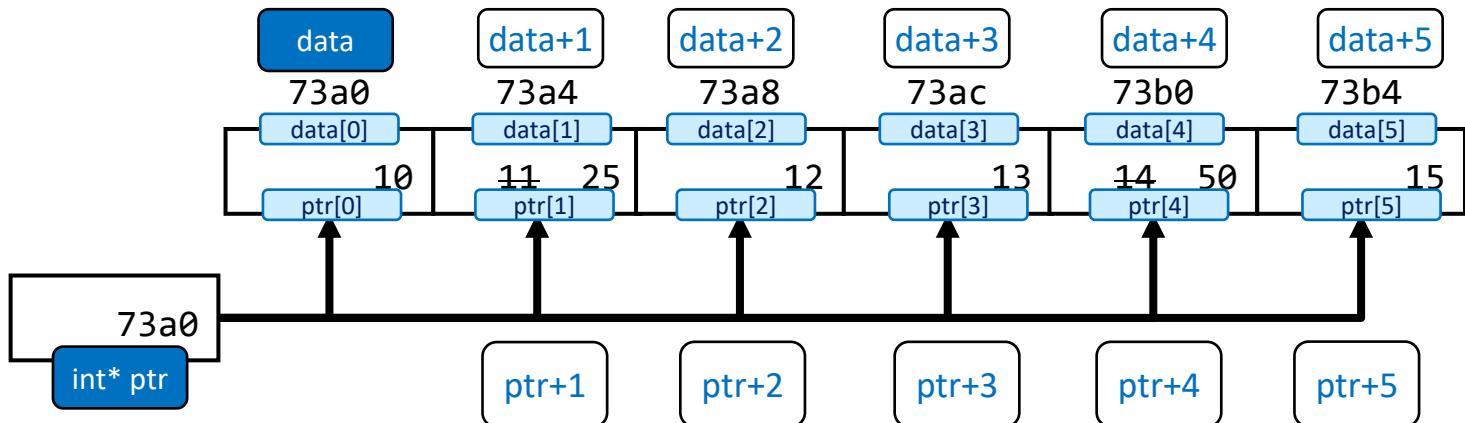
int dat[] = {103, 5, 1}
int len=0;
double gpa[3] = {3.7, 3.5, 3.1};
int *ptr1 = dat;
*ptr1 = 104;
ptr1 = ptr1 + 2; // addr. inc. by 2*4 (2*sizeof(int))
(*ptr1)++; // increment the dereferenced value
ptr1--; // addr. dec. by 1*4 (1*sizeof(int))
double *ptr2 = gpa;
ptr2 += 2; // ptr2 addr. + 2*8 (2*sizeof dbl)
*ptr2++ = 4.0; // set dereferenced value to 4.0 then
                // increment addr. by 1*8 (1*sizeof(double))
// *ptr2 = 2.9; What if??

```



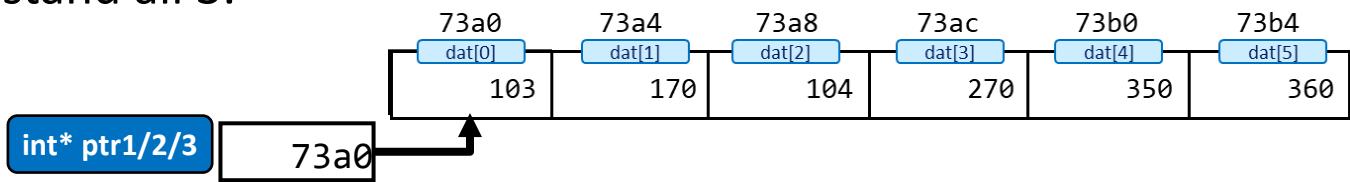
Pointer Arithmetic and Array Indexing

- **Pointer arithmetic and array indexing are really the same!**
- Array syntax: `data[i]`
 - Says **get** the value of the i -th integer in the data array
- Pointer syntax vs. Array syntax: `*(data + i) <=> data[i]`
 - `(data + i)` compute the address of the i -th value in an array and `*` operator gets its value
- We can use pointers and array names interchangeably (an array name is a pointer and a pointer can be treated as an array name and `[]` applied)
 - `int data[6] = {10, 11, 12, 13, 14, 15}; // data = 73a0;`
 - `*(data + 4) = 50; // treat data like a pointer and perform data[4] = 50;`
 - `int* ptr = data; // ptr now points at 73a0 too`
 - `ptr[1] = ptr[2] + ptr[3]; // treat ptr like array name (same as data[1]=data[2]+data[3])`



Arrays vs Pointers

- All 3 methods below perform the same task of initializing the array
 - Which do you prefer?
 - Remember, your goal is to make your code readable (option 1) but you should understand all 3.



Common Array Syntax

```
int main()
{
    int dat[10];
    int *ptr1 = dat;
    // initialize the array
    for(int i=0; i < 10; i++)
    {
        ptr1[i] = 0;
        // equivalent to
        // dat[i] = 0;
    }
    // use the array
}
```

Explicit pointer arithmetic

```
int main()
{
    int dat[10];
    int *ptr2 = dat;
    // initialize the array
    for(int i=0; i < 10; i++)
    {
        *(ptr2+i) = 0;
    }
    // use the array
}
```

"Walking" Pointer

```
int main()
{
    int dat[10];
    int *ptr3 = dat;
    // initialize the array
    for(int i=0; i < 10; i++)
    {
        *ptr3 = 0;
        ptr3++;
    }
    // use the array
}
```

Recall: Passing Arrays as Arguments

- In function declaration / prototype for the *formal* parameter use
 - type [] or type * to indicate an array is being passed
- When calling the function, simply provide the name of the array as the *actual* argument
 - In C/C++ using an array name without any index evaluates to the starting address of the array (a pointer to the 0th element)

```
// Function that takes an array
int sum(int data[], int size);
// or int sum(int* data, int size);

int sum(int data[], int size)
// or int sum(int* data, int size)
{
    int total = 0;
    for(int i=0; i < size; i++){
        total += data[i];
    }
    return total;
}

int main()
{
    int vals[100];
    /* some code to initialize vals */
    int mysum = sum(vals, 100);
    cout << mysum << endl;
    // prints sum of all numbers
    return 0;
}
```

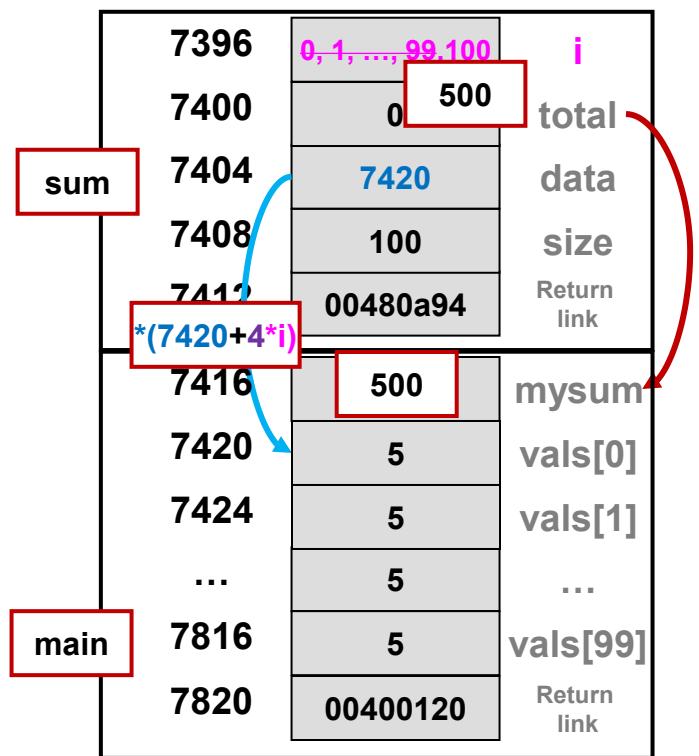
Recall: To access an element in an array, we need 3 pieces of info:

1. Start address of the array
2. Index/offset
3. Type of elements in the array (really the size of that type)



Stack View of Passing Arrays

- Main point: A pointer and an array name are interchangeable!



```

// Function that takes an array
int sum(int data[], int size);
// or int sum(int* data, int size);

int sum(int data[], int size)
// or int sum(int* data, int size)
{
    int total = 0;
    for(int i=0; i < size; i++){
        total += data[i]; // *(data+i)
    }
    return total;
}

int main()
{
    int vals[100];
    /* some code to size vals */
    int mysum = sum(vals, 100);
    cout << mysum << endl;
    // prints sum of all numbers
    return 0;
}

```

One or Many

- Strange question:
 - Is 3240 McClintock Ave. the address of a single-family house or a large dormitory with many suites?
- We can't know.
- In the same way, C/C++ does not differentiate whether a pointer points to a **single variable** or **an array** (i.e. it doesn't have additional syntax)
 - It can only be determined based on how the function uses the pointer (does it treat the pointer as being to an array OR to a single value)
 - Good commenting/documentation should describe this.

```
void f1(int* p)
{ // does p point to one int
  // or an array of ints?
}
```

```
// f1 decrements the integer
// pointed to by p
void f1(int* p)
{
  *p -= 1;
}
```

Pointer to a single variable

```
// f1 sets the array pointed to
// by p to all zeros
void f1(int* p)
{
  for(int i=0; i < 10; i++)
  { p[i] = 0; }
}
```

Pointer to an array

const or non-const

- The **const** modifier on a variable type means it may not be modified or changed after being initialized
- Why would we want that?
 - Because YOU are your OWN WORST ENEMY when programming! You make mistakes. The more we can enlist the compiler to help us catch mistakes, the better
 - If our intention is for a variable not to change, then declare it **const**.
- A const pointer means we can dereference the pointer to GET (view) the data but NOT use the pointer to CHANGE (edit) the data
 - Similar to "Can View" vs. "Can Edit" permission on a Google doc.

```
int main() {  
    const int size = 5;  
    // size cannot be modified  
    size = 6; // Compile Error  
}
```

```
void f1(int* p, int size)  
{  
    for(int i=0; i < size; i++)  
        { p[i] = 0; }  
}
```

Non-Const = "Can Edit"

```
void f1(const int* p, int size)  
{  
    for(int i=0; i < size; i++) {  
        cout << p[i] << endl;  
        p[i] = 0; // compile error  
    }  
}
```

Const = "Can View"



Const or Non-Const Example

- Which parameters of the functions should be marked as const?
- Does size need to be marked const?
 - Would it make sense to try to mark an email attachment as "view only"?
 - No! Since it is already a copy

```
void init(_____  
         int data[], int size)  
// or void init(_____  
         int* data, int size)  
{  
    for(int i=0; i < size; i++){  
        cin >> data[i];  
    }  
}  
int sum(_____  
       int data[], int size)  
// or int sum(_____  
       int* data, int size)  
{  
    int total = 0;  
    for(int i=0; i < size; i++){  
        total += data[i]; // *(data+i)  
    }  
    return total;  
}  
  
int main()  
{  
    int vals[100];  
    init(vals, 100); An int*  
    int mysum = sum(vals, 100);  
    cout << mysum << endl;  
        // prints sum of all numbers  
    return 0;  
}
```

C (not C++) String Function/Library (#include <cstring>)

- A library of functions was provided to perform operations on these character arrays representing strings (<cstring> in C++, <string.h> in C)
 - `int strlen(const char *dest) / int strlen(const char dest[])`
 - `int strcmp(const char *str1, const char *str2);`
 - Return 0 if equal, >0 if first non-equal char in str1 is alphanumerically larger, <0 otherwise
 - `char *strcpy(char *dest, const char *src);`
 - `char *strcat(char *dest, const char *src);`
 - Concatenates src to the end of dest
 - `char *strchr(const char *str, char c);`
 - Finds first occurrence of character 'c' in str returning a pointer to that character or NULL if the character is not found