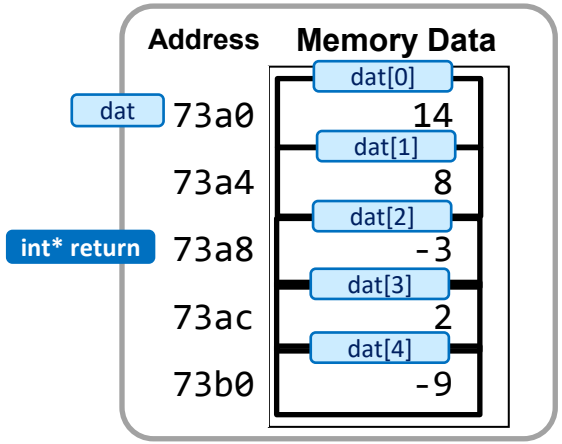
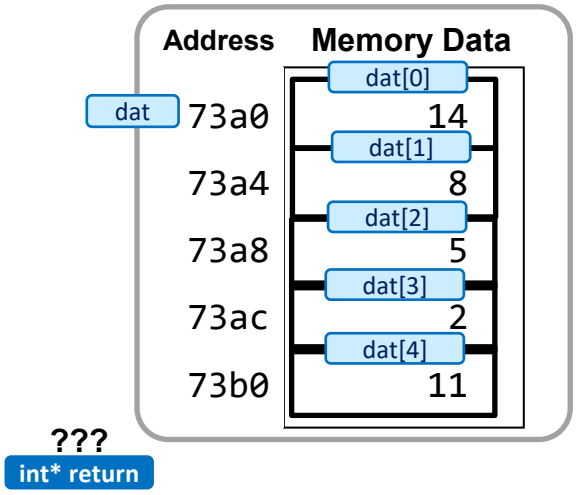
CS103 Unit 2c – Pointers to Pointers, Command line Arguments,

Invalid Pointers

- Suppose I asked you to write a function to return a **pointer** to the first ***negative*** integer in an array
  - `int* firstNeg(int dat[], int len);`
  - What should you return if there is NO negative integer?
- Another example from the <cstring> library is strchr which returns a pointer to the first occurrence of a character in a C-string.
  - `char *strchr(char *str, char c);`
  - What should be returned if the character does not occur in the string?
- It would be nice if there was some address/pointer value we could use to signify "INVALID" or "Bad Address"





NULL and nullptr

- Strange question: Is 3715 McClintock Ave. a valid street address?
- There's no way to really tell?!? Nothing about the address helps us know if it's valid or not (i.e. if any thing lives at that address)
  - Nothing about a memory address (e.g. `0x7fffecb0`) would tell us whether good data resides there
- SO…C/C++ chose **address 0** to mean **INVALID POINTER** and defined the keyword `NULL` (in <cstdlib>) or now `nullptr` (in C++11) as address 0
  - `NULL` or `nullptr` are literally replaced with 0
  - To use `nullptr` compile with the C++11 version:

```
$ g++ -std=c++11 -g -o test test.cpp
```

- You should NEVER dereference a null pointer (will likely cause a crash)!
- Use `NULL` or `nullptr` to:
  - Initialize a pointer variable when you don't know what it should point to yet
  - As a return value when a function can't return a pointer to "good" data
  - So you can write

```
int* p = nullptr;
if( p ){ /* will never get to this code */ }
// or p = firstNeg(...);  if(p != NULL) { /* use p */ }
```

- **NULL and nullptr are substitutes for address 0**
- **They should be used to indicate "INVALID pointer" (nothing good ever lives at address 0)**
- **Similar to the null character ('\0') but this is the NULL pointer**

REVISITING C-STRINGS (CHARACTER ARRAYS)

Constant Strings ⇔ const char*

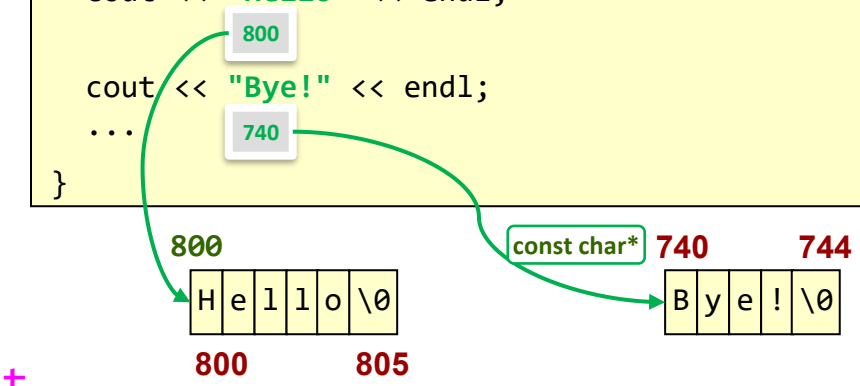
```
#include <iostream>
#include <cstring>
using namespace std;

int main() {
   // What will be printed?
   cout << "Hello" << endl;
   cout << *"Bye" << endl;

   cout << *(("Bye")+1) << endl;
   cout << ("Bye")[1] << endl;
   cout << "Hello"+1 << endl;


   // Try this
   // strcat("Good", "bye");
   // reminder the prototype for strcmp() in C++ documentation is:
   // char * strcat ( char * destination, const char * source );

   return 0;
}
```

C-String Constants

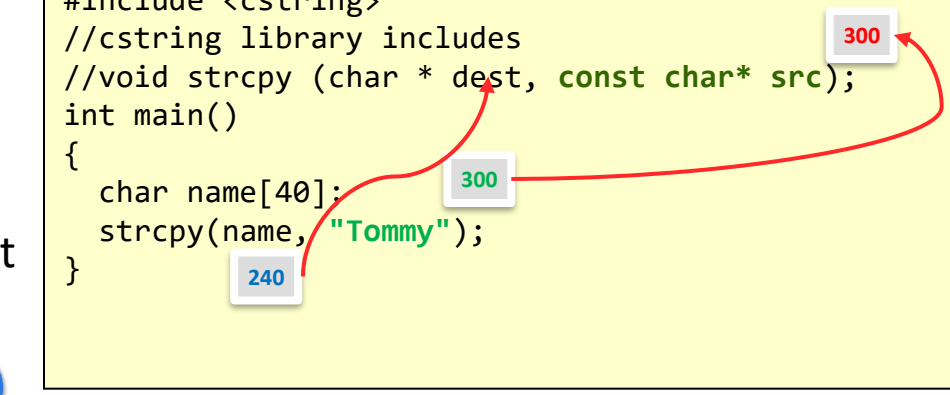
- C-String constants are the things we type in "..." and are stored somewhere in memory (chosen by the compiler and behind the scenes) as a **const character array**
  - **As if:** const anonymous-str1[6] = "Hello";
  - **And:** const anonymous-str2[5] = "Bye!";
- But when you write a string constant (e.g. "Hello"), the computer will give back the starting address where it chose to put the anonymous array and it has the type: const char*
  - char* because an array is ALWAYS known by its starting address in memory
  - const because you cannot/should not change this array's contents
- When you pass a **C-string** constant to

C/C++ considers the type of a string constant (e.g. "Hi") to be **const char***

```
int main()
{
  // These are examples of C-String constants
  cout << "Hello" << endl;
  cout << "Bye!" << endl;
  ...
}
```




```
#include <cstring>
//cstring library includes
//void strcpy (char * dest, const char* src);
int main()
{
  char name[40];
  strcpy(name, "Tommy");
}
```



Knowing vs. Owning an Address

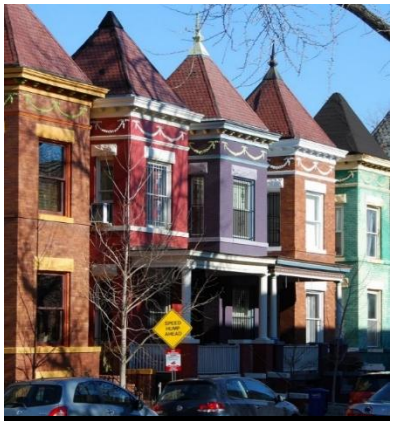
- Scenario:
  - I'm moving.
  - The movers ask: "What's the address of your new place where we should put all this stuff?"
  - I answer, "1600 Pennsylvania Ave."
- What will happen when they go to that address and try to put my stuff there?

- Knowing an address and OWNING the property at that address are VERY DIFFERENT.
  - When we give a pointer, we need to ensure we own the memory that pointer corresponds to

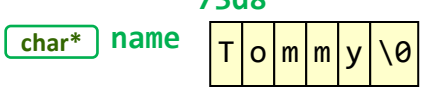
1600 Pennsylvania Ave.

Array / Pointer Relationship

- Owning a home vs. Street address
  - A house is known by its address
    - A house ***definitely*** has an address
  - But an address doesn't necessary correspond to a house
    - I could make up an address where no house exists or an address of a house that does exist but that I don't own
- `char[]` (char array/C-string) vs. `char*` (pointer)
  - A character array (i.e. C-String) like `char name[6]` is known by it starting address (i.e. a `char*`)
    - Array implies a valid pointer (e.g. `name`)
  - A `char*` (pointer) does **NOT** necessarily correspond to a character array (C-String)
    - Pointer does not imply an underlying array




```
char name[6]="Tommy";
```




```
char* p1 = name;
```

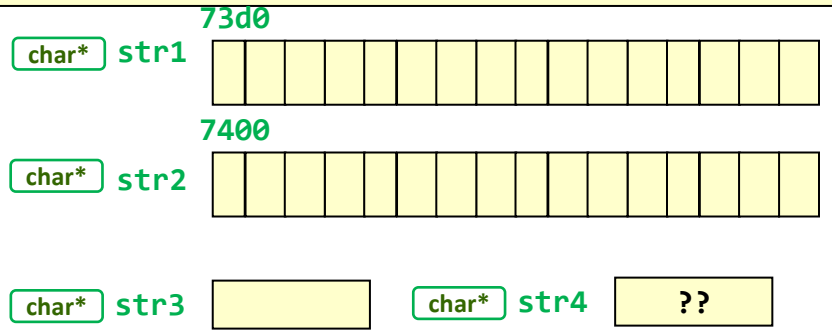

```
char* p2;
```

char* vs. char Array – Common Error

- Examine the code to the right
- Will we be able to copy the strings in each statement?

- Could you just make up an address and go move into that apartment?
  - No! You must allocate/rent space first before you can go to an address and fill it in
- Always **ensure** a **char*** POINTS at an array that you've allocated before you use the pointer

```
#include <iostream>
using namespace std;

int main()
{
  char str1[40] = "hello world";
  char* str2[40];
  char* str3 = str2;
  char* str4;

  // Try to copy
  strcpy(str2, str1); // Does this work?
  strcpy(str3, str1); // Does this work?
  strcpy(str4, str1); // Does this work?
  ...
}
```

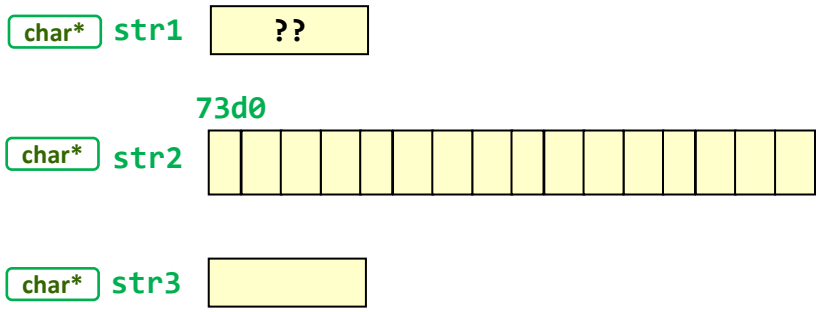


char* vs. char Array – Common Error

- Examine the code to the right
- Will we be able to read in the three strings?

- Could you just make up an address and go move into that apartment?
  - No! You must allocate/rent space first before you can go to an address and fill it in
- Always **ensure** a **char*** POINTS at an array that you've allocated before you use the pointer

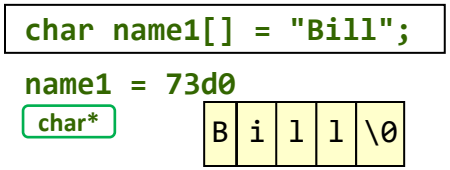
```
#include <iostream>
using namespace std;

int main()
{
  char* str1;
  char str2[40];
  char* str3 = str2;

  // Try to read in strings
  cin >> str1;  // Does this work?
  cin >> str2;  // Does this work?
  cin >> str3;  // Does this work?
  ...
}
```



C-Stings ⇔ char*

- Moving forward, when you see type `char*` you should automatically think: C-String [i.e. a character array terminated with a null character]
  - **BUT, that `char*` must point to an ACTUAL character array.**
- They should be synonymous in your head because…
  - 99% of the time, a `char*` will be used to point at a C-string
  - 1% of the time a `char*` will be pointing at a single character or array of characters that is not terminated with a null (which the documentation should describe)
- Many C/C++ library functions will ASSUME that a char* points at a C-string and treat it differently than other pointers (like int* or double*)
  - cin/cout are the best example

```
char name1[] = "Bill";
```



Expectations!

- All of the `cstring` library functions EXPECT that the `char*` you provide points to a **NULL-terminated character array**
  - `int strlen(const char *dest)`
  - `int strcmp(const char *str1, const char *str2);`
  - `char *strcpy(char *dest, const char *src);`
  - `char *strcat(char *dest, const char *src);`
  - `char *strchr(const char *str, char c);`

Prerequisites: Pointer Basics

POINTERS TO POINTERS

Pointer Analogy

- We can have multiple levels of pointers (indirection)
- Using C/C++ pointer terminology:
  - `*9 = gold in box 7  (9 => 7)`
  - `**16 = gold in box 7  (16 => 9 => 7)`
  - `***5 = gold in box 7  (5 => 16 => 9 => 7)`
- What is stored in one box might be:
  - **[Box 9]**: a **pointer-to** to data
  - **[Box 16]**: a **pointer-to** to a **pointer-to** data
  - **[Box 5]**: a **pointer-to** to a **pointer-to** 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 16
6 11	7	8 4	9 7	10 3	11
12	13 1	14	15	16 9	17 3

Pointer Analogy

- How would you differentiate whether the number in the box was data, a pointer, or a pointer-to-a-pointer?
  - You can't really. Context (**i.e. type**) is needed
- This is why we have to declare something as a pointer and give a type as well:
  - `int* p;` // pointer to an integer one hop (one level of indirection) away
  - `double **q;` // pointer to a double two hops (two levels of indirection) away

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

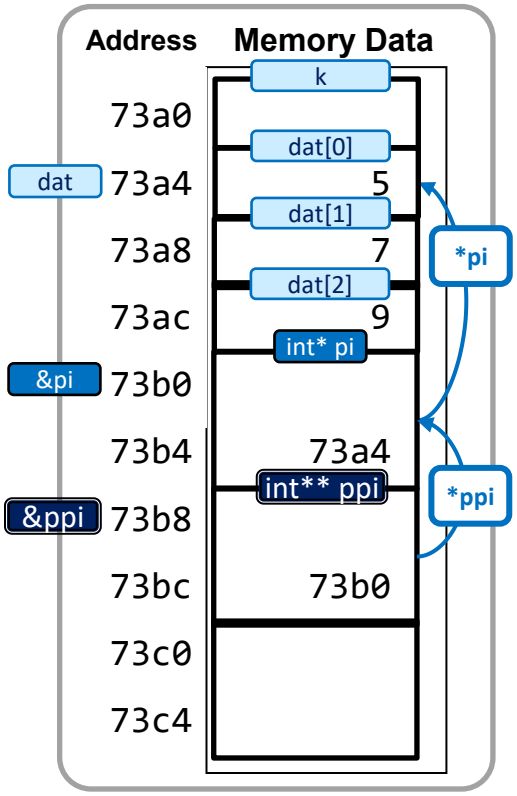
It does not matter if you place the ** next to the type or variable name. The following are the same.

```
double** q;
double **q;
```

Understand the Operations of Pointers to Pointers

- Pointers can point to other pointers
  - Essentially a chain of "links"
- Sample sequence
  - `int k, dat[3] = {5, 7, 9};`
  - `int *pi,`
  - `int **ppi;`
  - `pi = dat;`
  - `ppi = &pi;`
  - `k = *pi;` `// k=__`
  - `k = (**ppi) + 1;` `// k=__`
  - `k = *(*ppi + 1);` `// k=__`

This code does nothing useful and is just for illustration.



Recall:
- * with a type to the left is declaring a pointer (e.g. int *pi)
- * with NO type to the left is dereferencing the pointer (e.g k = *pi)

Check Yourself

- Consider these declarations:
  - `int k, dat[3] = {5, 7, 9};`
  - `int *pi = x, **ppi = &pi;`

- In declarations, the type (e.g. int) distributes when you declare multiple variables, but the *s do NOT! (@!**-ing C/C++ ☺)

- Tip: As a sanity check when you write code, ensure the types match on either side of an operator or an assignment
(e.g. `x = y; // are x and y // types compatible/the same`)

- Indicate the formal type that each expression evaluates to (i.e. `int, int *, int **`)

To figure out the type of data a pointer expression will yield…
- **Each * in the expression cancels a * from the variable type.**
- **Each & in the expression adds a * to the variable type.**

Orig. Type	Expr	Yields
`pi = int*`	`*pi`	`int`
`ppi = int**`	`**ppi`	`int`
`ppi = int**`	`*ppi`	`int*`
`k = int`	`&k`	`int*`
`pi = int*`	`&pi`	`int**`

Expression	Type
`&pi`	
`dat`	
`&k`	
`pi`	
`*pi`	
`pi + 2`	
`(*ppi) + 1`	
`&ppi`	

Understanding Types With & or * (Skip for time)

& operator (Address-of)

- Applying & to a variable of type T yields: a type T* result
- That is to say: & adds a * to the resulting type
  - `int x;` int
  - `double z;` double
  - `int *ptr1 = &x;`
    `// & int => int*`
  - `double* ptr2 = &z;`
    `// & double => double*`
  - `int **ptr3 = &ptr1;`
    `// & int* => int**`

* Operator (Dereference)

- Apply * to a variable of type T* yields a type T result (every * in the expression **cancels** a * from the type of variable)
  - `int a = *ptr1;`
    `// * int* => int`
  - `*ptr2 = 1.25;`
    `// * double* => double`
  - `*ptr3 = ptr1;`
    `// * int** => int*`
  - `**ptr3 = 5;`
    `// ** int** => int`

ARRAYS OF POINTERS

Recall: 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
- But for now (and in many contexts), a pointer will be pointing to an **array**!

```
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

Big Idea (T* ⇔ T[ ])

- In many cases (though not all), a pointer is synonymous with or implies an array (since the name of an array yields a pointer to it)
  - A `char* ⇔ char[]`
    - `name is a char*`
  - An `int* ⇔ int[]`
    - `dat is an int*`

```
char name[6] = "Anika";
```

char* **name=73d8**

A	n	i	k	a	\0

```
int dat[4] = {3,5,8,2};
```

int* **dat=7420**

3	5	4	2

- So what would an ***array*** of `T*` (e.g. `T* mat[]`) be?
  - An _____ of _____ = _____ array

Multidimensional Arrays

- Thus far, arrays can be thought of 1-dimensional (linear) sequences
  - only indexed with 1 value (coordinate)
  - `char dat[6] = {1,2,3,4,5,6};`
- We often want to view our data as 2D, 3D or higher dimensional data
  - Matrix data
  - Images (2D or 3D)
  - Index w/ 2 coordinates (row,col)

	Address	Mem.
	7419	ab
dat	7420	01
	7421	02
	7422	03
	7423	04
	7424	05
	7425	06
	7426	fe

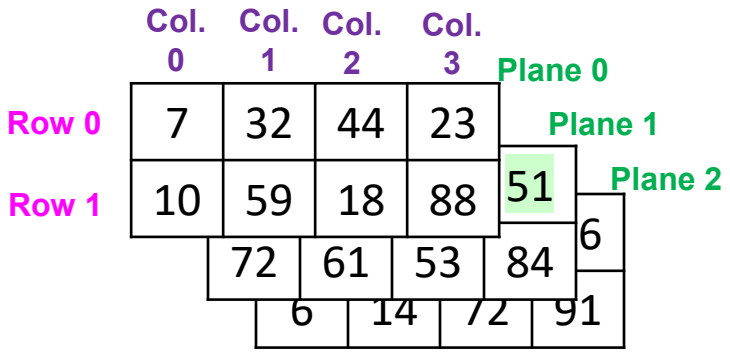
Image taken from the photo "Robin Jeffers at Ton House" (1927) by Edward Weston

Multidimension Array Declaration

- 2D: Declare by providing size along both dimensions (normally rows first then columns) and access with 2 indices
  - Declaration: `int matrix[2][3];`
  - Access elements with appropriate indices
    - `my_matrix[0][1]` evaluates to 3, `my_matrix [1][2]` evals to 2

	Col. 0	Col. 1	Col. 2
Row 0	5	3	1
Row 1	6	4	2

- 3D: Declare and access data with 3 indices
  - Declaration: `unsigned char image[2][4][3];`
  - Access elements with appropriate indices
    - `image[0][3][1]` evals to 51

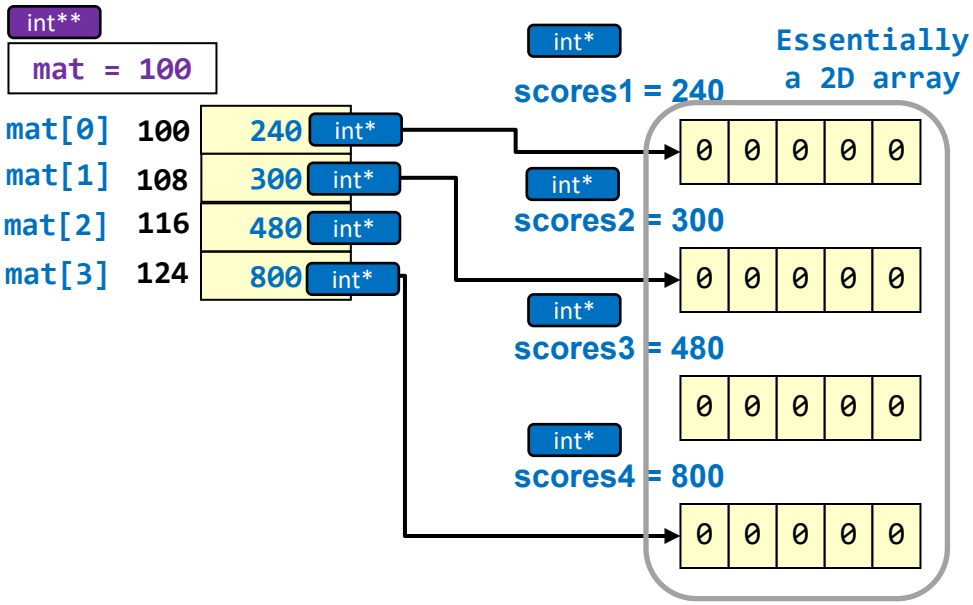


Limitations of Multidimensional Arrays

- Just like 1D arrays, multidimensional arrays must be declared of a FIXED, constant size and NOT a variable size
  - Legal Declaration: `int matrix[2][3];`
  - Legal Declaration: `unsigned char image[2][4][3];`
  - Illegal Declaration: `int matrix[n][m];`
  - Illegal Declaration : `unsigned char image[NX][NY][NZ];`
- Also, C++ new CANNOT even dynamically allocate a 2- or 3-D array (for reasons we'll explain in a future unit)
  - Does NOT work: `new matrix[n][m];`
  - Does NOT work : `new unsigned char[NX][NY][NZ];`
- **But there is a way! Arrays of pointers.**
  - Use many 1D-arrays and an array of pointers

Pointers-To-Pointers ⇔ 2D Arrays

- If 1D array is known by its starting address (i.e. a T* pointer)…
  - e.g. `int scores1[5];  // scores1 is an int*`
  - Suppose we had a few more integer arrays (scores2, scores3, etc.)
- …Then an ARRAY of pointers could be thought of as:
  an **ARRAY of ARRAYS (i.e. 2D array)**
  - e.g. `int* mat[4];`
- What would `mat` be?
  - An array's type is a pointer (T[ ] ⇔ T*), so apply substitution:
    T = int*, thus…
- `mat is an int**`

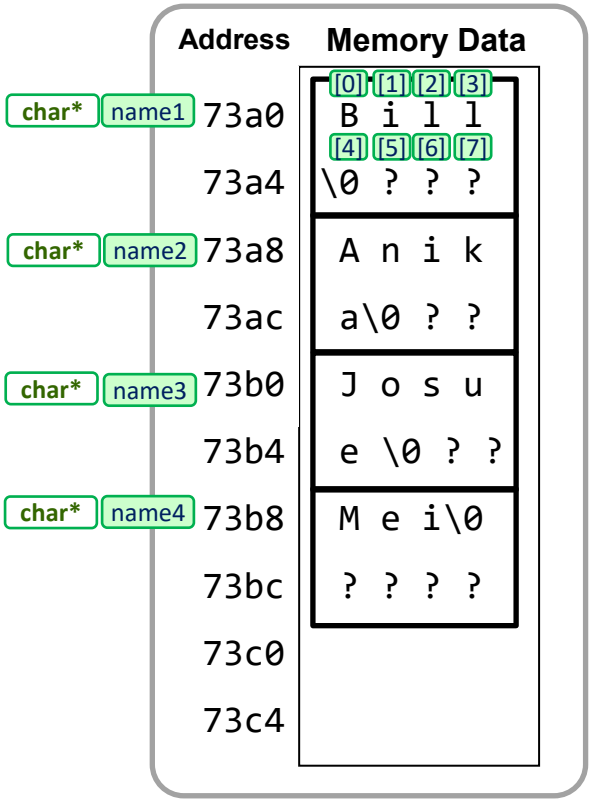


Arrays of pointers

- We often want to have several arrays of related data
  - Store several text strings (names of contestants)
  - We could declare each array separately but then we couldn't use a loop to process them

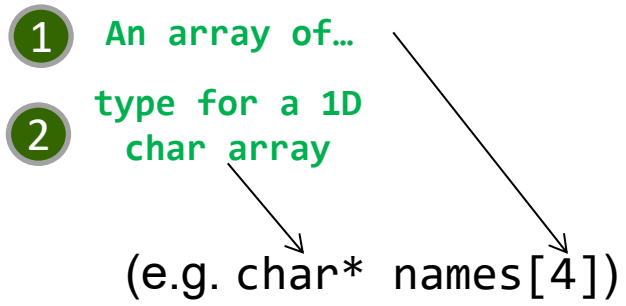
```
int main(int argc, char *argv[])
{
  char name1[8] = "Bill";
  char name2[8] = "Anika";
  char name3[8] = "Josue";
  char name4[8] = "Mei";

  // I would like to print out each name
  cout << name1 << endl;
  cout << name2 << endl;
  ...
}
```

Painful!

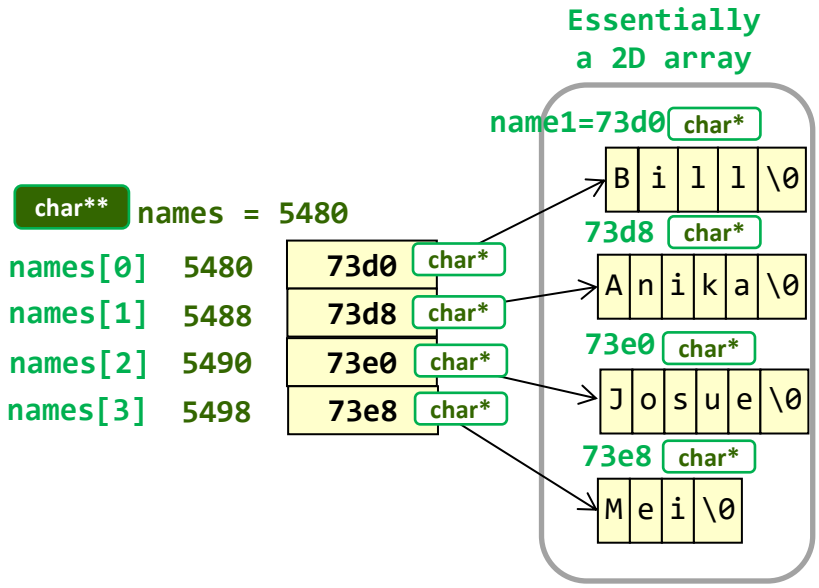


Pointers-To-Pointers ⇔ 2D Arrays

- Since a char* is the type for a 1D character array (C-String), an array of char* (e.g. char* [ ]) is like an array of arrays…
- …which is a 2D array



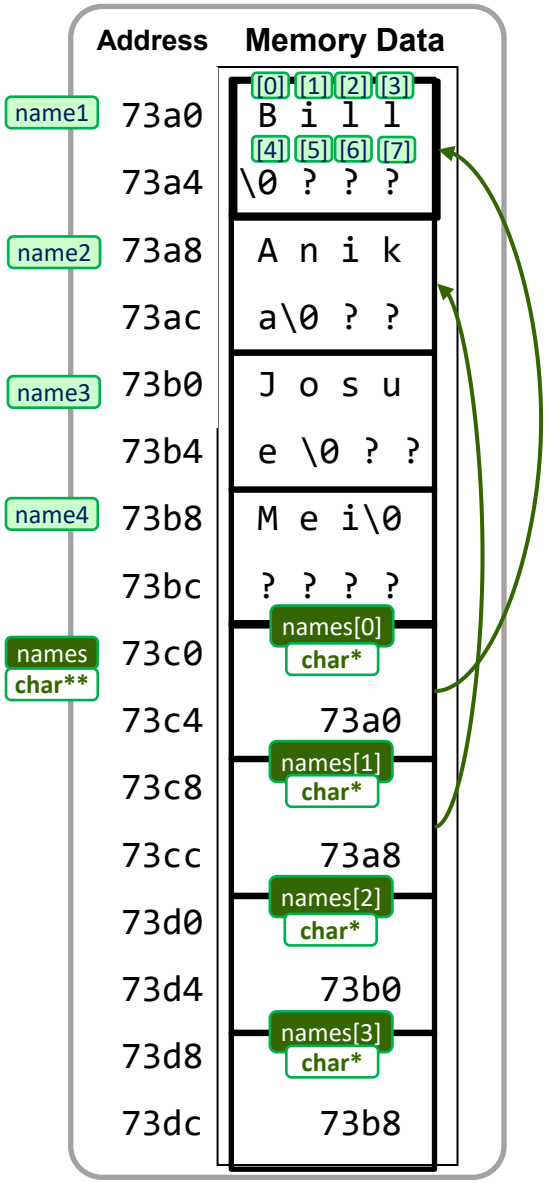

```
int main()
{
  char name1[8] = "Bill";  // name1 is a char*
  char name2[8] = "Anika";
  char name3[8] = "Josue";
  char name4[8] = "Mei";

  char* names[4] = {name1, name2, name3, name4};
  for(int i = 0; i < 4; i++){
    cout << names[i] << endl;
    // what type would 'names' be?
  }
  return 0;
}
```



Arrays of Pointers

- In essence, we want an array of arrays but in C/C++, this is usually accomplished as an array of pointers (to the individual arrays)
- What type is 'names'?
  - The address of the 0-th `char*` in the array
  - The address of a `char*` is really just a `char**`

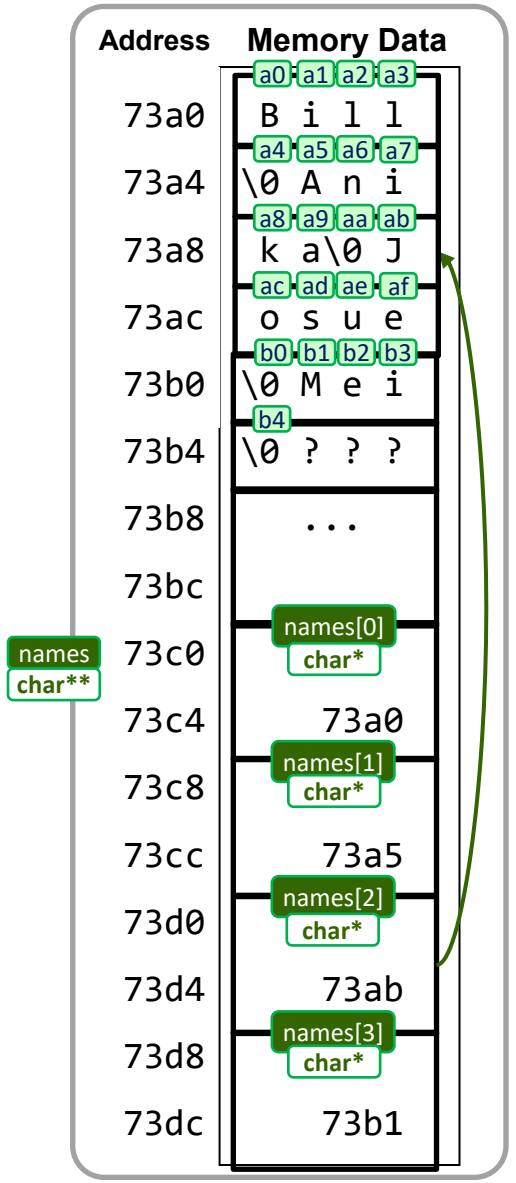
```
int main() {
  char name1[8] = "Bill";  // name1 is a char*
  char name2[8] = "Anika";
  char name3[8] = "Josue";
  char name4[8] = "Mei";

  char* names[4] = {name1, name2, name3, name4};
  for(int i = 0; i < 4; i++){
    cout << names[i] << endl;
    // what type would 'names' be?
  }
  return 0;
}
```



Arrays of constant C-Strings

- We can have arrays of pointers just like we have arrays of other data types
- An array of pointers is most commonly used to create an array of arrays (or 2D array) where each pointer points at separate array.

```
int main(int argc, char *argv[])
{
  const char *names[4] =
     { "Bill", "Anika", "Josue", "Mei" };

  for(i=0; i < 4; i++){
    cout << names[i] << endl;
  }

}
```



Intro to Command Line Arguments

- Currently, what method do we use to get input?
  - cin
- Method 1: `cin`
  - But this requires human interaction each time the program is run
  - But to support scripts (automation) and for other reasons, another method is commonly used
- Method 2: command line arguments
  - Data is input on the command line when we launch the application

```
int main()
{
  int seed;
  double threshold;
  // keyboard input
  cout << "Enter an int and double: ";
  cout << endl;
  cin >> seed >> threshold;

}
```

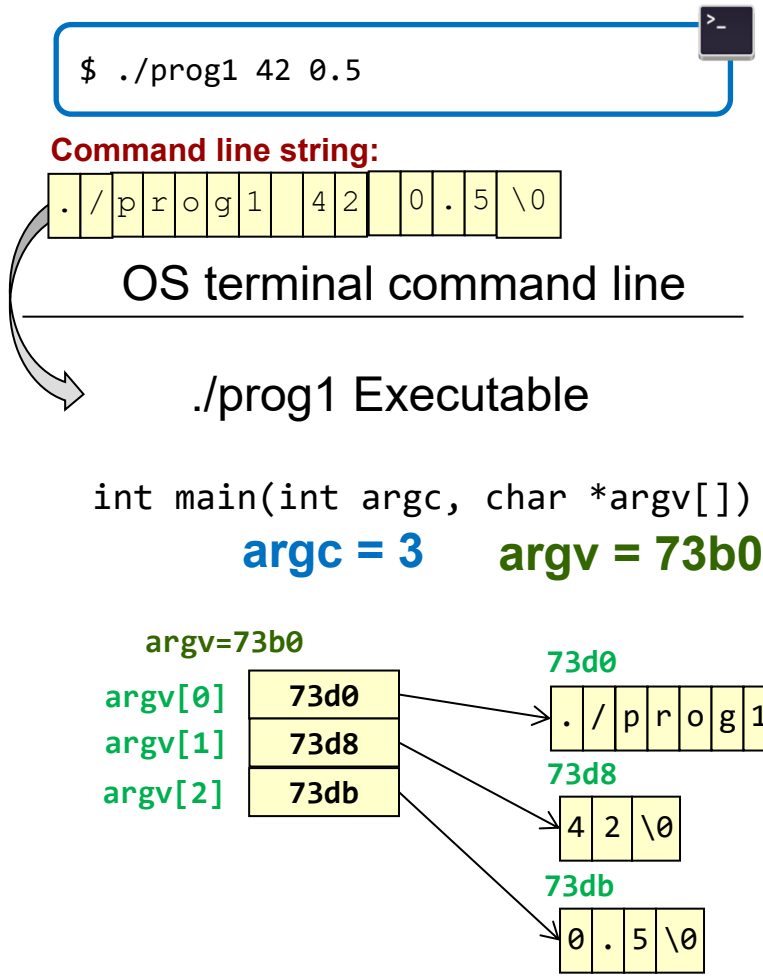
```
$ ./prog1
Enter an int and double:
42 0.5
```

```
int main(int argc, char *argv[])
{
  if(argc < 3) {
    cout << "Not enough inputs" << endl;
    return 1;
  }
  int seed = atoi(argv[1]);
  double threshold = atof(argv[2]);

}
```

```
$ ./prog1 42 0.5
```

Command Line Arguments

- Now we can understand the arguments passed to the main function [i.e. `main(int argc, char *argv[])`]
- At the command prompt we can give inputs to our program rather than making querying the user interactively:
  - `$ ./prog1 42 0.5`
  - `$ g++ -g app.cpp -o app`
- OS processes the command line string, breaking it at whitespaces, and copying it into individual strings packaged into an array (argv) of pointers to those strings
  - Each entry is a pointer to a string (`char *`)
- `argc` indicates the length of the `argv` array
- 0th entry (`argv[0]`) is always the string containing the program executable name

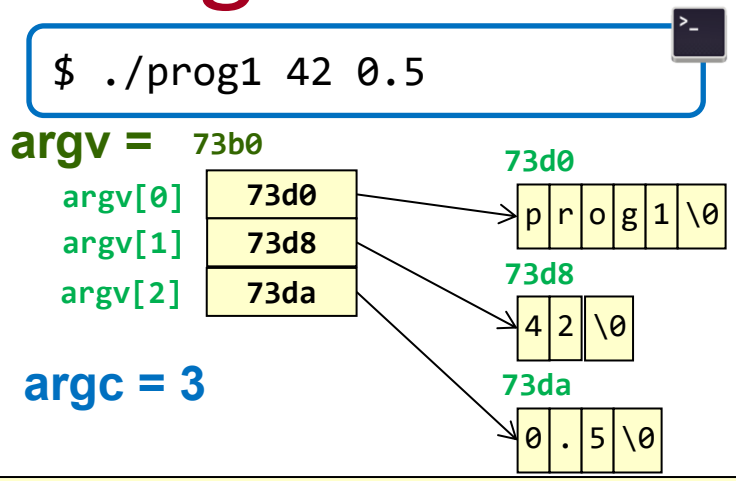


Using Command Line Arguments

- Step 1: Update `main()`'s signature to:

```
int main(int argc, char* argv[]) or
int main(int argc, char** argv)
```

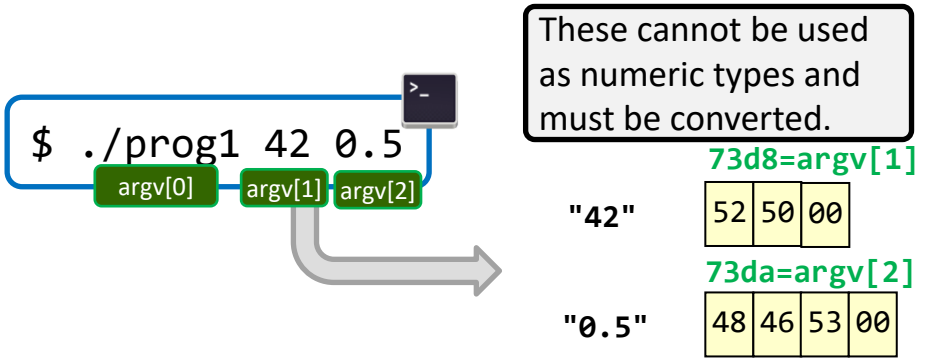
- Step 2: Check `argc` to ensure the user provides *enough* arguments when they start the program
  - Remember, the program name is one of the arguments
- Step 3: Use library functions to convert strings to `int`s or `double`s as needed for numeric inputs
  - atoi() or strtol() for ints
  - atof() or strtod() for doubles
  - All are in `<cstdlib>`
  - No need to convert text arguments




```
#include <iostream>
#include <csdtlib>
using namespace std;
//could also be => char **argv
int main(int argc, char *argv[])
{
  if(argc < 3) {
    cout << "Not enough inputs" << endl;
    return 1;
  }
  int seed = atoi(argv[1]); // "42"=>42
  double threshold =
              atof(argv[2]); // "0.5"=>0.5
  // use seed and threshold
  // ...
}
```

Converting C-Strings to Numeric Types

- Recall that text strings are represented as ASCII characters
  - ASCII digits have codes like 48='0', 49='1', etc.
- <cstdlib> provides functions that take char* (character string) inputs and convert to int or double types.
  - atoi() or strtol() for ints
    - ASCII to Integer and String to Long Int
  - atof() or strtod() for doubles
    - ASCII to float and String to Double
- No need to convert text arguments




```
#include <iostream>
#include <csdtlib>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc < 3) {
    cout << "Not enough inputs" << endl;
    return 1;
  }
  int seed = strtol(argv[1]); // "42"=>42
  double threshold =
            strtod(argv[2]); // "0.5"=>0.5
  // use seed and threshold
  // ...
}
```

strtol() and strtod() are newer and preferred to atoi() and atof() with extra features and error handling

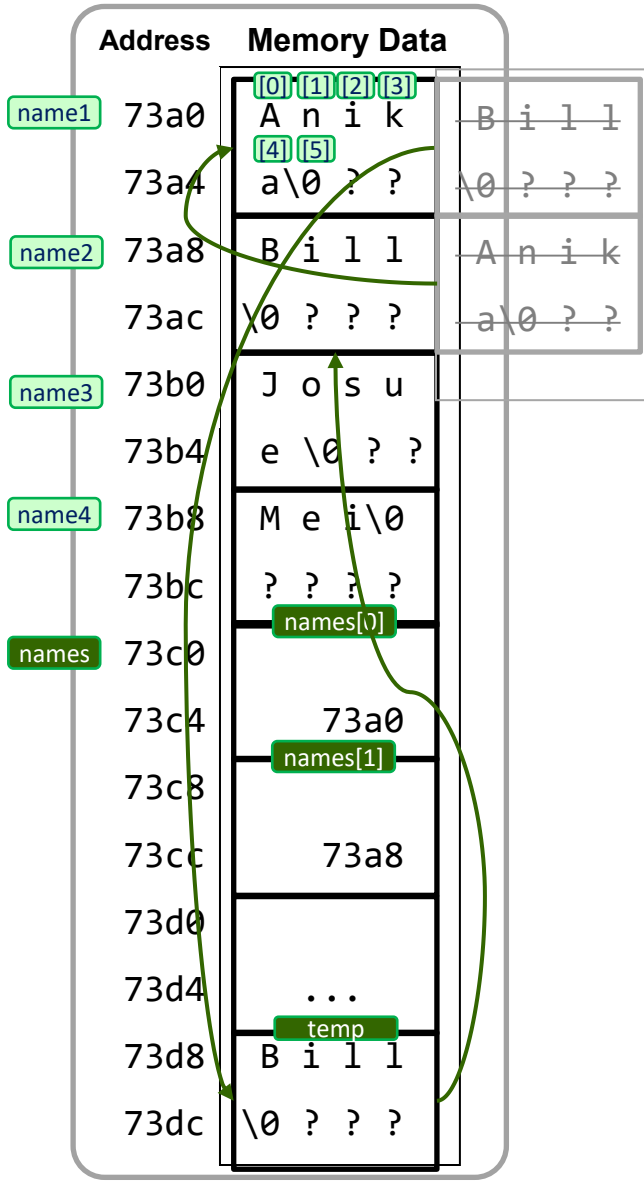
Exercises

- Cmdargs_sum
- Cmdargs_smartsum
- Cmdargs_smartsum_str
- toi

Why Pointers to Arrays? (1)

- Suppose we now wanted to alter the order of the arrays
  - Change order from "Bill", "Annika", etc. to "Annika", "Bill", …
- We could move/swap the data itself, but this could be inefficient (especially for larger arrays)

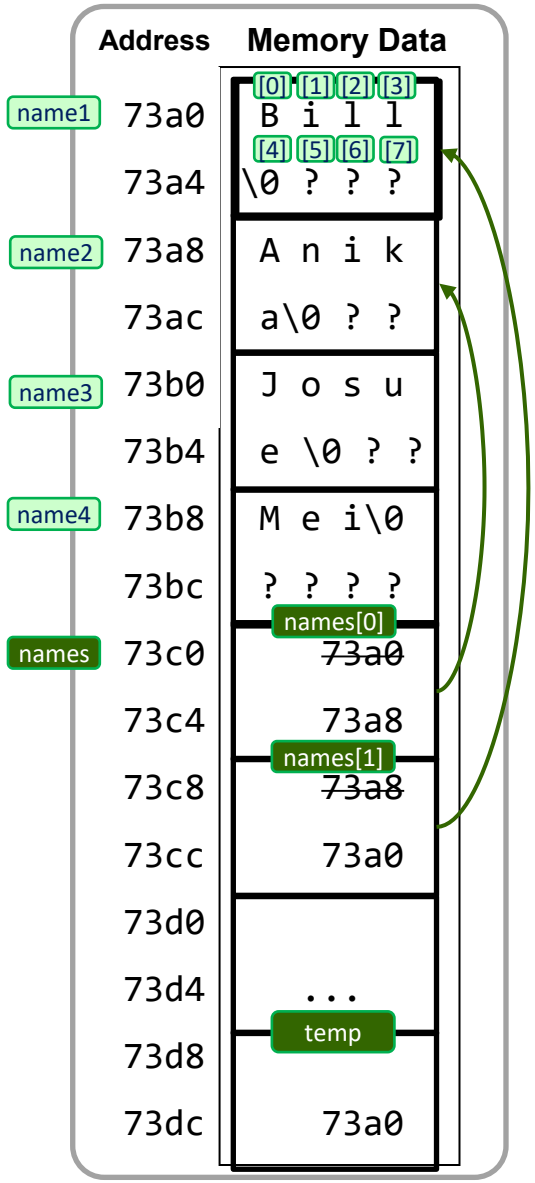
```
int main(int argc, char *argv[])
{
  char name1[8] = "Bill";  char name2[8] = "Anika";
  char name3[8] = "Josue"; char name4[8] = "Mei";

  char *names[4] = {name1, name2, name3, name4};

  char temp[8];
  strcpy(temp, names[0]);
  strcpy(names[0], names[1]);
  strcpy(names[1], temp);

  for(int i=0; i < 4; i++) {
     cout << names[i] << endl;
  }
}
```



Why Pointers to Arrays? (2)

- Or we could simply rearrange pointers which would be independent of the size of the array or object pointed to, and thus be quite fast.

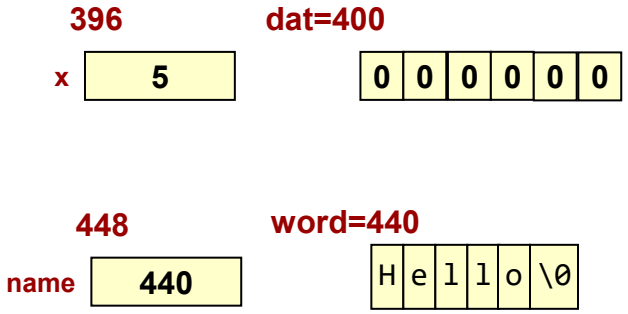
```
int main(int argc, char *argv[])
{
  char name1[8] = "Bill";  char name2[8] = "Anika";
  char name3[8] = "Josue"; char name4[8] = "Mei";

  char *names[4] = {name1, name2, name3, name4};
  char* temp = names[0];
  names[0] = names[1];
  names[1] = temp;

  for(int i=0; i < 4; i++) {
     cout << names[i] << endl;
  }
}
```



IF TIME

cin/cout & char*s

- `cin/cout` determine everything they do based on the **type** of data passed
- `cin/cout` have a unique relationship with `char*s`
- When `cout` is given a variable or expression of any type, it will print the value stored in that exact variable
  - Exception: When `cout` is given a `char*` it will assume it is pointing at a C-string, go to that address, and loop through each character, printing them out
- When `cin` is given a variable it will store the input data in that exact variable
  - Exception: When `cin` is given a `char*` it will assume it is pointing at a C-string, go to that address, and place the typed characters in that memory

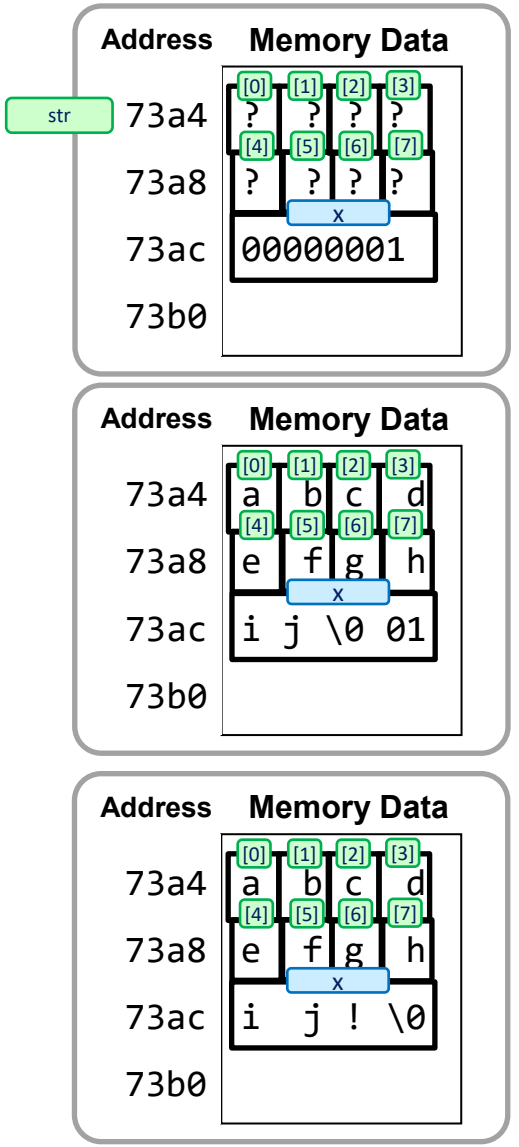



```
#include <iostream>
using namespace std;
int main()
{
  int x = 5, dat[10] = {0}; // dat is an int*
  char word[10] = "Hello";
  char *name = word;

  cout << x << endl;        // 5
  cout << dat << endl;      // 400
  cout << word << endl;     // Hello
  cout << name << endl;     // Hello
  cout << name[0] << endl; // H
  cout << (void*) name << endl;  // 440

  cin >> dat;     // Doesn't work, use a loop
  cin >> name;    // Store many chars
                  // starting at 440
  return 0;
}
```

C-String Library Vulnerabilities

- What could go wrong with these library functions?
  - Consider the code below.

```
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
  char str1[8];
  int x = 1;
  strcpy(str1, "abcdefghij");

  strcat(str1, "!");

  cout << str3 << endl;

  return 0;
}
```



Safe C-String Library

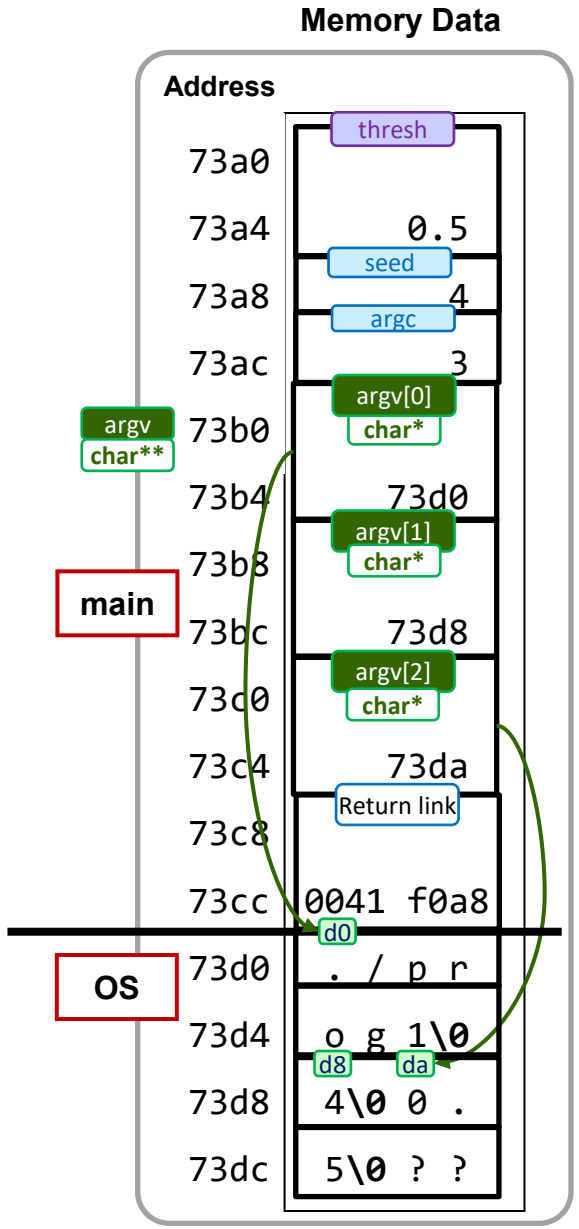
- The `<cstring>` library was updated in subsequent versions of C++ to provide safer alternatives to avoid array buffer overflows with many functions now having a counterpart with `n` in the name representing a maximum length to read or write.
  - `int strlen(const char *dest)`
  - `int strncmp(const char *str1, const char *str2, size_t num);`
    - Return 0 if equal, >0 if first non-equal char in str1 is alphanumerically larger, <0 otherwise
    - Compares a maximum of n characters (which should match the length of the shortest input)
  - `char *strncpy(char *dest, const char *src, size_t num);`
    - Maximum of num characters copied
  - `char *strncat(char *dest, const char *src, size_t num);`
    - Maximum of num characters concatenated plus a NULL
- See the documentation (https://cplusplus.com/reference/cstring/)

A Stack View

- Here is a memory/stack view of the command line arguments, and the argc, argv arguments passed to main()

```
$ ./prog1 4 0.5
```

```
#include <iostream>
#include <csdtlib>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc < 3) {
    cout << "Not enough inputs" << endl;
    return 1;
  }
  int seed = strtol(argv[1]);
  double threshold = strtod(argv[2]);
  // use seed and threshold
  // ...
}
```



BACKUP

Why Pointers To Arrays

- 4 friends got sequential hotel rooms.
- Alice hates that her room number is 413 and would like to be "next" to her friend Gina.
- She asks Tim to swap rooms. Tim doesn't care but DOESN'T want to move all his stuff.
- Kyle has an idea to "satisfy" both. Can you guess his approach?