CS103 Unit 1d – Arguments Pass-by-Value and Pass-by-Reference

PASS-BY-VALUE, LOCAL VARIABLES, AND SCOPE

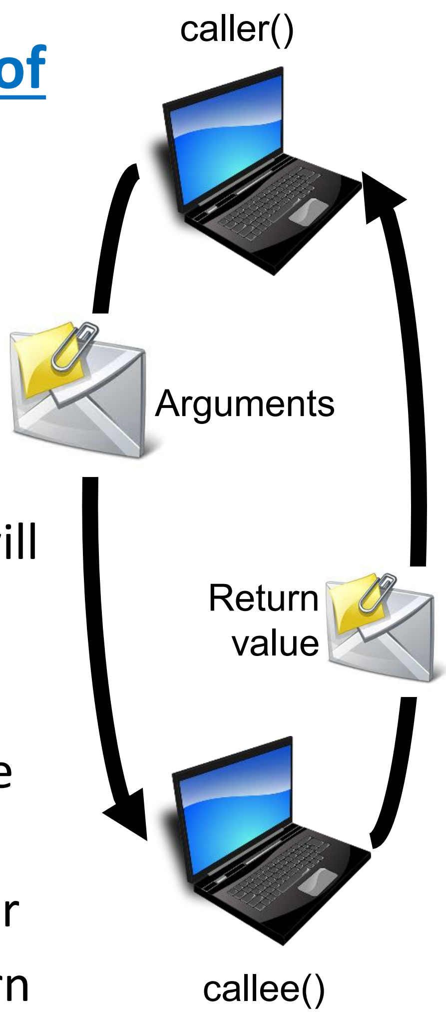
Motivating Question

- What will this code print?

```
void dec(int);

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

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

Argument Passing (Pass-by-Value)

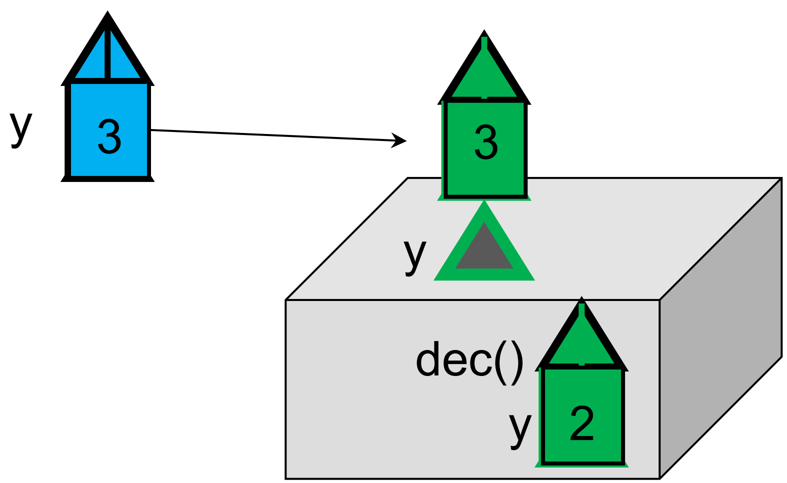
- Passing an argument to a function **makes a copy of the argument**
  - In fancy CS-lingo, we call this **pass-by-value**
- Pass-by-value is like e-mailing an attached document
  - You still have the original on your PC
  - The recipient has a **copy** which she can modify, but it will not be reflected in your version
- Communication is essentially one-way
  - Caller communicates arguments to callee, but these are copies.
  - Any processing the callee does is not visible to the caller
  - The only communication back to the caller is via a return value.



Pass by Value (1)

- **Fact**: Function **arguments/parameters act like local variables to that function**
  - They are only in scope (only live) in the function {…} (curly braces) and then get deallocated.
- When arguments are passed a **copy** of the actual argument value (e.g. 3) is given to the function's input argument
  - So, the function is operating on a copy and that copy will die when the function ends!

```
void dec(int);
int main()
{
  int y = 3;
  dec(y);
  cout << y << endl;
  return 0;
}
void dec(int y)
{
   y--;
}
```



Pass by Value (2)

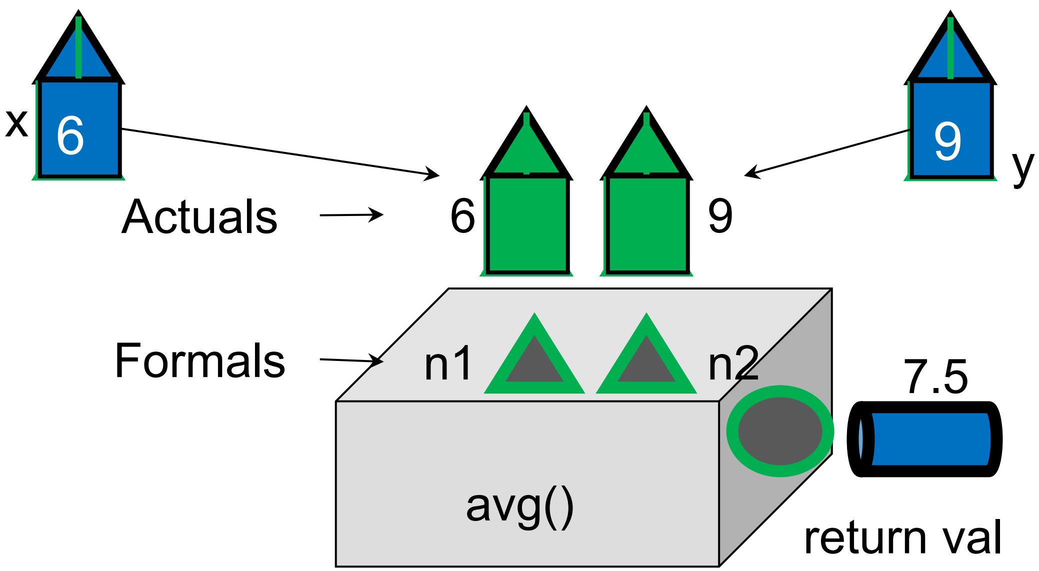
- Wait! But they have the same name, 'y'
  - What's in a name…Each function is a separate entity and so two 'y' variables exist (one in main and one in decrement it)
  - The only way to communicate back to main is via return
  - Try to change the code appropriately
- **Main Point:** Each function is a completely separate "sandbox" (i.e. is isolated from other functions and their data) and copies of data are passed and returned between them

```
void dec(int);
int main()
{
  int y = 3;
  dec(y);
  cout << y << endl;
  return 0;
}
void dec(int y)
{
   y--;
}
```

```
____ dec(int);
int main()
{
  int y = 3;
  ____________ dec(y);
  cout << y << endl;
  return 0;
}
___ dec(int y)
{
   y--;

   ____________________
}
```

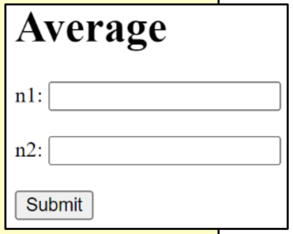
Formals and Actuals (1)

- **Formal** parameters, n1 and n2
  - Placeholder names that will be used internally to the function to refer to the values passed (Similar to how generic placeholders/titles used in contracts like "CEO" or "professor" that will be assigned or replaced real value)
- **Actual** parameters, x and y
  - Actual values to be passed (i.e. the actual values to be substituted for the placeholders ("Jeff Bezos", "Mark")
  - A **copy** is made and given to function




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

double avg(int n1, int n2)
{
  double sum = n1 + n2;
  return sum/2.0;
}

int main()
{
   int x=6, y = 9;   double z;
   z = avg(x, y);
   cout << "AVG is " << z << endl;
   z = avg(x, 2);
   cout << "AVG is " << z << endl;
   return 0;
}
```

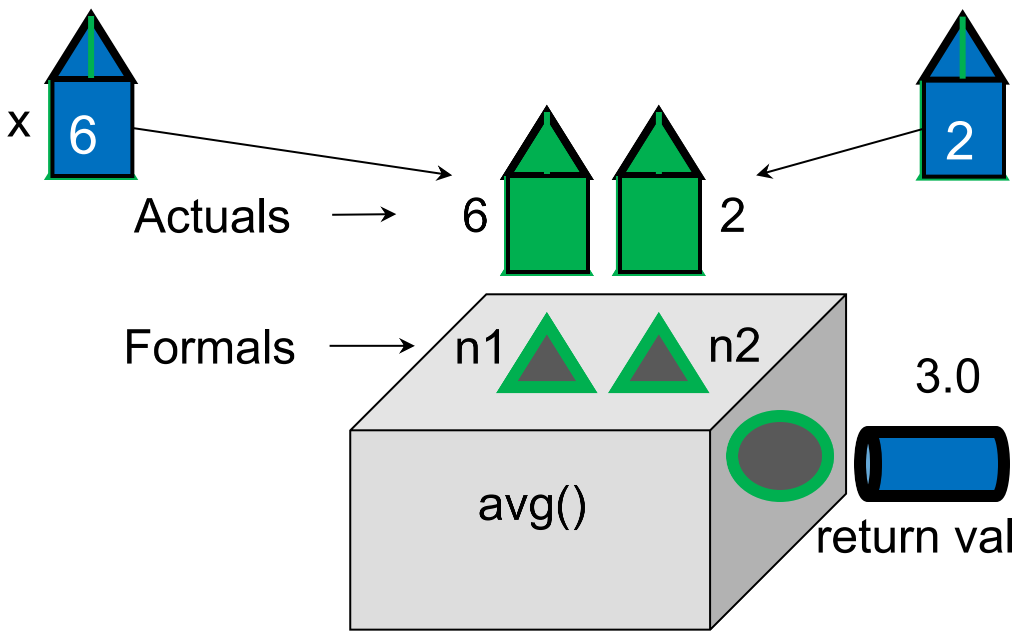
copy copy

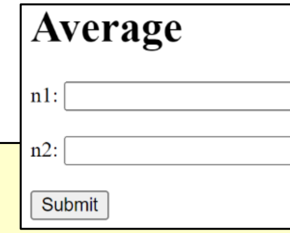


Each type is a "different" shape (int = triangle, double = square, char = circle). Only a value of that type can "fit" as a parameter..

Formals and Actuals (2)

- **Formal** parameters, n1 and n2
  - Placeholder names used inside the function
- **Actual** parameters
  - Actual values, 6 and 9 passed to n1 and n2, on the first call
  - Actual values, x and 2 passed to n1 and n2, on the second call
  - A **copy** is made and given to function






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

double avg(int n1, int n2)
{
  double sum = n1 + n2;
  return sum/2.0;
}

int main()
{
   int x=6, y = 9;  double z;
   z = avg(x,y);
   cout << "AVG is " << z << endl;
   z = avg(x, 2);
   cout << "AVG is " << z << endl;
   return 0;
}
```

copy copy

Each type is a "different" shape (int = triangle, char = square, double = circle). Only a value of that type can "fit" as a parameter.

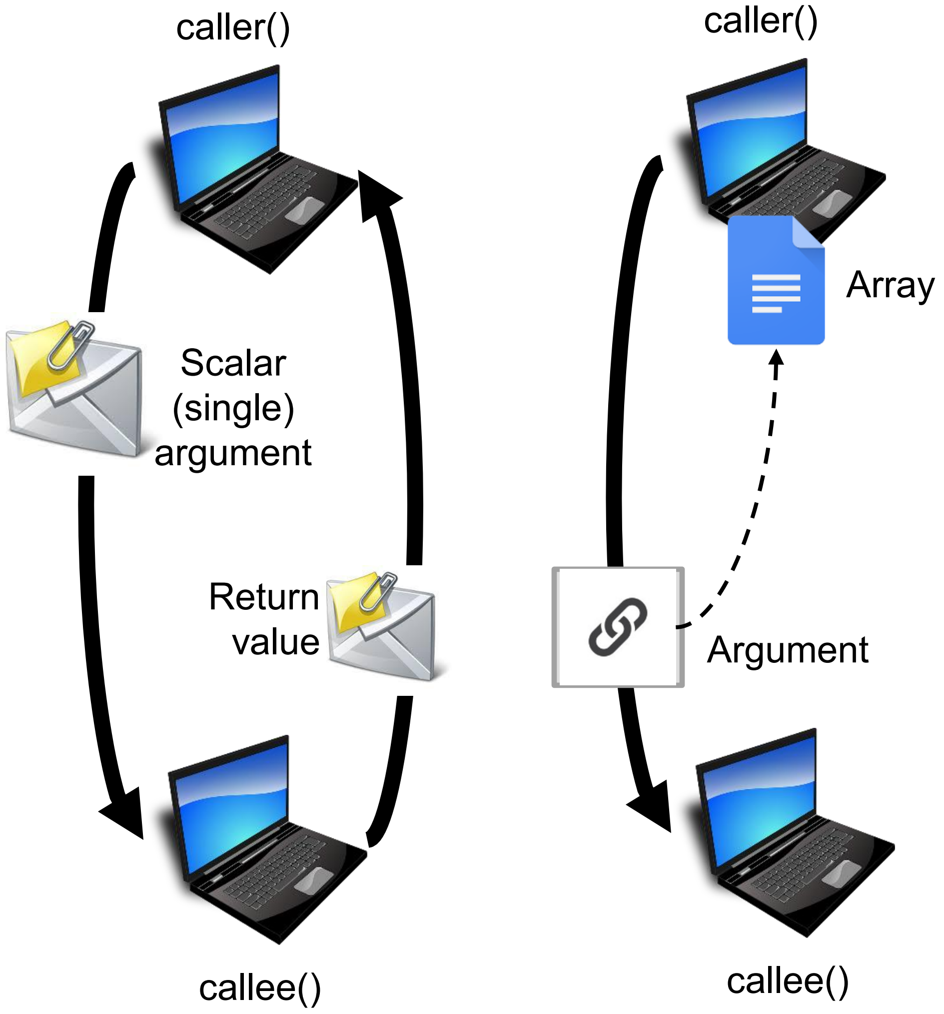
Pass-by-Value & Pass-by-Reference

- What are the pros and cons of emailing a LARGE document by:
  - Attaching it to the email
  - Sending a link (URL) to the document on some cloud service (etc. Google Docs)

- **Pass-by-value** is like emailing an attachment
  - A **copy** is made and sent
- **Pass-by-reference** means emailing a link to the original
  - **No copy is made** and **any modifications by the other party are seen by the originator**

Arrays and Pass-by-Reference

- Single (scalar) variables are **passed-by-value** in C/C++
  - Copies are passed
  - Like email attachments
- Arrays are **passed-by-reference**
  - Links (addresses) are passed
  - Like a link to a shared doc



Passing Arrays As Arguments

- Syntax:
  - **Step 1**: In the prototype and function definition:
    - Put empty square brackets **[ ]** after the **formal** parameter name if it is an array (e.g. `int data[]`) ..OR..
    - Put an ***** between the type and **formal** parameter name (e.g. `int* data`)
    - We'll prefer `int data[]` for now but `int* data` is **JUST AS VALID** and we'll learn more about it when we cover pointers)
  - **Step 2**: When you call the function, just provide the name of the array as the **actual** parameter

```
// Prototype
int init(int data[], int max_size);

int main()
{
  int vals[100];
  int len = init(vals, 100);
  // some code to process the input
  // in the vals array
  for(int i=0; i < len; i++) {
    cout << vals[i] << endl;
  }
  return 0;
}

int init(int data[], int max_size)
{
  int i=0, num;
  cin >> num;
  while( i < max_size && num != -1) {
    data[i] = num;
    i++;
    cin >> num;
  }
  return i;
}
```

Pass-by-Value / Reference

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

// Function prototypes
int initScalarInt();
void initArrayOfInts(int x[], int len);
void printVals(int x1, int x2[], int x2len);

int initScalarInt()
{
    return 42;
}

// Set all array elements to 42
void initArrayOfInts(int x[], int len)
{
    for(int i=0; i < len; i++){
        x[i] = 42;
    }
}
```

```
// Function definitions
int main()
{
    int x1;
    int x2[5];

    // Print initial values
    cout << "Before setting" << endl;
    printVals(x1, x2, 5);

    // Set values
    x1 = initScalarInt();
    initArrayOfInts(x2, 5);

    // Print values after they should have been set
    cout << "After setting" << endl;
    printVals(x1, x2, 5);

    return 0;
}

void printVals(int x1, int x2[], int x2len)
{
    cout << "X1: " << x1 << endl;
    cout << "X2: ";
    for(int i=0; i < x2len; i++) {
        cout << x2[i] << " ";
    }
    cout << endl;
}
```

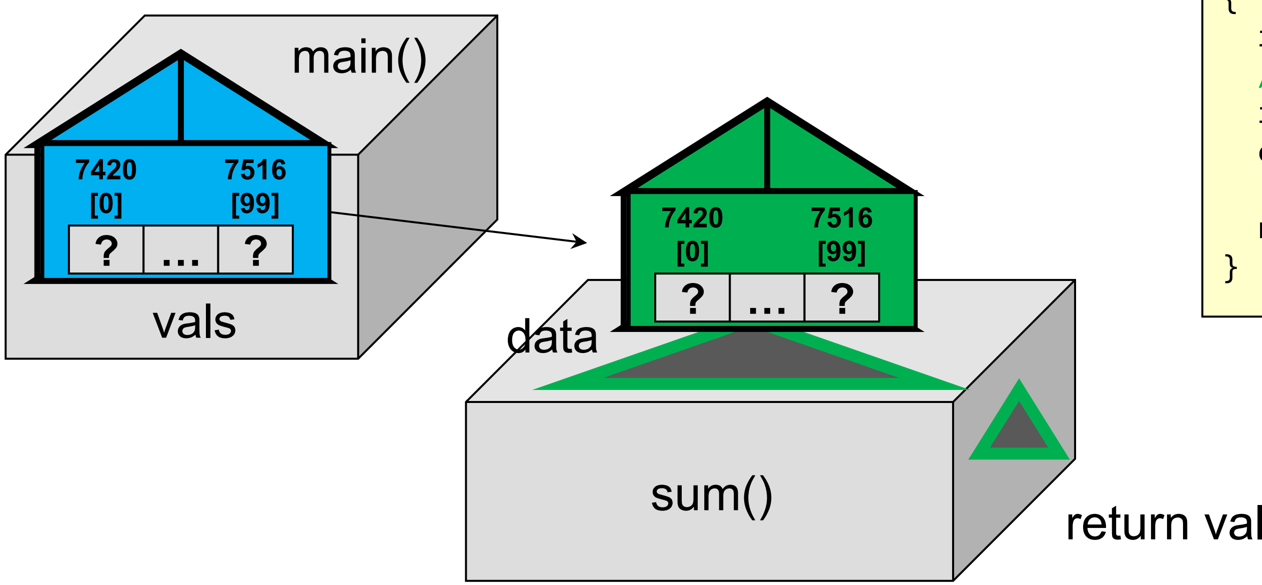
Passing arrays to other functions

ARRAYS AS ARGUMENTS AND ACCESSING ELEMENTS IN MEMORY

But Why Are Arrays Pass-by-Reference?

- If we used pass-by-value, then we'd have to make a copy of a potentially HUGE amount of data (what if the array had a million elements)
- To avoid copying vast amounts of data, we pass a link

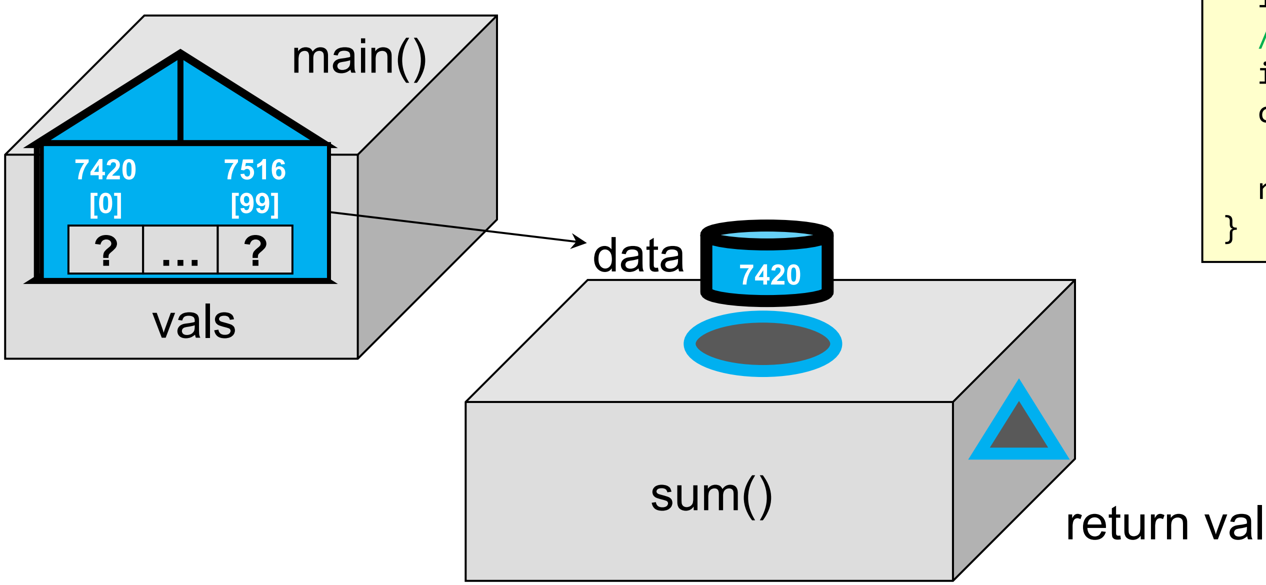
```
// Function that takes an array
int sum(int data[], int size);

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;
}
```



So What Is Actually Passed?

- The "link" that is passed is just the starting address of (pointer to) the array in memory (e.g. 7420).
- Once the function has the start address and the type, it will produce its own index values and be able to access the array in the caller's memory

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

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;
}
```



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)**

Arrays And Pass-by-Reference

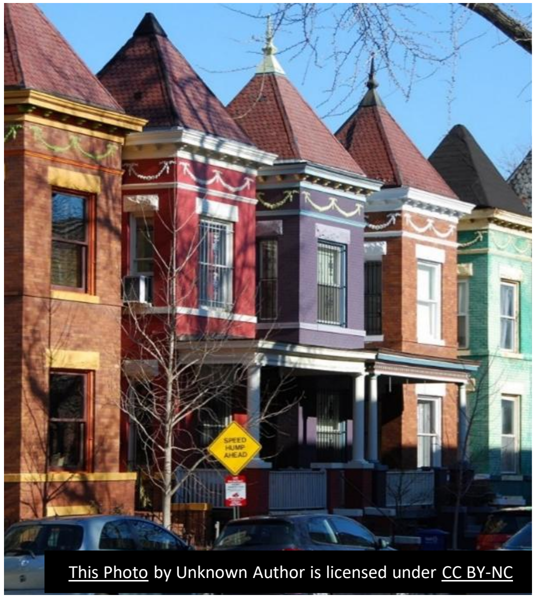
- Arrays are **passed-by-reference**
  - Links (addresses) are passed
    - **These links are actually memory addresses where the array starts.**
    - Using these addresses, any function can go to those locations and modify the data (array) from another function
  - Thus, changes to the array by a function are visible upon return to the caller
  - In this example, nums and vals refer to the **same** array

```
void init(int data[], int size);

int main()
{
  int vals[10];
  init(vals, 10);
  cout << vals[2] << endl;
     // prints -1
  return 0;
}

void init(int nums[], int size)
{  // nums is really a link to vals
   for(int i=0; i < size; i++){
     nums[i] = -1;
     // changing vals[i]
   }
}
```

Index:	[0]	[1]	[2]	...	[9]
vals:	-1	-1	-1	-1	-1

Strange Question

- The first house on the on the block of a street has address 7420.
- How many houses are on the block?



- Look at the memory to the right. An array starts at address 7420. How many elements are in that array?
- Having the start address doesn't allow us to know how big the array is.

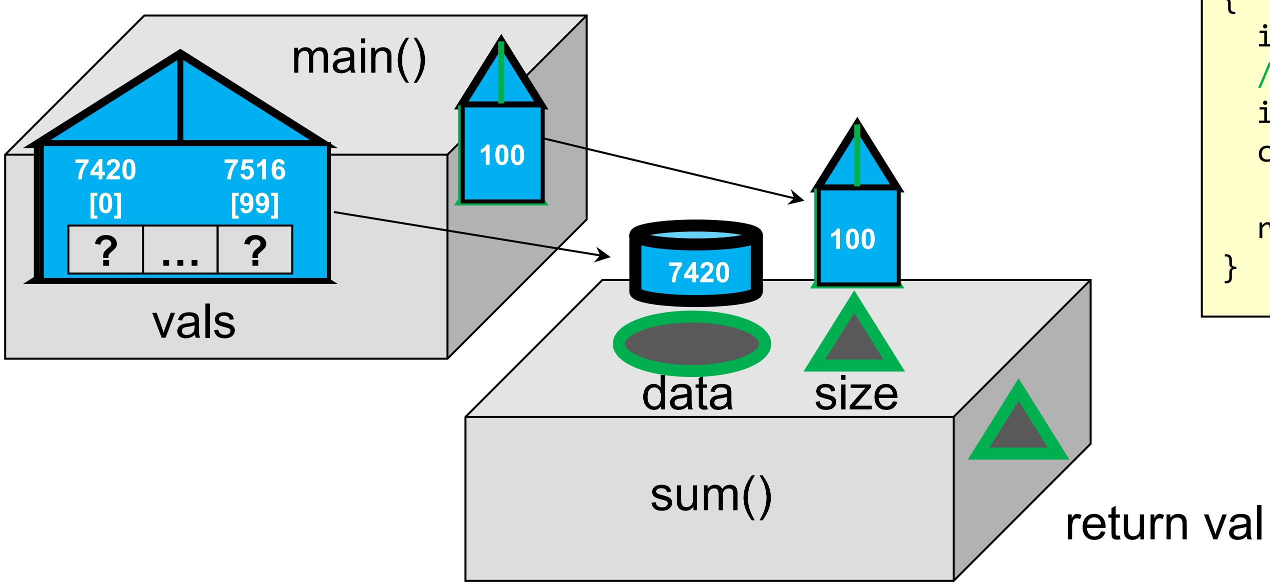
	Address	Memory Data	
	7412	a184beef	07818821
array	7420	5621930c	e400cc33
	7428	a184beef	07818821
	7436	5621930c	e400cc33
	...	...	

- We must also track / pass the **size**!

Arrays in C/C++ vs. Other Languages

- Notice that if sum() only has the start address it **would not know** how big the array is
- **Unlike Java** or other languages where you can call some function or access some property to give the size of an array, **C/C++ require you to track the size yourself in a separate variable and pass it as a secondary argument**

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

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;
}
```



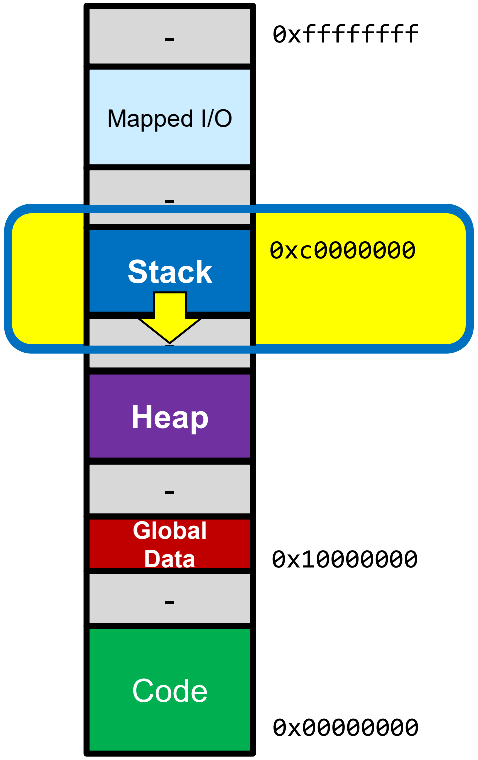
Understanding how functions utilize the **stack** area of computer memory

PASSING ARGUMENTS: A DEEPER LOOK

Memory Organization

- 32-bit address range (`0x0 to 0xffffffff`)
  - Note **`0x`** indicates a hexadecimal number
- **Code** usually sits at lower addresses
- **Global** variables/data somewhere after code
- **Heap**: Area of memory that can be allocated and de-allocated during program execution (i.e. dynamically at run-time) based on the needs of the program
  - More in a few lectures
- **Stack (our focus):** Memory for all information related to each running instance of a **function**
  - Arguments to the function
  - Local variables
  - Return link (where in the code to return)

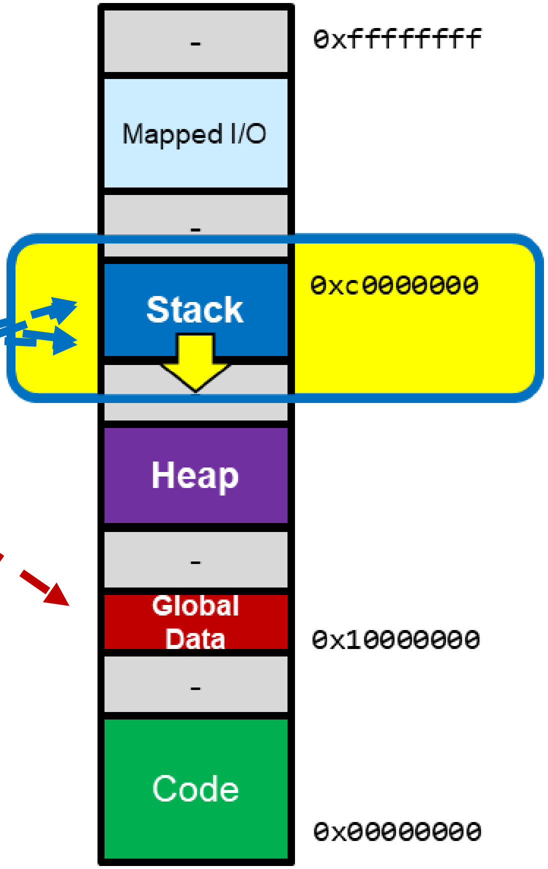
Memory (RAM) Layout of Program



Mapping of Info to Memory

```
#include <iostream>
#include <algorithm>
#include <cmath>
using namespace std;
int timesCalled = 0;  // global variable

int factorial(int n)
{
    int f = 1;
    for(int i = 1; i <= n; i++) {
        f *= i;
    }
    timesCalled++;
    return f;
}
int main() {
  int n;
  cin >> n;
  int res = factorial(n);
  cout << res << " " << timesCalled << endl;
  return 0;
}
```

Memory (RAM)
Layout of Program

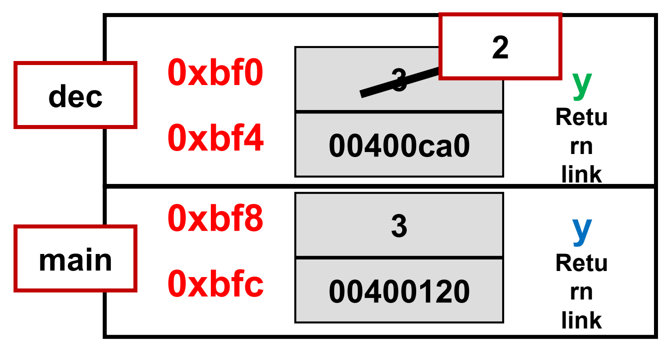


Understanding the Stack and Pass-by-Value

- Each program allocates an area of memory known as 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
- Each time a function is called, the computer **allocates** memory for that function on the top of the stack and creates a link for where to return
- When a function returns/ends, that memory is **deallocated** (destroying all arguments and local variables) and control is returned to the function now on top

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

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

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

Stack Area of RAM

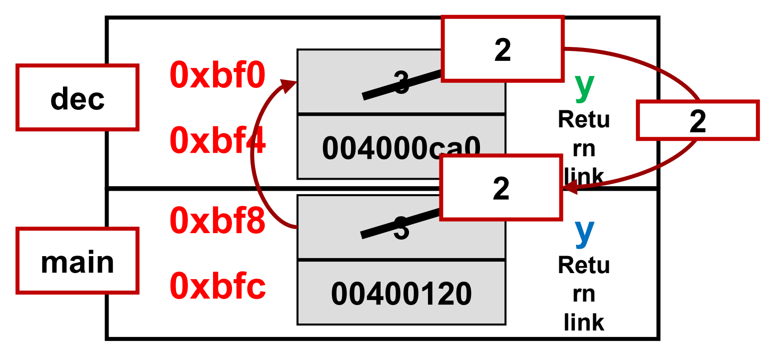


Understanding the Stack and Pass-by-Value

- Each program allocates an area of memory known as 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
- Each time a function is called, the computer **allocates** memory for that function on the top of the stack and creates a link for where to return
- When a function returns/ends that memory is **deallocated** (destroying all arguments and local variables) and control is returned to the function now on top

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

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

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

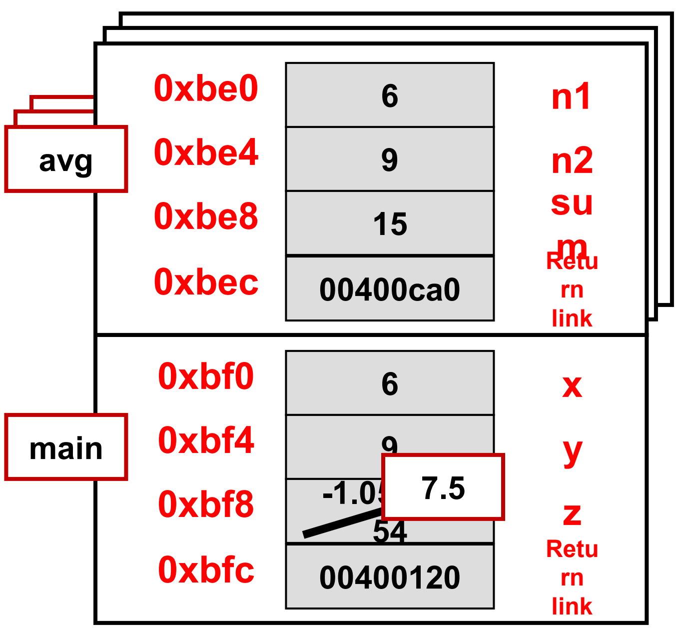
Stack Area of RAM



Another Example

- Each program allocates an area of memory known as 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

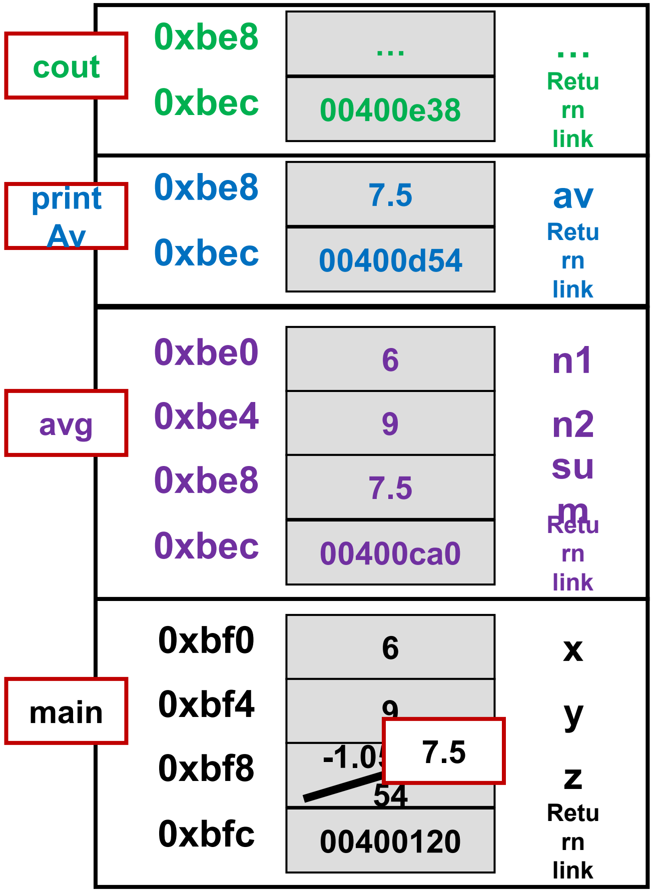
Stack Area of RAM




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

double avg(int n1, int n2); // Prototype

int main()
{
   int x=6, y = 9;  double z;

   z = avg(x,y);
   cout << "AVG is " << z << endl;

   z = avg(x, 2);
   cout << "AVG is " << z << endl;
   return 0;
}

double avg(int n1, int n2)
{
  double sum = n1 + n2;
  return sum/2.0;
}
```

Scope and Stack Example

- The **scope** of local variables and arguments are only for the lifetime of the function in which they live
- One function cannot access the local variables of another

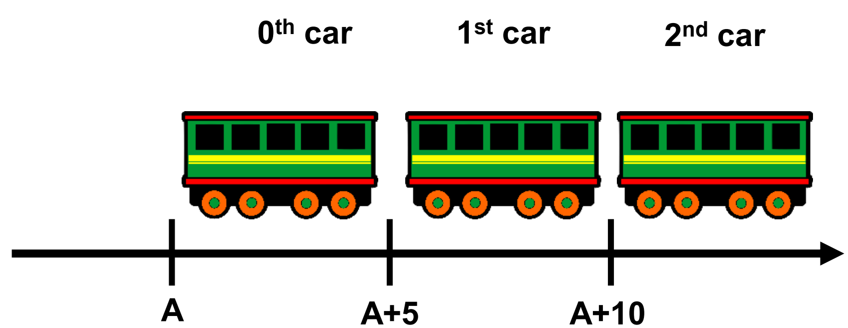



```
#include <iostream>
using namespace std;

double avg(int n1, int n2); // Prototype
void printAv(double x);     // Prototype

int main()
{
   int x=6, y = 9;  double z;

   z = avg(x,y);

   z = avg(x, 2);

   return 0;
}

double avg(int n1, int n2)
{
  double sum = (n1 + n2)/2.0;
  printAv(sum);
  return sum;
}

void printAv(double av)
{
  cout << "Average is " << av << endl;
}
```

A Quick Tangent: Array Element Addresses

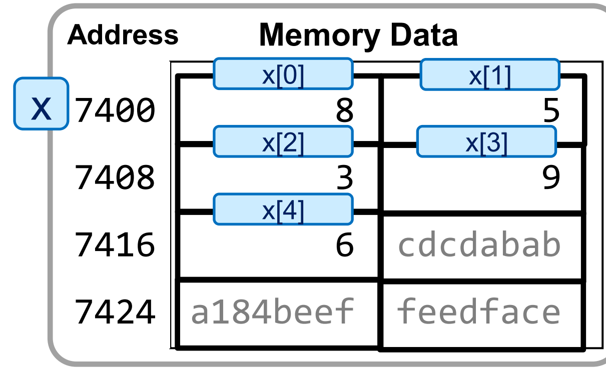
- Consider a train with many copies of the same car
  - The "0^{th}" car starts at **point A** on the number line
  - Each car is **5 meters** long
- Write an arithmetic expression for where the **i-th car** is located. (At what meter on the number line does it start?)
- Suppose an array of integers starts at memory address A, write an expression for where the **i-th integer** starts?
- Suppose an array of doubles starts at memory address A, write an expression for where the **i-th double** starts?

Formula for address of i-th element:



Formula for Addressing Array Elements

- Assume a 5-element int array
  - `int x[5] = {8,5,3,9,6};`
- **Fun Fact 3 (after Unit 0's Fact 1 & 2):** Using the **name of an array by itself (e.g. x)** w/o square brackets, evaluates to the **starting address** in memory of the array (i.e. address of 0th entry).
- When you access x[2], the CPU uses x (to know the starting address) and adds the product of the index, 2, times the size of the data type (i.e. int = 4 bytes)
  - x[2] => int. @ address 7400 + 2*4 = 7408
  - x[3] => int. @ address 7400 + 3*4 = 7412
  - x[i] @ start address of array + i * (size of int)
- **Recall**: C/C++ does NOT perform bounds checking to stop you from attempting to access an element beyond the end of the array
  - x[6] => int. @ address 7400 + 6*4 = 7424 (Garbage!!)



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)**

Formula: `start_addr + i*data_size`

Fun Fact 3: If you use the **name of an array (e.g. x)** w/o square brackets it will evaluate to the **starting address** in memory of the array (i.e. address of 0th entry)

Fun Fact 3b: Fun Fact 3 usually appears as one of the first few questions on the midterm.

Array Elements vs. Array Names

- **In C/C++ using an array name without any index evaluates to the starting address of the array**
- **Example:**
  - `vals[0]` yields data
  - `vals` yields an address

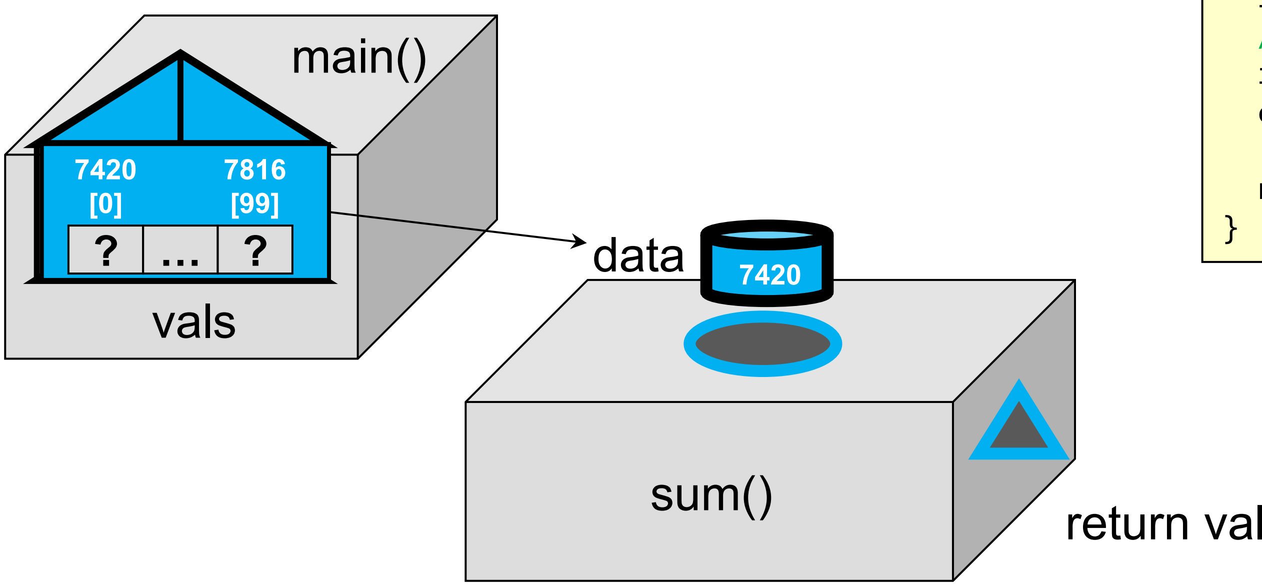
Index:	[0]	[1]	[2]	[3]	[4]
vals @ (0x7420)	7	4	9	2	3

```
int main()
{
  int vals[5] = {7,4,9,2,3};

  cout << vals[0] << endl;
     // prints 7
  cout << vals << endl;
     // prints ___________
 return 0;
}
```

Recall: Passing Arrays

- The "link" that is passed is just the starting address of (pointer to) the array in memory (e.g. 7420).
- Once the function has the start address and the type, it will produce its own index values and be able to access the array in the caller's memory

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

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;
}
```

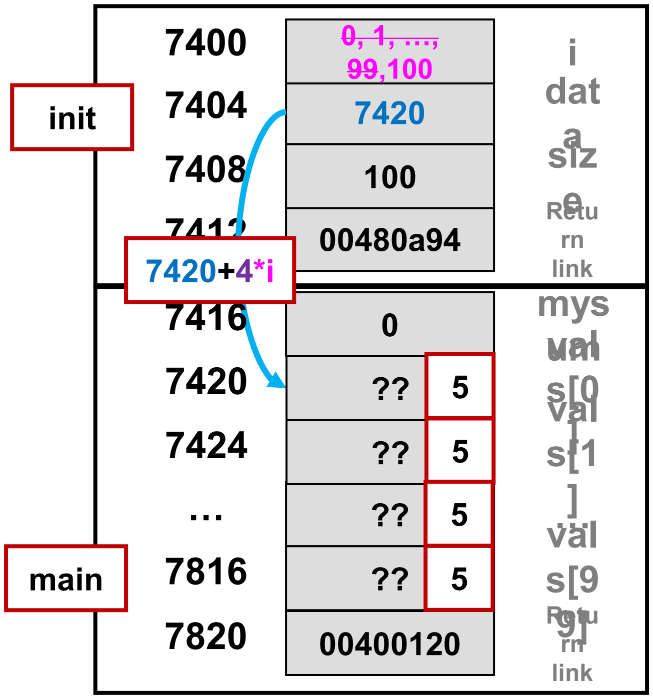


To access an element in an array, we need 3 pieces of info:

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

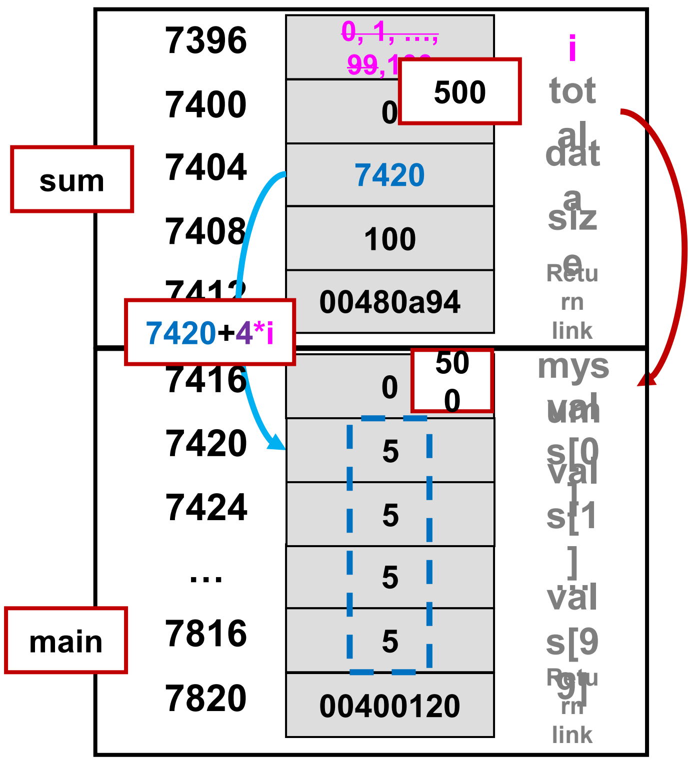
- The function receives the starting address of the array which it can use along with the type (e.g. `int`) and index to access the appropriate values from main's stack area of memory.




```
void init(int data[], int size);
int sum(int data[], int size);
int main()
{
  int vals[100], mysum = 0;

  init(vals, 100);
  mysum = sum(vals, 100);
  cout << mysum << endl;

  return 0;
}

void init(int data[], int size)
{
  for(int i=0; i < size; i++){
     data[i] = 5;
  }
}
```

Stack View of Passing Arrays

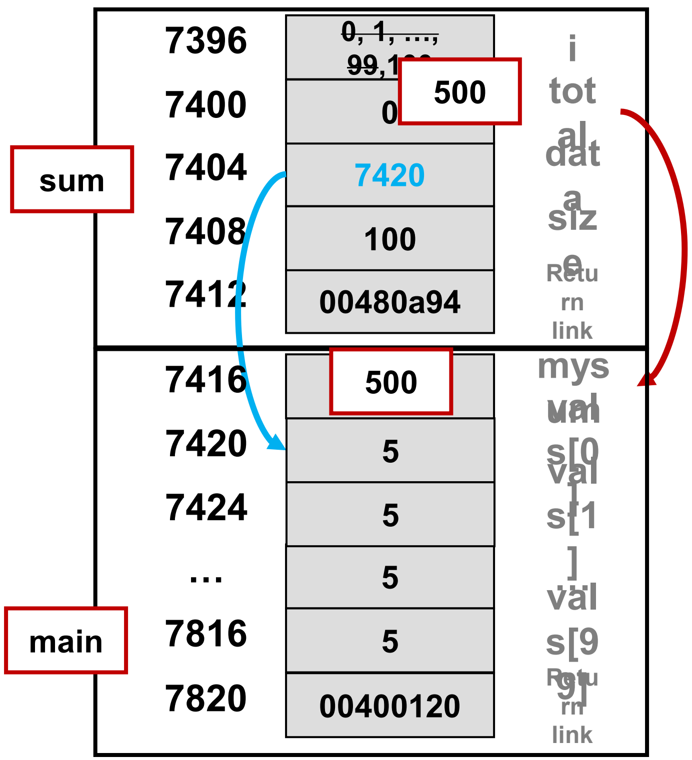
- The function receives the starting address of the array which it can use along with the type (e.g. `int`) and index to access the appropriate values from main's stack area of memory.




```
void init(int data[], int size);
int sum(int data[], int size);

int main()
{
  int vals[100], mysum = 0;

  init(vals, 100);
  mysum = sum(vals, 100);
  cout << mysum << endl;

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

Why Empty Brackets

- Why don't we just supply the array size in the formal argument?
  - Now we can only process arrays of size 100. We'd like our functions to be more general and handle any size array
  - C/C++ doesn't do bounds checking anyway, so what good would writing 100 be?

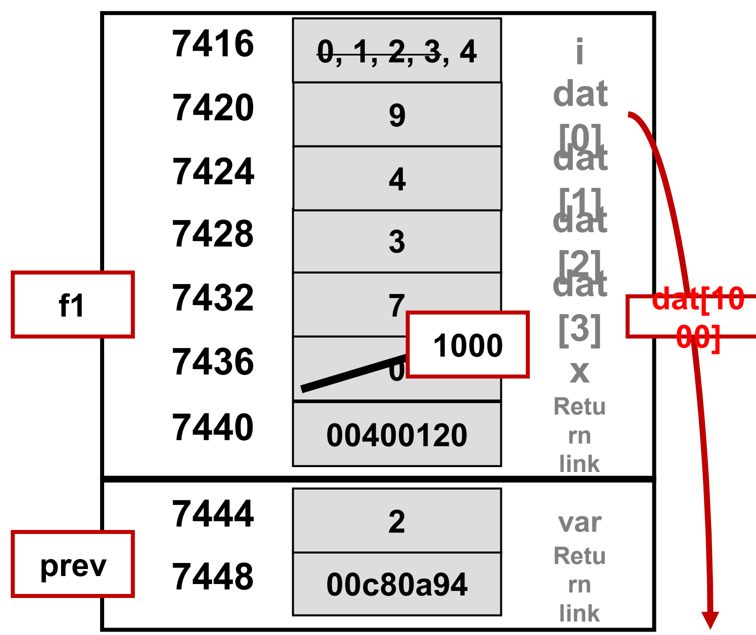



```
int sum(int data[100], int size);

int sum(int data[100], 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;
}
```

(Lack of) Array Bounds Checking

- C++ does **NOT** bounds check the index used to access an element
  - It will simply treat all of memory as part of the array (i.e. larger positive indices go past the end of the array while negative offsets are before the array start)
  - Thus, allowing you to read and write data you shouldn't (including variables from your own function or another function since local variables live on the stack)
  - This issue is a common exploit by hackers (more in future courses like CS 356)

```
int f1()
{
  int dat[4], x=0;

  for(int i=0; i <= 4; i++){
     cin >> dat[i];  // 9, 4, 3, 7, 1000
  }
  // using i=4 overwrites 'x'
  cout << x << endl;
     // 1000 would print, not 0.
  cout << dat[x] << endl;
     // likely segmentation fault
  return 0;
}
```



Array Summary

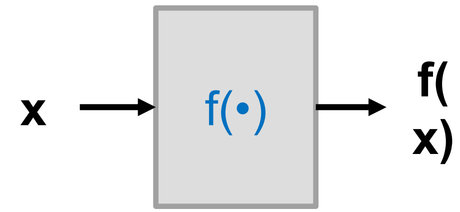
- Arrays must be declared with a **FIXED** size (cannot use a variable for its length)
  - GOOD: `int data[50];`
  - BAD: `int data[n];`
- After declaring an array, C/C++ only "remembers" the **starting address** of the array and the **type** of data it holds (to know the **data size**)
  - Which is all it needs to know to access any element using the formula: `start_addr + i*data_size`
- C/C++ do NO bounds checking
  - Will simply apply the formula above to **WHATEVER** index you provide or calculate
  - Most common source of a program crash (and also security vulnerabilities). If your program crashes in CS 103, suspect a bad array access

Using arrays as a lookup table

LOOKUP TABLES

Motivation and Approaches

- Problem Statement: Given an input, x, convert it to an output using some function, f(x)
- Possible approaches
  - Use an arithmetic relationship, when the relationship can be easily generalized
  - Break it into cases with if statements when there are a reasonable number of cases
- What if there is little pattern or many cases?
- Consider use of an **array** as a **"look-up table"**



x:	0	1	2	3	4
f(x):	1	3	5	7	9

f(x) = ______

x:	0	1	2	3	4	5
f(x):	1	1	1	2	2	2

f(x) = ___, if ________
f(x) = ___, otherwise

x:	0	1	2	3	4	5
f(x):	4	1	0	2	5	3

f(x) = ______

Arrays as Look-Up Tables

- **Look-up Table Idea**: Store pre-computed results in an array and then "look-up" the desired result using the input as the array index
- Can extend this to process many inputs (an array of inputs)
  - Suppose an instructor with 8 students gives a quiz worth 10 points and we use the customary (>90% = A, 80% = B, 70% = C, 60% = D, <60% = F) and we want to map the points to the letter grade. How would you do it?

x:	0	1	2	3	4	5
f(x):	4	1	0	2	5	3

```
int main()
{
  int myf[] = {4, 1, 0, 2, 5, 3};
  int x;
  cin >> x;
  cout << myf[x] << endl;
  return 0;
}
```

Problem from Previous Slide

```
int main()  {
  int scores[8] = {9,7,10,9,8,4,6,8};
  char grades[11] =

 {'F','F','F','F','F','F','D','C','B','A','A'};
  for(int i=0; i < 8; i++){
    // output the letter grade for each score
    cout << "Score: " << scores[i] << " => "

         << "Grade: " << ____________________
         << endl;
  }
  return 0;
}
```

Grade Mapping Problem

C-STRINGS, COUT, AND CIN

Character Arrays and Strings (1)

- Recall that in C/C++ string constants (the text in between " ") are just character arrays
  – Each character consumes 1 element in the array
  – Ends with the null character (e.g. 0 decimal or '\0' ASCII)
- This approach of using an array of char's to store a string is referred to as a **C-String** because there was no string type in C (i.e. before C++)

```
#include <string>
using namespace std;
int main()
{
  char str1[3] = {'C', 'S', '\0'};
  // For char arrays easier to use ""
  char str2[7] = "CS 103"
  /* Initializes the array to "CS 103"*/

  cout << str1 << endl;    // prints "CS"
  cout << str2 << endl;    // prints "CS 103"

  str2[5] = '4';
  cout << str2 << endl;    // prints "CS 104"

  cin >> str2; // get a new string from
               // the user (suppose user
               // types "hello"
  cout << str2;
}
```

Addr: Index:	520 [0]	521 [1]	522 [2]	523 [3]	524 [4]	525 [5]	526 [6]
str2:	'C'	'S'	' '	'1'	'0'	'3'	'\0'

Computer Memory

Program Output:

```
CS
CS 103
CS 104
hello
```

Character Arrays and Loops

- How many things can a computer do at a time?
- To printout a string/character array, we'd have to print one character at a time!
- But C/C++ treats character arrays specially. cout has a loop inside its code to print strings/character arrays.
- Though not shown, cin also has a loop inside to input a string.
- We say cout and cin have a **special relationship with character arrays.**

Addr: Index:	520 [0]	521 [1]	522 [2]	523 [3]	524 [4]	525 [5]	526 [6]
str1:	'C'	'S'	' '	'1'	'0'	'3'	'\0'

Computer Memory

```
#include <string>
using namespace std;
int main()
{
  char str1[7] = "CS 103"
  /* Initializes the array to "CS 102"*/

  // Usually in C/C++ we must use a loop to do
  // many operations
  for(int i=0; str[i] != '\0'; i++) {
    cout << str[i];
  }
  cout << endl;

  // but cout has its own loop so you don't
  // have to write the loop above but just
  // what you see below.
  cout << str1 << endl;    // prints "CS 102"
}
```

Program Output:

```
CS 103
CS 103
```

cout's Special Relationship with Character Arrays

- To print out all elements of **any array type *OTHER than a character array*** (i.e. int, double, bool, etc.) you must **write your OWN loop** (i.e. because computers can only do 1 thing at a time)
- But for **character arrays**, you can just give cout the name of the array and it will use its **own INTERNAL loop** to print out all characters for you
  - So, internally it is actually looping over the characters so you don't have to
  - It just assumes when you give it a character array that you WANT it to print out all the characters in the array
- Thus, we say **cout** treats **character** arrays specially

Index:	[0]	[1]	[2]	[3]	[4]
data:	9	7	9	9	5

Index:	[0]	[1]	...	[9]	[10]
str1:	'M'	'a'	...	s	\0

```
int main()
{
  int data[5] = {9, 7, 8, 9, 5};
  char str1[] = "Many chars";
  // right way to print int array contents
  for(int i=0; i < 5; i++){
    cout << data[i] << " ";
  }
  cout << endl;

  // doesn't work for an int, double
  // or any other type of array
  cout << data << endl;

  // cout treats char. arrays specially
  cout << str1 << endl;
}
```

Program Output:

```
9 7 8 9 5
0x7fffce40
Many chars
```

cin's Special Relationship with Character Arrays

- To get input for all elements of an array type *OTHER than character arrays* (i.e. `int`, `double`, etc.) you must **write your OWN loop**
- But for character arrays, you can just give `cin` the name of the array and it will use its **own INTERNAL loop** to receive all characters the user types and store them sequentially in the array
  - So, internally it is actually looping over the characters so you don't have to
  - It just assumes when you give it a character array that you WANT it to get a full string (stopping at the next space)
- **`cin`** treats character arrays specially

```
int main()
{
  int data[5]; //5 garbage values to start
  char str1[8];//8 garbage values to start
  int sum = 0;
  // doesn't work for an int, double
  // or any other type of array
  cin >> data;  // won't even compile

  // right way to get int array contents
  for(int i=0; i < 5; i++){
    cin >> data[i];
  }

  // cin treats char. arrays specially
  cin >> str1;
}
```

	520 [0]	521 [1]	522 [2]	523 [3]	524 [4]	525 [5]	526 [6]	527 [7]	528 sum
str1:	?	?	?	?	?	?	?	?	0

user types: CS103

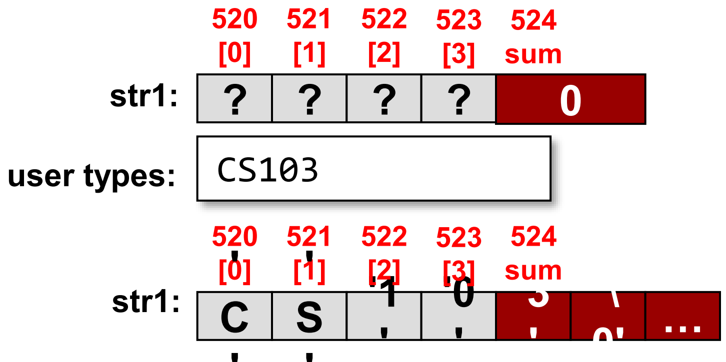
	520 [0]	521 [1]	522 [2]	523 [3]	524 [4]	525 [5]	526 [6]	527 [7]	528 sum
str1:	C	S	1	0	3	\0	?	?	0

A Problem with `cin` and Character Arrays

- What if the user types in **TOO** much (*more characters than our array has room to store*)?
- `cin` will not stop! It will keep storing the characters the user types, overwriting whatever data and variables came after the array
- **Warning**: `cin` does not CHECK that the string typed by the user will fit in the array; instead it simply **overwrites memory leading to undefined (bad) behavior!**

- **C++ strings fix this issue, allocating more space based on what is typed.**

```
int main()
{
  char str1[4];
  int sum = 0;
  // What if user types in "CS102"
  cin >> str1;

  cout << sum << endl;
  // won't see 0 because sum was modified
  // when cin received the string that was
  // too long!

  string s2;
  cin >> s2;
  // works regardless of user input length
}
```



Exercises

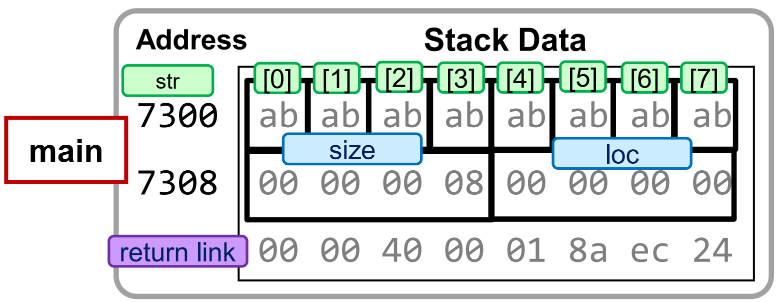
- **Cipher**: Using an array as a Look-Up Table
  - Let’s create a cipher code to encrypt text
  - `abcdefghijklmnopqrstuvwxyz` => `ghijklmaefnzyqbcdrstuopvwx`
  - char orig_string[] = “helloworld”;
  - char new_string[11];
  - After encryption:
    - new_string = "akzzbpbrzj"
  - Define another array
    - char cipher[27] = "ghijklmaefnzyqbcdrstuopvwx";
    - How could we use the original character to index (“look-up” a value in) the cipher array

Input Buffer Overflow [Only if Time]

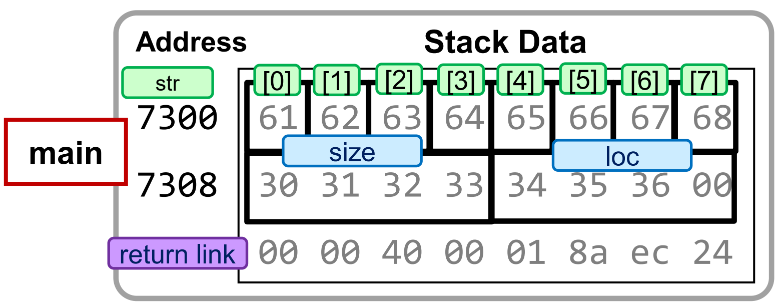
- Depending on user input, this program will likely crash.

```
$ ./prog1
abcdefgh0123456
```

Stack before execution of `cin >> str`**.**



Stack after execution of `cin >> str`**.**



Find location of first capital letter in text

```
#include <iostream>
#include <cctype>
using namespace std;
int main()
{
  char str[8];
  const int size = 8
  int loc = 0;

  // User types "abcdefgh0123456"
  cin >> str;

  // size may now be garbage (not 8)
  for( int i=0; i < size; i++){
    if( isupper(str[i]) )
      { loc = i; break; }
  }
  // You'll be lucky to even get here
  cout << loc << endl;
  return 0;
}
```

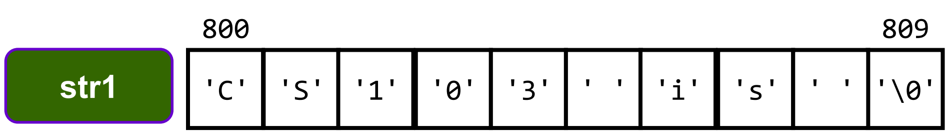
NULL Terminated character arrays

C-STRINGS (CHARACTER ARRAYS)

C-Strings

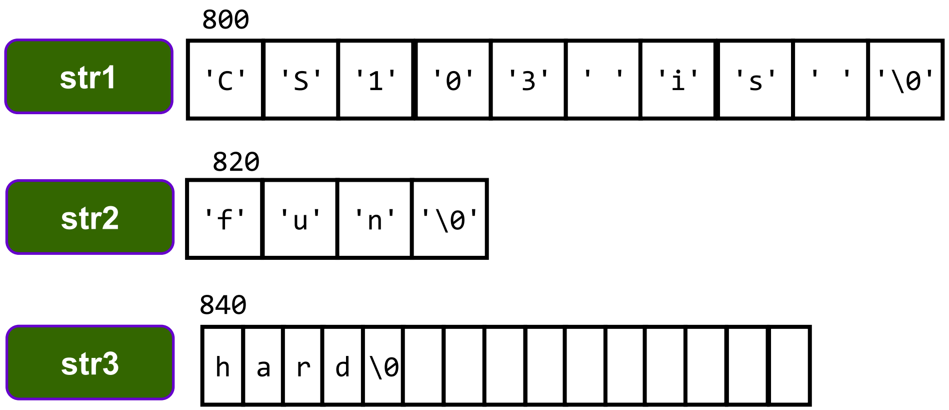
- In C:
  - strings were not a first-class type (i.e. no `string` type)
  - strings were simply character arrays (`char []`) **terminated by the null character** (0 dec. = '\0' ASCII)
  - These were known as C-Strings
- No operations/operators other than typical array operations are provided
  - **No** comparison (== or !=)
  - **No** assignment/copy (=)
  - **No** append/concatenate (+)

```
int main()
{
  char str1[] = "CS103 is ";
  char str2[] = "fun";
  char str3[15];

  cin >> str3;   // user enters "CS103"

  // What is this actually comparing?
  if(str3 == str2)
    { cout << "Match" << endl; }

  str3 = str1;
  str3 += str2;

  cout << str3 << endl;
  return 0;
}
```



Errors

```
int main()
{
  char str1[] = "CS103 is ";
  char str2[] = "fun";
  char str3[15];

  cin >> str3;  // user enters "CS103"

  // What is this actually comparing?
  if(str3 == str2)
    { cout << "Match" << endl; }

  // Intuitively this makes sense but
  // will not compile in C/C++. Using your
  // knowledge of types and other info,
  // what is this actually attempting to do.
  str3 = str1;
  str3 += str2;

  cout << str3 << endl;
  return 0;
}
```



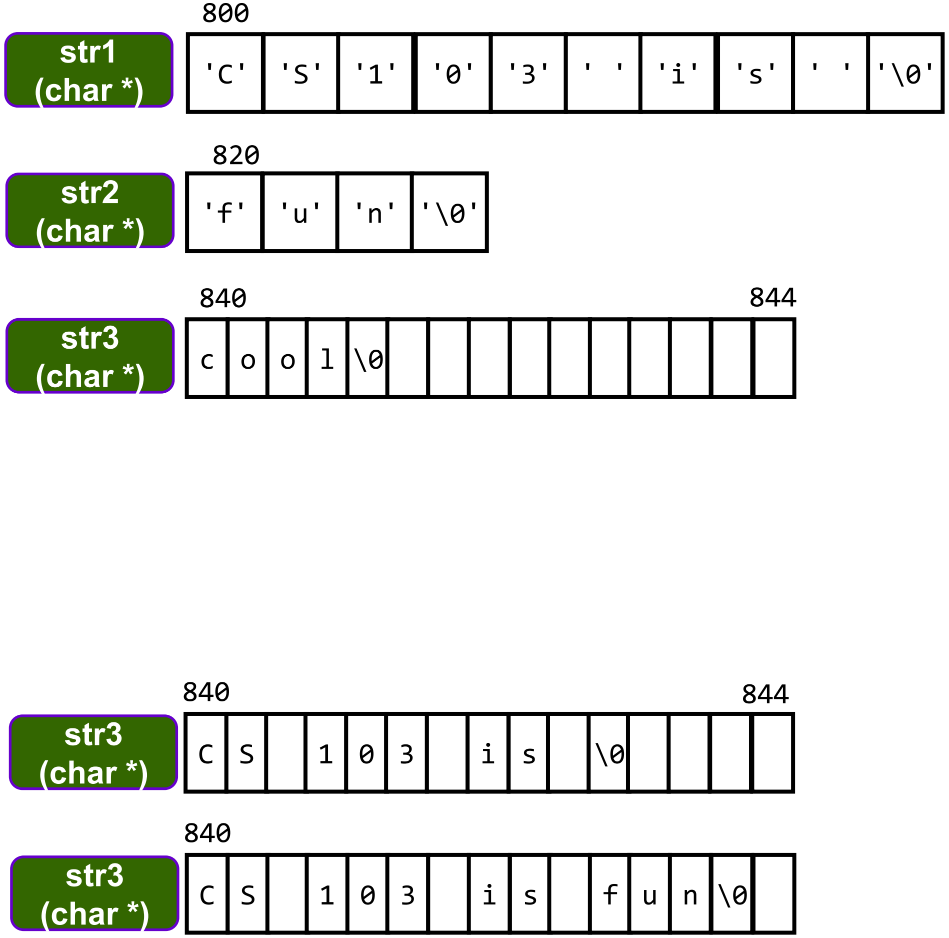

```
main.cpp:15:13: warning: comparison between two arrays is deprecated;
if(str3 == str2) { cout << "Match" << endl; }
~~~~ ^ ~~~~ main.cpp:15:13: warning: array comparison always evaluates to false

main.cpp:20:10: error: array type 'char[15]' is not assignable
str3 = str2;
~~~~ ^
main.cpp:21:10: error: invalid operands to binary expression ('char[15]' and
'char[7]')
str3 += str2;
~~~~ ^ ~~~~
```

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(char dest[]);`
    - Returns the length of the string (not counting the null character)
  - `int strcmp(char str1[], char str2[]);`
    - Return 0 if equal, >0 if str1 is alphanumerically larger than str2, <0 if str1 is less than str2
  - `char* strcpy(char dest[], char src[]);`
    - Copies the whole C-string from src to the dest array (overwriting what's in dest)
    - Ignore the return type for now (think of it as a void function)
  - `char* strcat(char dest[], char src[]);`
    - Concatenates src to the end of dest
    - Ignore the return type for now (think of it as a void function)

Use of the C-String Library

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

int main()
{
  char str1[] = "CS103 is ";
  char str2[] = "fun";
  char str3[15];

  cin >> str3;  // user enters "cool"

  // What is this actually comparing?
  if(str3 == str1)
  if(0 == strcmp(str1,str3))
    { cout << "Match" << endl; }

  // Intuitively this makes sense but
  // will not compile in C/C++. Using your
  // knowledge of types and other info,
  // what is this actually attempting to do.
  str3 = str1;
  strcpy(str3, str1);

  str3 += str2;
  strcat(str3, str2);

  cout << str3 << endl;
  return 0;
}
```



Sample Implementations

- Exercises
  - strlen
  - strcpy

(Self study and ask questions)

SOLUTIONS AND MORE FUNCTION EXAMPLES

Pass by Value Solution

- Wait! But they have the same name, 'y'
  - What's in a name…Each function is a separate entity and so two 'y' variables exist (one in main and one in decrement it)
  - The only way to communicate back to main is via return
  - Try to change the code appropriately
- **Main Point:** Each function is a completely separate "sandbox" (i.e. is isolated from other functions and their data) and copies of data are passed and returned between them

```
void dec(int);
int main()
{
  int y = 3;
  dec(y);
  cout << y << endl;
  return 0;
}
void dec(int y)
{
   y--;
}
```

```
int dec(int);
int main()
{
  int y = 3;
  y = dec(y);
  cout << y << endl;
  return 0;
}
int dec(int y)
{
   y--;
   return y;
}
```

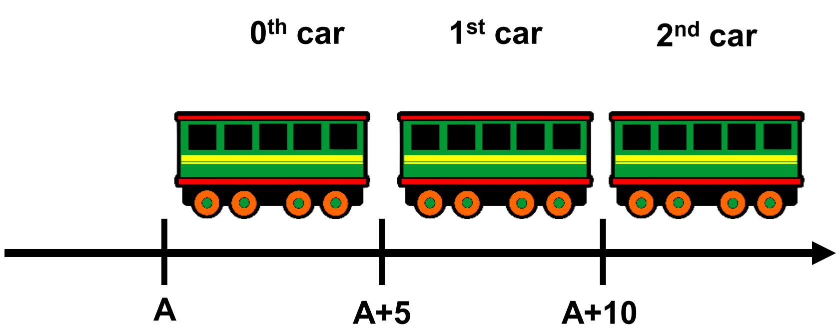
Exercise Solution

- Consider a train with many copies of the same car
  - The "0th" car starts at point A on the number line
  - Each car is 5 meters long
- Write an expression of where the i-th car is located (at what meter does it start?)
- Suppose a set of integers start at memory address A, write an expression for where the i-th integer starts?
- Suppose a set of doubles start at memory address A, write an expression for where the i-th double starts?

A + 5*i

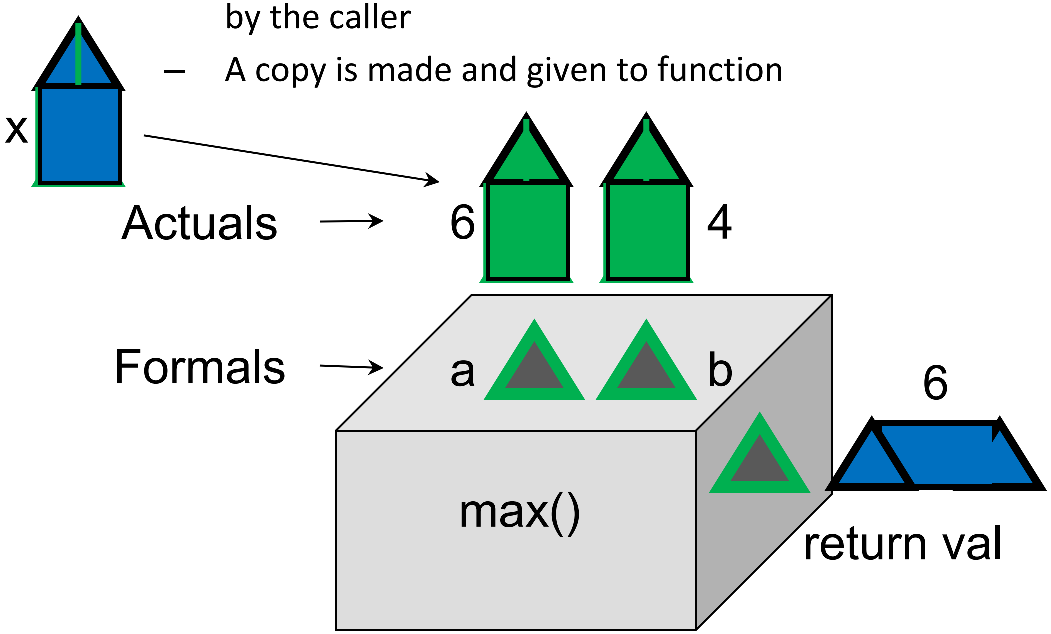
A + 4*i

A + 8*i



Parameter Passing (1)

- **Formal** parameters, a and b
  - Type of data the parameter expects
  - Names that will be used internal to the function to refer to the values passed (e.g. generic placeholders/titles used in contracts like "CEO" or "professor" that will be assigned or replaced real value)
- **Actual** parameters
  - Actual values ("Jeff Bezos", "Mark") input to the function by the caller
  - A copy is made and given to function




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

int max(int a, int b)
{
  if(a > b)
    return a;
  else
    return b;
}

int main()
{
   int x=6, z;
   z = max(x,4);
   cout << "AVG is " << z << endl;
   z = avg(x, 2);
   cout << "AVG is " << z << endl
   return 0;
}
```

Formals

Actuals

Actuals

Each type is a "different" shape (int = triangle, double = square, char = circle). Only a value of that type can "fit" as a parameter.

Scope Example

- Globals live as long as the program is running
- Variables declared in a block { … } live as long as the block has not completed
  - { … } of a function
  - { … } of a loop, if statement, etc.
- When variables share the same name, the closest declaration will be used by default

```
#include <iostream>
using namespace std;
int x = 5;
int main()
{
  int a, x = 8, y = 3;
  cout << "x = " << x << endl;
  for(int i=0; i < 10; i++){
    int j = 1;
    j = 2*i + 1;
    a += j;
  }
  a = doit(y);
  cout << "a=" << a ;
  cout << "y=" << y << endl;
  cout << "glob. X" << ::x << endl;
}

int doit(int x)
{
   x--;
   return x;
}
```