

## Introduction to C++: Part 2

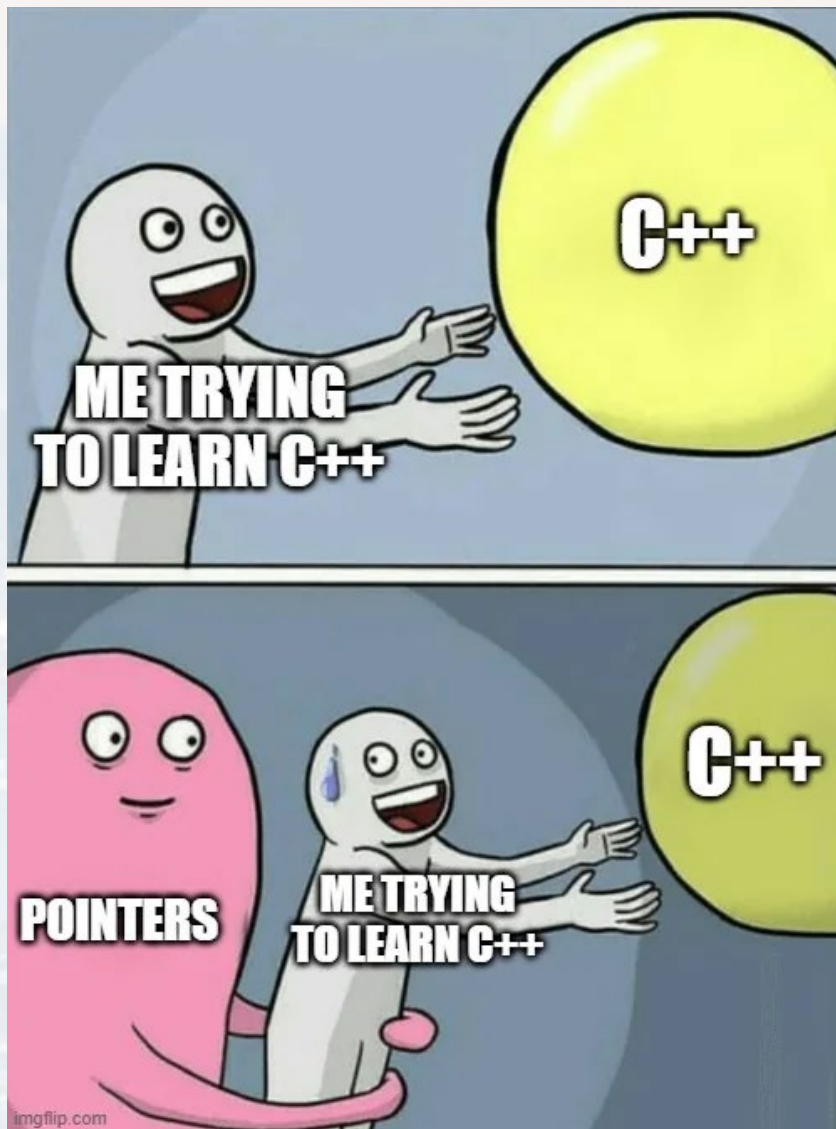


Jessica Nash

University California, Berkeley

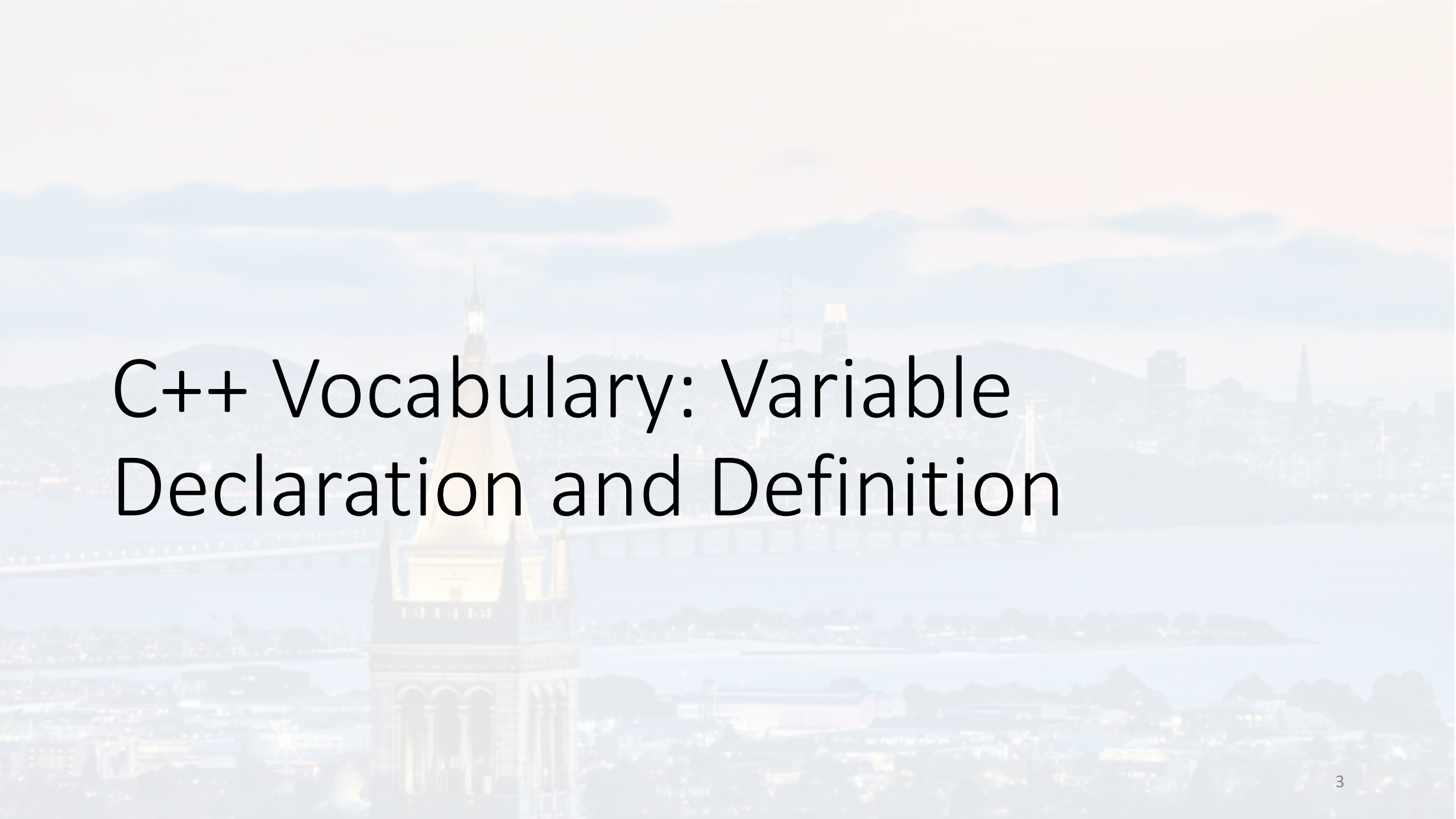
**Python for Molecular Sciences**

MSSE 272, 3 Units



### Outline

- C++ Vocabulary: Variable Definition and Declaration
- C++ Function Definition
- C++ Control Structures: Loops
- C++ Addresses and Pointers
- Data Types: Arrays and Vectors



# C++ Vocabulary: Variable Declaration and Definition





In C++ there are two “stages” to creating a variable or function.

These are variable declaration and variable definition.

Declaration defines the variable type and name. You can “declare” a variable in C++ without giving it a value.

Definition gives the variable a value.

```
int main() {  
  
    // "Declare" myVar.  
    // It exists but doesn't have  
    // a meaningful value  
    int myVar;  
  
    // Define myVar to be 10  
    myVar = 10;  
  
    // We usually do this all at once  
    int myVar = 10;  
  
    return 0;  
}
```



# C++ Control Structures: Loops



Initialization      Test      Increment

for(int i = 0; i < 10; i++)  
{  
}

```
#include <iostream>

void say_hello(std::string name)
{
    std::cout << "Hello, " << name << "!" << std::endl;
}

int main(void)
{
    for(int i = 0; i < 10; i++)
    {
        say_hello("MSSE Student");
    }
    return 0;
}
```





```
#include <iostream>

void say_hello(std::string name)
{
    std::cout << "Hello, " << name << "!" << std::endl;
}

int main(void)
{
    int i = 0;
    while(i < 10)
    {
        say_hello("Dr. Nash");
        i++;
    }
    return 0;
}
```



```
#include <iostream>

void say_hello(std::string name)
{
    std::cout << "Hello, " << name << "!" << std::endl;
}

int main(void)
{
    int i = 0;
    while(i < 10)
    {
        say_hello("Dr. Nash");
        i++;
    }

    return 0;
}
```

Initialization of variable outside of loop. Condition in parenthesis





```
#include <iostream>

void say_hello(std::string name)
{
    std::cout << "Hello, " << name << "!" << std::endl;
}

int main(void)
{
    int i = 0;
    while(i < 10)
    {
        say_hello("Dr. Nash");
        i++;
    }
    return 0;
}
```

Must increment or you will get an infinite loop!



```
#include <iostream>

int main(void)
{
    int i = 0;
    // Print only even numbers, up to 18
    for(int i = 0; i < 200; i++)
    {
        // if i is odd, then continue to the next iteration
        if(i % 2 == 1)
        {
            continue;
        }

        // print the number. This only gets run if
        // i is even
        std::cout << i << std::endl;

        // if i is 18, break out of the loop
        if(i == 18)
        {
            break;
        }
    }

    std::cout << "Loop done!" << std::endl;

    return 0;
}
```

**continue** : stops execution of the body and starts a new iteration from the beginning of the body

**break**: exits the loop immediately, with execution continuing on the statement after the loop



```
#include <iostream>

int main(void)
{
    int i = 0;
    // Print only even numbers, up to 18
    for(int i = 0; i < 200; i++)
    {
        // if i is odd, then continue to the next iteration
        if(i % 2 == 1)
        {
            continue;
        }

        // print the number. This only gets run if
        // i is even
        std::cout << i << std::endl;

        // if i is 18, break out of the loop
        if(i == 18)
        {
            break;
        }
    }

    std::cout << "Loop done!" << std::endl;
    return 0;
}
```

**continue** : stops execution of the body and starts a new iteration from the beginning of the body

**break**: exits the loop immediately, with execution continuing on the statement after the loop

**Can you trace the program execution? What will this print?**





```
#include <iostream>

int main(void)
{
    int i = 0;
    // Print only even numbers, up to 18
    for(int i = 0; i < 200; i++)
    {
        // if i is odd, then continue to the next iteration
        if(i % 2 == 1)
        {
            continue;
        }

        // print the number. This only gets run if
        // i is even
        std::cout << i << std::endl;

        // if i is 18, break out of the loop
        if(i == 18)
        {
            break;
        }
    }

    std::cout << "Loop done!" << std::endl;
    return 0;
}
```

For loop sets up i  
to go up to 200.



```
#include <iostream>

int main(void)
{
    int i = 0;
    // Print only even numbers, up to 18
    for(int i = 0; i < 200; i++)
    {
        // if i is odd, then continue to the next iteration
        if(i % 2 == 1)
        {
            continue;
        }

        // print the number. This only gets run if
        // i is even
        std::cout << i << std::endl;

        // if i is 18, break out of the loop
        if(i == 18)
        {
            break;
        }
    }

    std::cout << "Loop done!" << std::endl;
    return 0;
}
```

Check if the remainder of i  
divided by 2 is 1.



```
#include <iostream>

int main(void)
{
    int i = 0;
    // Print only even numbers, up to 18
    for(int i = 0; i < 200; i++)
    {
        // if i is odd, then continue to the next iteration
        if(i % 2 == 1)
        {
            continue;
        }

        // print the number. This only gets run if
        // i is even
        std::cout << i << std::endl;

        // if i is 18, break out of the loop
        if(i == 18)
        {
            break;
        }
    }

    std::cout << "Loop done!" << std::endl;
    return 0;
}
```

When we do not go through the previous conditional (ie i is even), we print the value of i.





```
#include <iostream>

int main(void)
{
    int i = 0;
    // Print only even numbers, up to 18
    for(int i = 0; i < 200; i++)
    {
        // if i is odd, then continue to the next iteration
        if(i % 2 == 1)
        {
            continue;
        }

        // print the number. This only gets run if
        // i is even
        std::cout << i << std::endl;

        // if i is 18, break out of the loop
        if(i == 18) ←
        {
            break;
        }
    }

    std::cout << "Loop done!" << std::endl;
    return 0;
}
```

We next check if *i* is equal to 18. If it isn't, we go to the next loop iteration.



```
#include <iostream>

int main(void)
{
    int i = 0;
    // Print only even numbers, up to 18
    for(int i = 0; i < 200; i++)
    {
        // if i is odd, then continue to the next iteration
        if(i % 2 == 1)
        {
            continue;
        }

        // print the number. This only gets run if
        // i is even
        std::cout << i << std::endl;

        // if i is 18, break out of the loop
        if(i == 18)
        {
            break;
        }
    }

    std::cout << "Loop done!" << std::endl;
    return 0;
}
```

Check if the remainder of i  
divided by 2 is 1.



```
#include <iostream>

int main(void)
{
    int i = 0;
    // Print only even numbers, up to 18
    for(int i = 0; i < 200; i++)
    {
        // if i is odd, then continue to the next iteration
        if(i % 2 == 1)
        {
            continue; ←
        }

        // print the number. This only gets run if
        // i is even
        std::cout << i << std::endl;

        // if i is 18, break out of the loop
        if(i == 18)
        {
            break;
        }
    }

    std::cout << "Loop done!" << std::endl;
    return 0;
}
```

If the remainder of i divided by 2 is one (aka if is odd), continue to next loop iteration.

This means everything in the loop below this “if” is skipped.





```
#include <iostream>

int main(void)
{
    int i = 0;
    // Print only even numbers, up to 18
    for(int i = 0; i < 200; i++)
    {
        // if i is odd, then continue to the next iteration
        if(i % 2 == 1)
        {
            continue;
        }

        // print the number. This only gets run if
        // i is even
        std::cout << i << std::endl;

        // if i is 18, break out of the loop
        if(i == 18)
        {
            break;
        }
    }

    std::cout << "Loop done!" << std::endl;
    return 0;
}
```

If i is equal to 18, we use “break” – we exit the loop entirely.



```
#include <iostream>

int main(void)
{
    int i = 0;
    // Print only even numbers, up to 18
    for(int i = 0; i < 200; i++)
    {
        // if i is odd, then continue to the next iteration
        if(i % 2 == 1)
        {
            continue;
        }

        // print the number. This only gets run if
        // i is even
        std::cout << i << std::endl;

        // if i is 18, break out of the loop
        if(i == 18)
        {
            break;
        }
    }

    std::cout << "Loop done!" << std::endl;
    return 0;
}
```

What does it print?

0  
2  
4  
6  
8  
10  
12  
14  
16  
18  
Loop done!



```
#include <iostream>

int main(void)
{
    int i = 0;
    // Print only even numbers, up to 18
    for(int i = 0; i < 200; i++)
    {
        // if i is odd, then continue to the next iteration
        if(i % 2 == 1)
        {
            continue;
        }

        // print the number. This only gets run if
        // i is even
        std::cout << i << std::endl;

        // if i is 19, break out of the loop
        if(i == 19)
        {
            break;
        }
    }

    std::cout << "Loop done!" << std::endl;
    return 0;
}
```

Quiz yourself –

What does this print and why?





Attempting to  
compile this code  
will result in a  
**compiler error.**

**error: 'inside\_loop'  
was not declared in  
this scope**

Why?

```
#include <iostream>

int main() {

    for (int i=0; i<10; i++) {
        int inside_loop = i*2;
    }

    std::cout << "Value of inside loop "
                << inside_loop
                << std::endl;

    return 0;
}
```



Think back to variable definition and declaration.

In C++, the scope of a variable is more strict. The variable `inside_loop` was declared and defined in the loop and does not exist outside of it.

```
#include <iostream>

int main() {

    for (int i=0; i<10; i++) {
        int inside_loop = i*2;
    }

    std::cout << "Value of inside loop "
                << inside_loop
                << std::endl;

    return 0;
}
```



Think back to variable definition and declaration.

Move declaration outside of loop and set value inside of for loop.

```
#include <iostream>

int main() {
    int inside_loop;

    for (int i=0; i<10; i++) {
        inside_loop = i*2;
    }

    std::cout << "Value of inside loop "
               << inside_loop
               << std::endl;

    return 0;
}
```





- C++ `for` and `while` loops allow control flow in programs similar in logic to Python `for` and `while` loops, but with different syntax.
  - Curly braces `{ }` define scope while semicolons terminate statements (contrast to Python which uses indentation)
- Use `for` loops when the number of iterations is known; `while` loops when the number of iterations depend on a condition.
- `continue` skips the rest of the current loop iteration. `break` exits the loop entirely
- Variables declared inside of a loop or any `{ }` in C++ are not accessible outside of the block.



# C++ Memory Management: Addresses and Pointers





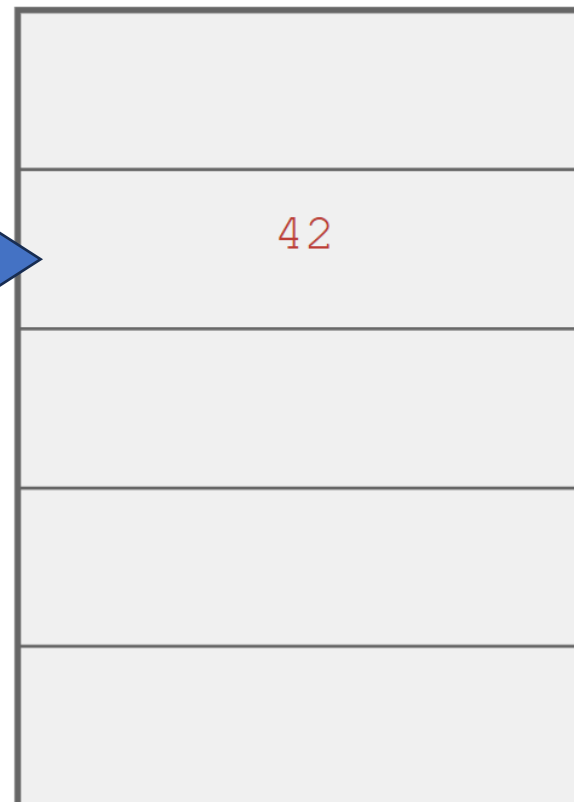
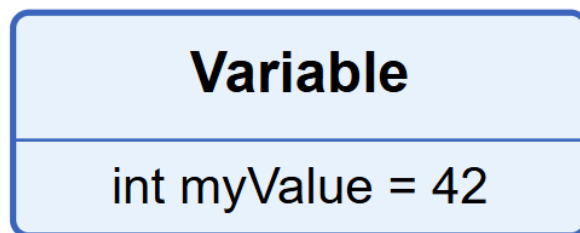
- When you write a computer program, every variable is stored in the computer's RAM (Random Access Memory) while the program is running.
- Each variable has a **location in memory (an address)**, a specific size that depends on type (we covered this last week)
- In Python, this is usually hidden from you.
- In C++, you will often work with the memory addresses directly.

The location in memory of a variable is called its **address**. A variable that points to a value through an address is called a **pointer**.





## Computer Memory

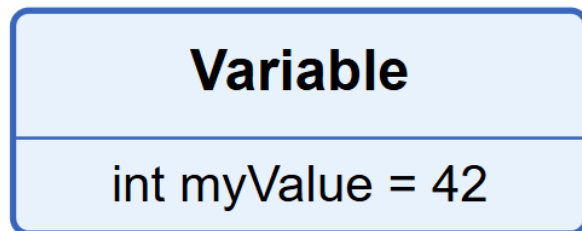


`myValue` is known as the variable's "name" or identifier. We can access the information in the variable using its name. This is what we have learned before.



The variable can also be identified by its location in memory. This is called its **address**.

## Computer Memory



`myValue` is known as the variable's "name" or identifier. We can access the information in the variable using its name. This is what we have learned before.

0x1000

0x1004

0x1008

0x100C

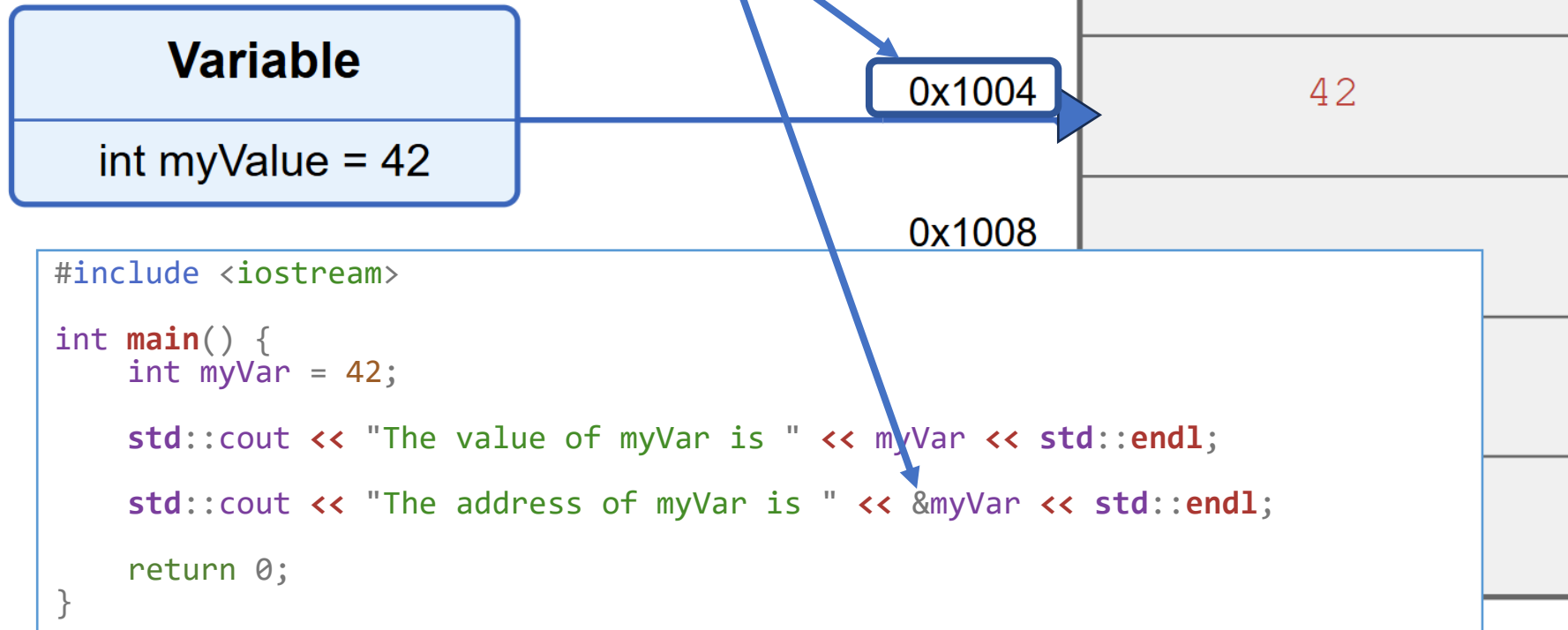
0x1010

42



We can access the location of a variable using & in front of the variable name. This gives us the memory address.

## Computer Memory







In C++, we can access a variable through its address by using a pointer.

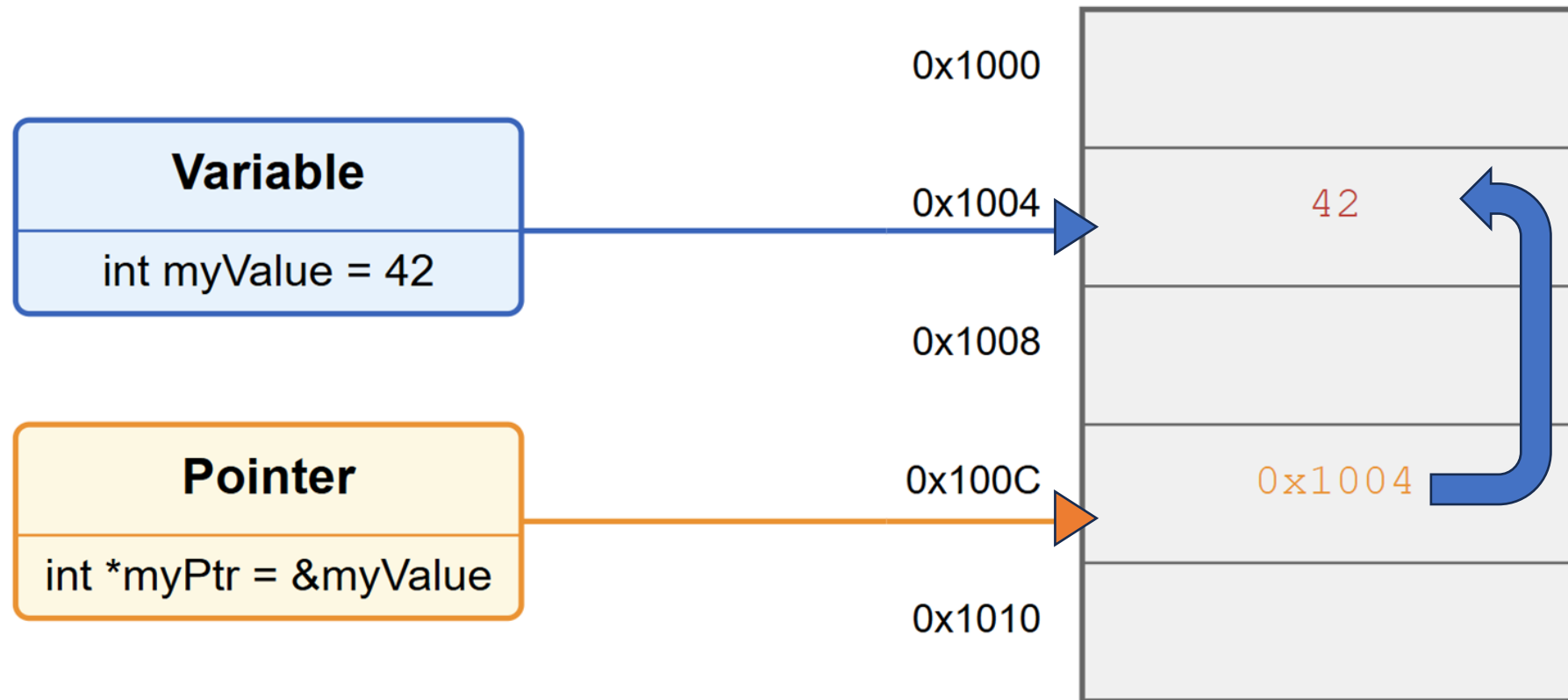
You can think of a pointer as a **special data type** that allows modification of a variable through its address.

Pointers are created using `*` before the variable name.

```
int *myptr = &myVar
```

Notice the use of `&` that we earlier saw corresponded to the variable's address

## Computer Memory





In C++, we can access a variable through its address by using a pointer.

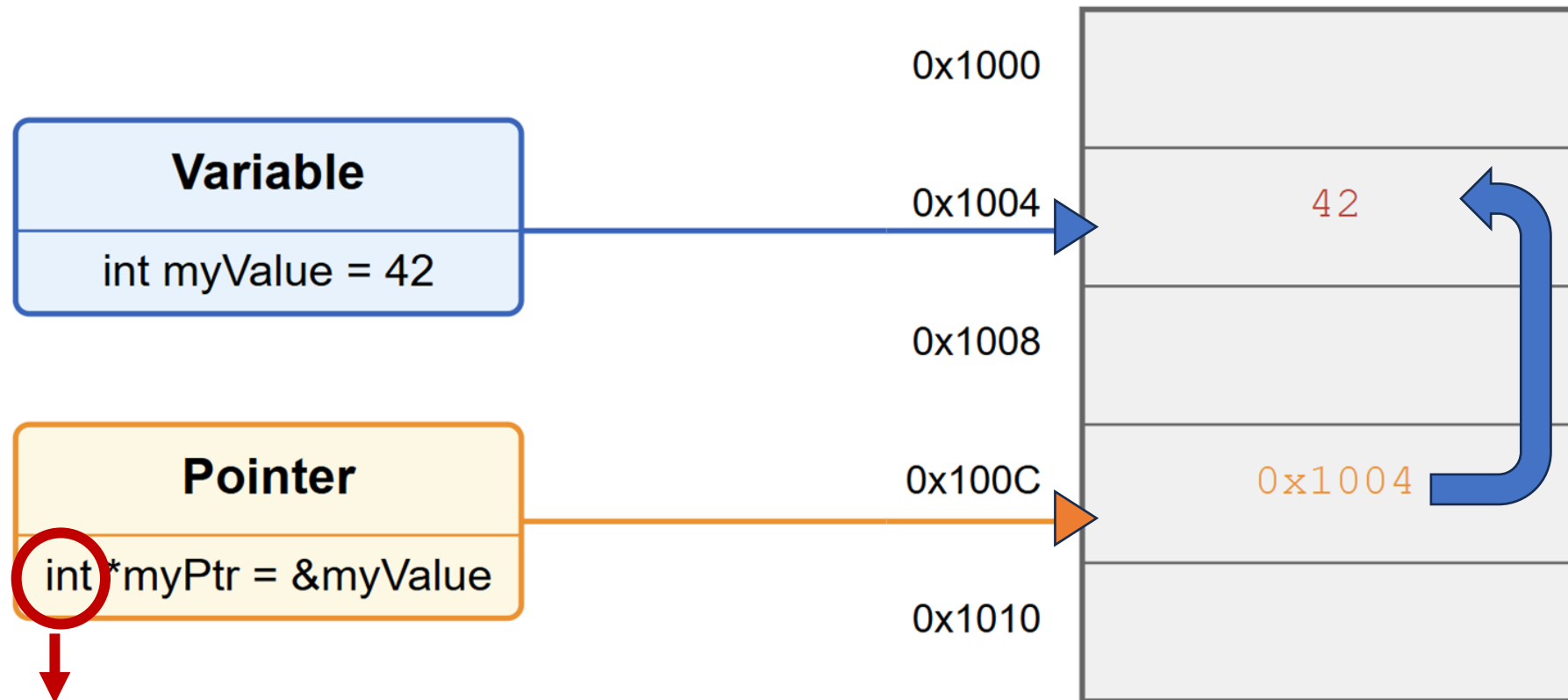
You can think of a pointer as a **special data type** that allows modification of a variable through its address.

Pointers are created using `*` before the variable name.

```
int *myptr = &myVar
```

Notice the use of `&` that we earlier saw corresponded to the variable's address

## Computer Memory



Notice that the type before `*` is the same type as the variable at the address.



- Pointers “point to” a variable whose address they store.
- You can access the value of the variable that is pointed to by using the *dereference operator*.
- The *dereference operator* is an asterisk in front of the variable name (similar to how we used it to define a pointer

```
#include <iostream>

int main() {
    int x = 42;      // A normal integer variable
    int *ptr = &x;   // Pointer storing the address of x

    std::cout << "Value of x: " << x << std::endl;
    std::cout << "Address of x: " << ptr << std::endl;

    // This will print 42. We are “dereferencing” ptr with the *
    std::cout << "Value pointed to by ptr: " << *ptr << std::endl;

    return 0;
}
```

Using the  
dereference  
operator.





We can modify the value of a variable through `*ptr`.

We'll see reasons we might do this in later sections!

```
#include <iostream>

int main() {
    // A normal integer variable
    int x = 42;

    // Pointer storing the address of x
    int* ptr = &x;

    // Modifying the value of x through ptr
    *ptr = 100;

    // This will output 100
    std::cout << "New value of x: " << x << std::endl;

    return 0;
}
```



We can modify the value of a variable through `*ptr`.

We'll see reasons we might do this in later sections!

```
#include <iostream>

int main() {
    // A normal integer variable
    int x = 42;

    // Pointer storing the address of x
    int *ptr = &x;

    // Modifying the value of x through ptr
    *ptr = *ptr + 5;

    // This will output 47
    std::cout << "New value of x: " << x << std::endl;

    return 0;
}
```



```
#include <iostream>
int main() {
    int a = 10;
    int b = 20;
    int *ptr = &a;

    std::cout << "ptr points to a: "
               << *ptr
               << std::endl;

    ptr = &b; // ptr now points to b
    std::cout << "ptr now points to b: "
               << *ptr
               << std::endl;

    return 0;
}
```

We can change the address a pointer points to – this is “pointer reassignment”.

After doing this, we could modify the value of b through ptr.





- Every variable has a memory address where its data is stored.
- You can access that address using the & operator (address-of)
- A pointer is a variable that stores a memory address.
- You can declare a pointer using \* (type of pointer should match type its pointing to)
- You “dereference” a pointer using \* to access or modify the value it points to.
- You can reassign a pointer to point to a different variable using assignment and &.



# C++: C-Style Arrays



# Warning!

The content on C style arrays in the next few slides are shown for completeness and so that you have context and get practice (there may be some high performance applications where they are preferable). However, we will learn a better way to do much of this next week, and that is the approach you should take most of the time!

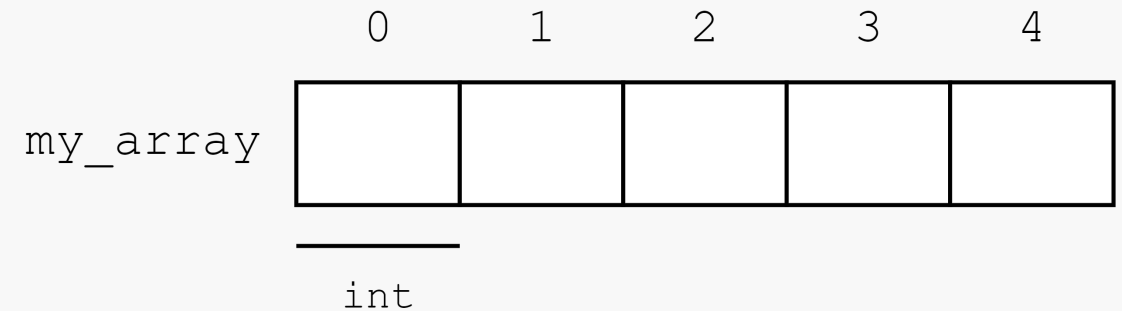




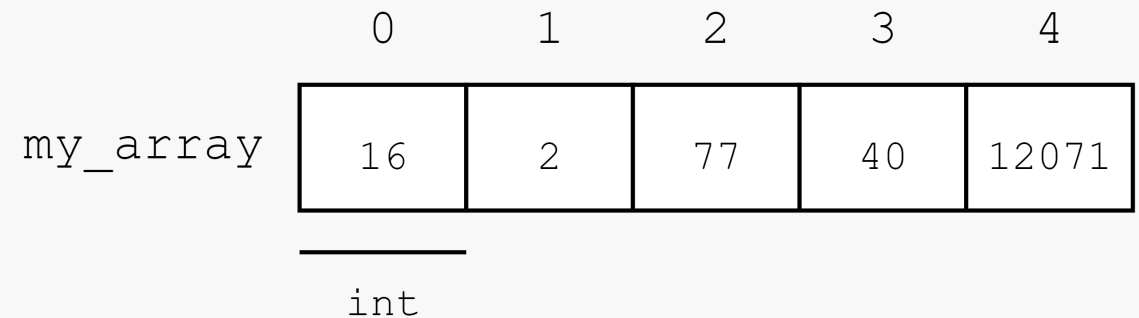
- Arrays are data structures that allow you to group objects together.
- Unlike Python lists (but similar to NumPy arrays), C arrays contain all of the same data type.
- Arrays in C++ are placed into **contiguous (sequential) memory**.
- C style arrays are zero indexed (like Python lists!) – the first element is element 0.
- Size fixed at compile time.

```
// general syntax  
// for declaring an array  
type name[num_elements];
```

```
int my_array[5]; // my_array is declared
```



```
int my_array[] = { 16, 2, 77, 40, 12071 };
```





## Array Creation

```
// Create an array with five elements  
int my_array[5];  
  
// Create an array with five elements  
// with all zero values  
int my_array[5] = {};  
  
// Create an array with five  
// elements and set the array values  
int my_array[5] = { 16, 2, 77, 40, 12071 };
```

## Array Access

```
// Access values of an array  
// using array index and []  
my_array[0]; // This will get 16  
  
// You can set values of arrays using  
// indexing  
my_array[0] = 100;
```



- Use a for loop and index into the array to perform array operations.

```
#include <iostream> // for std::cout, std::endl

int main(void)
{
    int arr[10] = { 3, 5, 7, 9, 11, 13, 15, 17, 19, 21 };

    for(int i = 0; i < 10; i++)
    {
        std::cout << "Element " << i << ": " << arr[i] << std::endl;
    }

    return 0;
}
```





- Use a for loop and index into the array to perform array operations.

```
#include <iostream> // for std::cout, std::endl

int main(void)
{
    int arr[10] = { 3, 5, 7, 9, 11, 13, 15, 17, 19, 21 };

    for(int i = 0; i < 10; i++)
    {
        std::cout << "Element " << i << ": " << arr[i] << std::endl;
    }

    return 0;
}
```

Index into array with  
counter variable.



There are two big limitations to C Style arrays that make them less preferable to use:

- (1) They do not check the bounds of the array. It is possible to index outside of the bounds, which will cause problems!
- (2) The size of the array must be known at compile time

**Let's see more about this in the next slides!**



- C style arrays do not include bounds checking. This means you can write to or attempt to access outside of the size of the array. This will result in problems with your code, even if everything appears okay!

```
#include <iostream>

int main() {

    int arr[5] = {1, 2, 3, 4, 5};

    for (int i=0; i<6; i++) {
        std::cout << i
                  << " "
                  << arr[i]
                  << std::endl;
    }

    return 0;
}
```





- C style arrays do not include bounds checking. This means you can write to or attempt to access outside of the size of the array. This will result in problems with your code, even if everything appears okay!

```
#include <iostream>

int main() {

    int arr[5] = {1, 2, 3, 4, 5};

    for (int i=0; i<6; i++) {
        std::cout << i
                  << " "
                  << arr[i]
                  << std::endl;
    }

    return 0;
}
```

## Output

```
0 1
1 2
2 3
3 4
4 5
5 32764
```



- C style arrays do not include bounds checking. This means you can write to or attempt to access outside of the size of the array. This will result in problems with your code, even if everything appears okay!

```
#include <iostream>

int main() {

    int arr[5] = {1, 2, 3, 4, 5};

    for (int i=0; i<6; i++) {
        std::cout << i
                  << " "
                  << arr[i]
                  << std::endl;
    }

    return 0;
}
```

## Output

0 1

1 2

2 3

3 4

4 5

5 32764

Not part of our array! (If you try this example, you will probably see a different number)



- Sometimes writing outside of the bounds will cause an error – this is sometimes referred to as “stack smashing”.

```
#include <iostream>
```

```
int main() {
```

```
    int arr[5] = {1, 2, 3, 4, 5};
```

```
    for (int i=0; i<7; i++) {
```

```
        arr[i] = arr[i] + 1;
```

```
        std::cout << i
```

```
            << " "
```

```
            << arr[i]
```

```
            << std::endl;
```

```
    }
```

```
    return 0;
```

```
}
```

Increased count by one

Modifying array  
values





- Sometimes writing outside of the bounds will cause an error – this is sometimes referred to as “stack smashing”.

```
#include <iostream>

int main() {

    int arr[5] = {1, 2, 3, 4, 5};

    for (int i=0; i<7; i++) {
        arr[i] = arr[i] + 1;
        std::cout << i
                    << " "
                    << arr[i]
                    << std::endl;
    }
    return 0;
}
```

## Output

0 2

1 3

2 4

3 5

4 6

5 32765

6 310445569

\*\*\* stack smashing detected \*\*\*:  
terminated

Aborted (core dumped)



Traditionally C style arrays  
require that you know the size at  
compile time.

Let's look at this code.

```
#include <iostream>

int main() {

    int size;

    std::cout << "Input a vector size"
               << std::endl;

    std::cin >> size;

    int arr[size] = {};

    for (int i=0; i<size; i++) {
        arr[i] = i;
    }
    return 0;
}
```



Variable size is  
declared but has  
no value.

```
#include <iostream>
```

```
int main() {
```

```
    int size;
```

```
    std::cout << "Input a vector size" << std::endl;
```

```
    std::cin >> size;
```

```
    int arr[size] = {};
```

```
    for (int i=0; i<size; i++) {  
        arr[i] = i;
```

```
    }
```

```
    return 0;
```

```
}
```





Variable size is declared but has no value.

This is new. `std::cin` takes input from the user when run and stores it in `size`.

```
#include <iostream>

int main() {

    int size;

    std::cout << "Input a vector size"
               << std::endl;

    std::cin >> size;

    int arr[size] = {};

    for (int i=0; i<size; i++) {
        arr[i] = i;
    }
    return 0;
}
```



Variable size is declared but has no value.

This is new. `std::cin` takes input from the user when run and stores it in `size`.

Create an array of specified size. Not known at compile time.

```
#include <iostream>
```

```
int main() {
```

```
    int size;
```

```
    std::cout << "Input a vector size" << std::endl;
```

```
    std::cin >> size;
```

```
    int arr[size] = {};
```

```
    for (int i=0; i<size; i++) {  
        arr[i] = i;  
    }
```

```
    return 0;
```

```
}
```



This may work for you, but it is not guaranteed to work! (May successfully compile and run on one computer, but not another one)

```
#include <iostream>

int main() {

    int size;

    std::cout << "Input a vector size"
               << std::endl;

    std::cin >> size;

    int arr[size] = {};

    for (int i=0; i<size; i++) {
        arr[i] = i;
    }
    return 0;
}
```





```
#include <iostream>
```

```
int main(void) {
```

```
// I can also allocate arrays with pointers
```

```
int *p_arr = new int[5];
```

```
//Fill in the array
```

```
for (int i=0; i<5; i++) {
```

```
    p_arr[i] = i*2;
```

```
}
```

```
// Print out the array
```

```
for (int i=0; i<5; i++) {
```

```
    std::cout << "Element "
```

```
    << i
```

```
    << " : "
```

```
    << p_arr[i]
```

```
    << std::endl;
```

```
}
```

```
delete p_arr;
```

```
}
```

Use a pointer and the "new" keyword

Can index with counter same way as array

Must delete when done.



Pointers can be used for dynamic allocation

Use a pointer and the “new” keyword

```
#include <iostream>

int main() {

    int size;

    std::cout << "Input a vector size"
               << std::endl;

    std::cin >> size;

    int * arr = new int[size];

    for (int i=0; i<size; i++) {
        arr[i] = i;
    }

    // When done, free it
    delete [] arr;

    return 0;
}
```



Use a pointer and  
the “new” keyword

What would you happen if you printed  
arr to the screen? (You would see  
something that looks like a pointer)

arr is a pointer to the first element of  
an array.

```
#include <iostream>

int main() {

    int size;

    std::cout << "Input a vector size"
               << std::endl;

    std::cin >> size;

    int * arr = new int[size]

    for (int i=0; i<size; i++) {
        arr[i] = i;
    }

    // When done, free it
    delete [] arr;

    return 0;
}
```





## Stop and think!

Anything surprising about this code given what we know about pointers so far?

```
#include <iostream>

int main() {

    int size;

    std::cout << "Input a vector size"
               << std::endl;

    std::cin >> size;

    int * arr = new int[size]

    for (int i=0; i<size; i++) {
        arr[i] = i;
    }

    // When done, free it
    delete [] arr;

    return 0;
}
```



## Stop and think!

Anything surprising about this code given what we know about pointers so far?

How are we setting a value to a pointer here without dereferencing?

```
#include <iostream>

int main() {

    int size;

    std::cout << "Input a vector size"
               << std::endl;

    std::cin >> size;

    int * arr = new int[size]

    for (int i=0; i<size; i++) {
        arr[i] = i;
    }

    // When done, free it
    delete [] data;

    return 0;
}
```



## Stop and think!

Anything surprising about this code given what we know about pointers so far?

How are we setting a value to a pointer here without dereferencing?

In C++, using `[]` with a C-style array is a “short cut” for dereferencing a pointer

`arr[i]` is equivalent to `*(arr + i)` – `arr` refers to a location in memory and `i` is equivalent to the amount of space to move along the array.

```
#include <iostream>

int main() {

    int size;

    std::cout << "Input a vector size"
               << std::endl;

    std::cin >> size;

    int * arr = new int[size]

    for (int i=0; i<size; i++) {
        arr[i] = i;
    }

    // When done, free it
    delete [] data;

    return 0;
}
```





# Introduction to C++ Functions



Return	Name	Arguments/Signature	
double	calculate_area	(double length , double width )	
{			
return length * width;			Body
}			

- The syntax for defining a function is shown in the figure above. When defining a function in C++, we start by specifying the return type, giving the function name, and in parenthesis giving the function arguments with their types.
- The example defines a function called “calculate\_area”. The function returns a double and takes in two doubles as arguments.



Return	Name	Arguments/Signature	
double	calculate_area	(double length= 10.0 , double width= 5.0 )	
	{		
	double area = length * width;		
	return area;		Body
	}		

- Default arguments can be added in a way similar to Python by using the equals sign and setting the argument equal to a value.
- Unlike Python, you must always match the default argument to the specified type. In Python, we might sometimes use a default of None.





By default, C++ uses what is called “pass by value”.

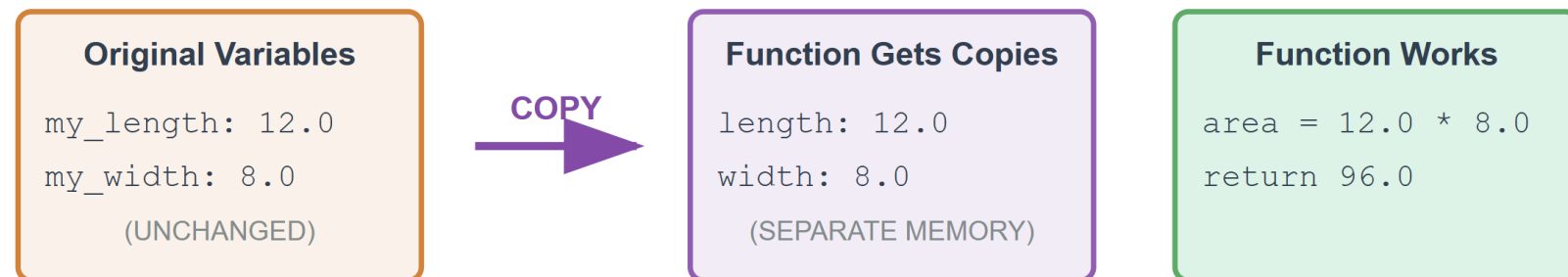
This means that every variable passed to a function is copied. A new copy of it is created in memory.

There are other ways to pass variables so that they are not copied, but we will learn more about this later.

### 1. Before function call:

```
double my_length = 12.0;  
double my_width = 8.0;  
calculate_area(my_length, my_width);
```

### 2. During function call - VALUES ARE COPIED:



### The Key Point:

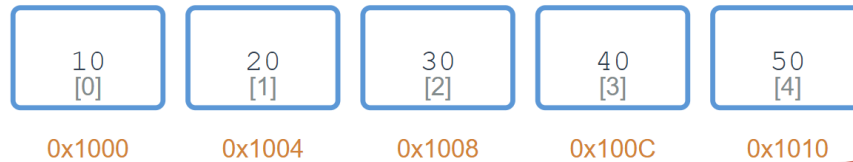
The function gets COPIES of your values, not the original variables.  
Your original variables stay exactly the same!



## 1. Array in memory:

```
int numbers[5] = {10, 20, 30, 40, 50};  
process_array(numbers, 5);
```

### Array in Memory:



Address passed to function

### Function Receives:

- A POINTER to first element
- Address: 0x1000
- NOT a copy of the array!

## 2. What gets passed to the function:

### Function Definition (these are equivalent):

```
void process_array(int arr[], int size) // Array notation  
void process_array(int *arr, int size) // Pointer notation (same thing!)
```

### Key Differences from Regular Variables:

- Arrays are NOT copied - only the address is passed
- Function can modify the original array elements
- Much more efficient for large arrays (no copying overhead)



Demo





- In C++, you must declare a variable's type before using it. Declaration sets the type and name, while definition assigns a value
- for and while loops allow repeating tasks. continue skips to the next iteration, and break exits the loop.
- A pointer stores a memory address. & gives the address, and \* accesses or modifies the value at that address.
- C-style arrays are fixed-size and lack bounds checking, which can cause errors.
- Dynamic allocation with new creates memory at runtime; delete is required to avoid memory leaks.
- Functions in C++ must define the return type and all of the types of arguments
- By default functions are (mostly) pass by value (a copy is made of arguments)
- Arrays passed to functions decay to pointers.

# Thank you for your attention!

