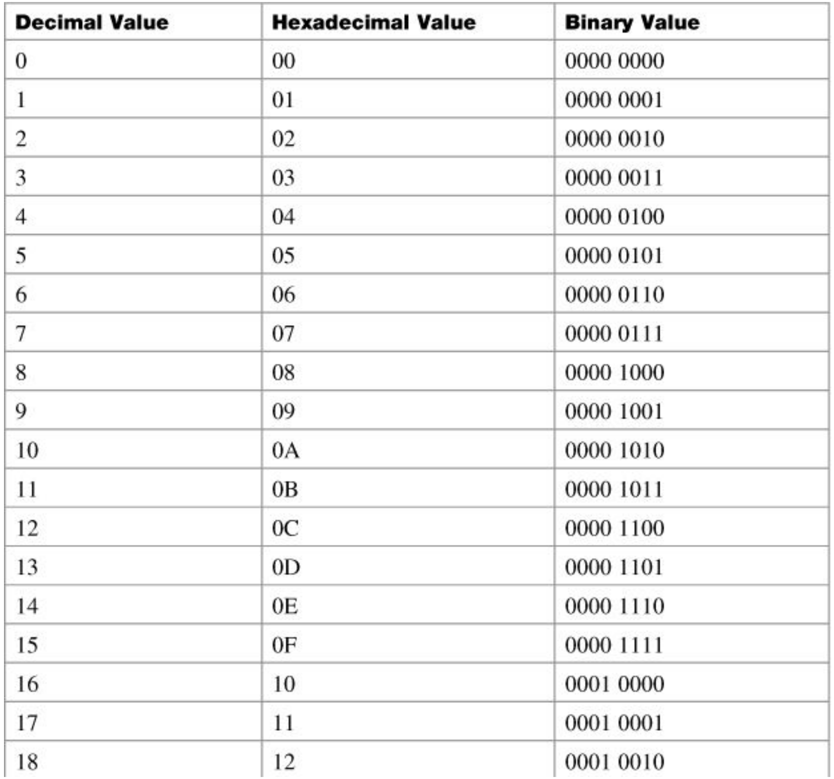
Chapter 7 - Pointers

CSE 2010
Week 10

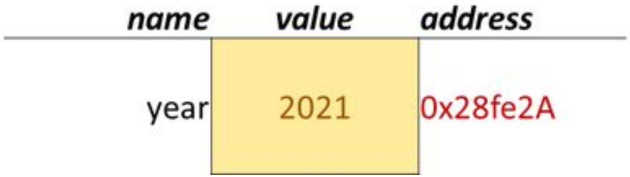
Background

- As we briefly discussed at the beginning of this course, computer memory is made up of sequences of bytes (1 byte = 8 bits, and 1 bit is either 0 or 1)
  - Different data types require different # of bytes
  - Each byte of memory has a **physical memory address** represented in hexadecimal
    - The #'s we are used to are in decimal form, base 10
    - Binary = base 2
    - Hexadecimal = base 16



Decimal Value	Hexadecimal Value	Binary Value
0	00	0000 0000
1	01	0000 0001
2	02	0000 0010
3	03	0000 0011
4	04	0000 0100
5	05	0000 0101
6	06	0000 0110
7	07	0000 0111
8	08	0000 1000
9	09	0000 1001
10	0A	0000 1010
11	0B	0000 1011
12	0C	0000 1100
13	0D	0000 1101
14	0E	0000 1110
15	0F	0000 1111
16	10	0001 0000
17	11	0001 0001
18	12	0001 0010

Background



- When we have a variable or objects, their names are labels for specific locations in memory containing a value we can reuse.
  - This is helpful because without names we would need to refer to the physical memory address (imagine having to memorize hexadecimal numbers!)

- To get the address of a variable, we can use the address operator (&)
  - We've used this operator before when we pass by reference.
  - Recall that parameters passed by reference refer to an already existing value in memory.

- We are now going to learn that it is beneficial to obtain the address of a variable or object in order to complete specific tasks.

What are pointers?

- Pointers are variables whose values are **memory addresses**.
- These memory addresses are the addresses of other variables or objects.

Declaring/Initializing pointers:

1. `datatype* pointerName = &existing variable/object;`

   a.
   ```
   int year = 2021;
   int* pYear = &year; //pYear = memory address of year
   ```

   b.
   ```
   double salary = 50000.00;
   double* pSalary = &salary;
   ```

   c.
   ```
   Pet pet1(“Obi”, “cat”, ‘M’, 18.0);
   Pet* p = &pet1;
   ```

   Note: the pointer datatype needs to match the type of the variable whose address it stores.

2. `datatype* pointerName = NULL;`

   a.
   ```
   double* pPrice = NULL;
   ```

   Note: Use this method when you need to declare a pointer but don’t yet know what it needs to point to.

Let’s take a close look at a simple example

```
int year = 2021; //declaring an int variable
int* pYear = &year; //declaring an int pointer variable

cout << “Value of year: “ << year << “\n”;
cout << “Address of year: “ << &year << “\n”;
cout << “Value of pYear: “ << pYear << “\n”;
cout << “Address of pYear: “ << &pYear << “\n”;
```

Output:

```
Value of year: 2021
Address of year: 0x28fe2A
Value of pYear: 0x28fe2A
Address of pYear: 0x28fe63
```

name	***value***	***address***
year	2021	0x28fe2A
pYear	0x28fe2A	0x28fe63

Dereferencing Pointers

- Creating pointers allows us to use the address stored to access the value at that address using the dereference operator - *
- Example:

```
int year = 2021;
int* pYear = &year;
cout << “Value of pYear: “ << pYear << “\n”; //displays 0x28fe2A
cout << “Value of *pYear: “ << *pYear << “\n”; // displays 2021
```

Let’s see a programming example…

name	*value*	*address*
year	2021	0x28fe2A
pYear	0x28fe2A	0x28fe63

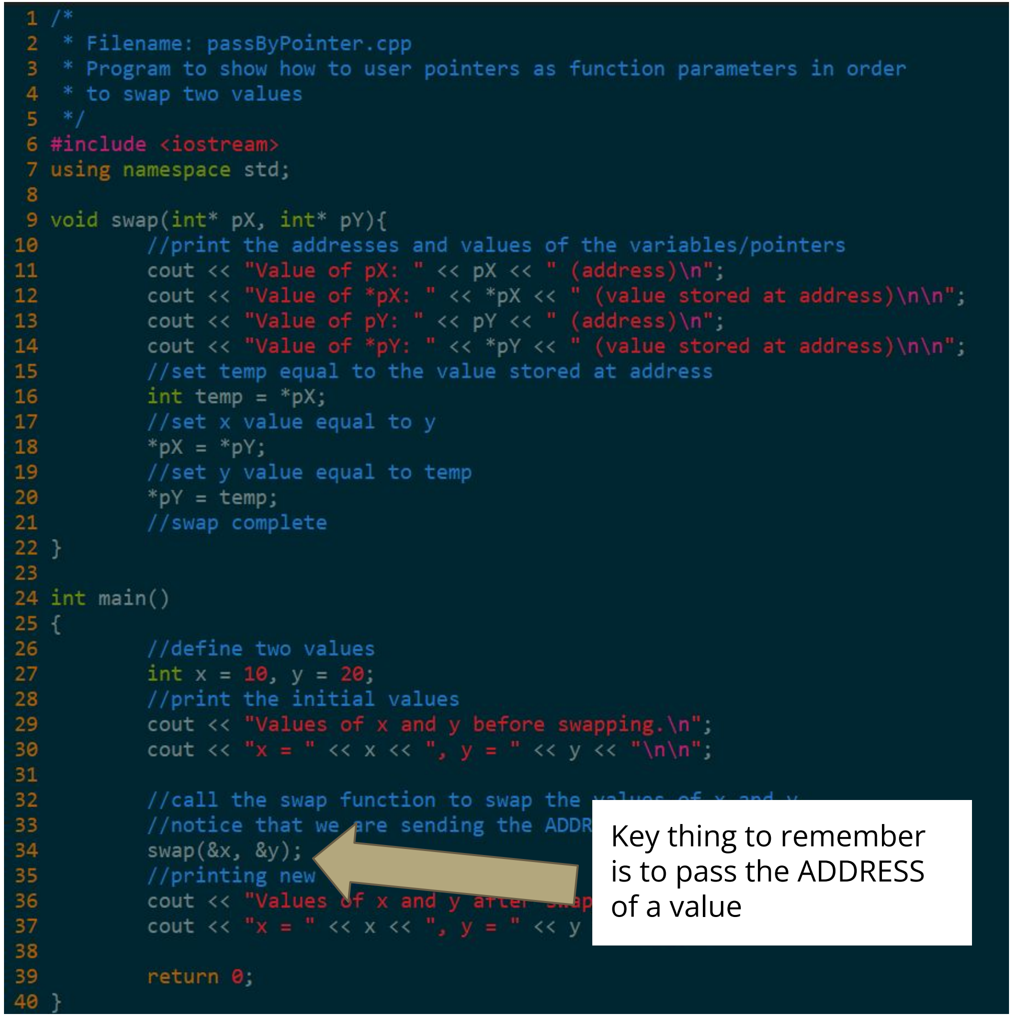
You might be thinking…….

- Why do we use pointers and * when we can just directly access the value with the variable name?
- Depending on the task at hand, we will need to either directly or indirectly access a value and pointers provide a more efficient way of doing this and sometimes it’s the only way to accomplish it.
    - Passing arguments by pointers to functions
    - Representing, accessing, and modifying arrays.
    - Used for iterators in STL library
    - Accessing heap memory, modifying a value inside of a function
    - Sharing of attributes between classes (We’ll learn this next chapter!)

Pass by Pointer

- We’ve learned the difference between pass-by-value and pass-by-reference, now let’s look at pass-by-pointer.
- In the pass-by-pointer methods, the calling function sends the **address of the argument** to the called function, and the called function stores it in a pointer.
- What is the difference between pass by pointer and pass by reference?
  - **Pass-by-reference:** argument and parameter share the **same memory**. The parameter refers to an already existing space in memory.
  - **Pass-by-pointer:** parameter stores the **address of the argument** and uses it to change its values as needed.
  - Both of these methods costs less than pass-by-value.

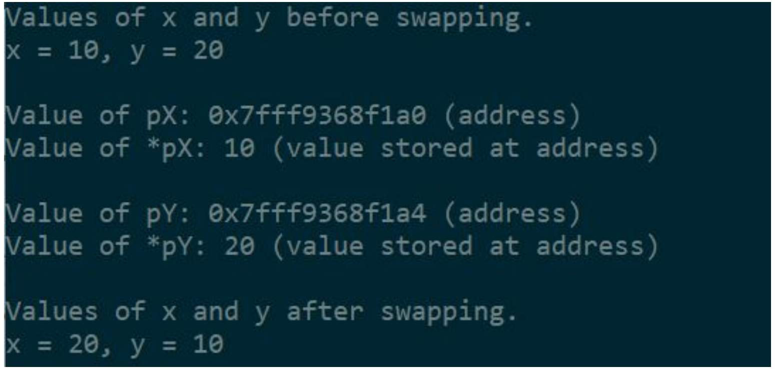
Swapping with Pointers



```cpp
/*
 * Filename: passByPointer.cpp
 * Program to show how to user pointers as function parameters in order
 * to swap two values
 */
#include <iostream>
using namespace std;

void swap(int* pX, int* pY){
        //print the addresses and values of the variables/pointers
        cout << "Value of pX: " << pX << " (address)\n";
        cout << "Value of *pX: " << *pX << " (value stored at address)\n\n";
        cout << "Value of pY: " << pY << " (address)\n";
        cout << "Value of *pY: " << *pY << " (value stored at address)\n\n";
        //set temp equal to the value stored at address
        int temp = *pX;
        //set x value equal to y
        *pX = *pY;
        //set y value equal to temp
        *pY = temp;
        //swap complete
}

int main()
{
        //define two values
        int x = 10, y = 20;
        //print the initial values
        cout << "Values of x and y before swapping.\n";
        cout << "x = " << x << ", y = " << y << "\n\n";

        //call the swap function to swap the values of x and y
        //notice that we are sending the ADDR
        swap(&x, &y);
        //printing new
        cout << "Values of x and y after swap
        cout << "x = " << x << ", y = " << y

        return 0;
}
```

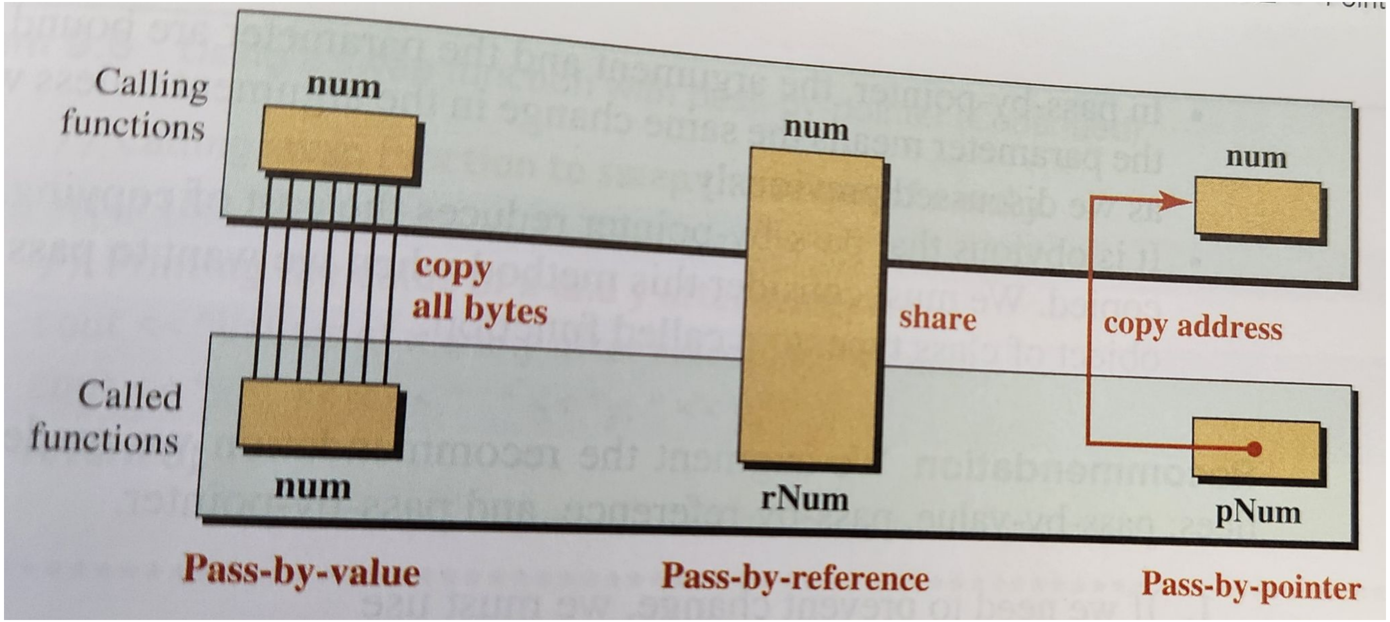
Key thing to remember is to pass the ADDRESS of a value

Output:



```
Values of x and y before swapping.
x = 10, y = 20

Value of pX: 0x7fff9368f1a0 (address)
Value of *pX: 10 (value stored at address)

Value of pY: 0x7fff9368f1a4 (address)
Value of *pY: 20 (value stored at address)

Values of x and y after swapping.
x = 20, y = 10
```

Different ways of Passing Data to Functions



Arrays + Pointers = BFFs

- We've already been using pointers without even realizing it..
- `int arr[5];//what actually happens in memory?`
  - The system creates 5 sequential memory locations of type `int`.
  - The system then creates a **constant pointer** of type `int` that points to the first element of the array, this pointer by default is called **`arr`.**

- A constant pointer means that its contents (an address) cannot be changed, therefore that pointer is always pointing at the first element in the array.

- The **address** of the elements at:
  - index 0 = `arr+0`
  - index 1 = `arr+1`
  - index 2 =`arr+2` , and so on.
- This means that we can access array elements using the (*) operator, similarly to [ ]

```
arr[0] == *(arr + 0)
arr[1] == *(arr + 1)
arr[2] == *(arr + 2)
```

Pointer Arithmetic

Pointer arithmetic allows a limited number of arithmetic operators to be applied to pointer types.

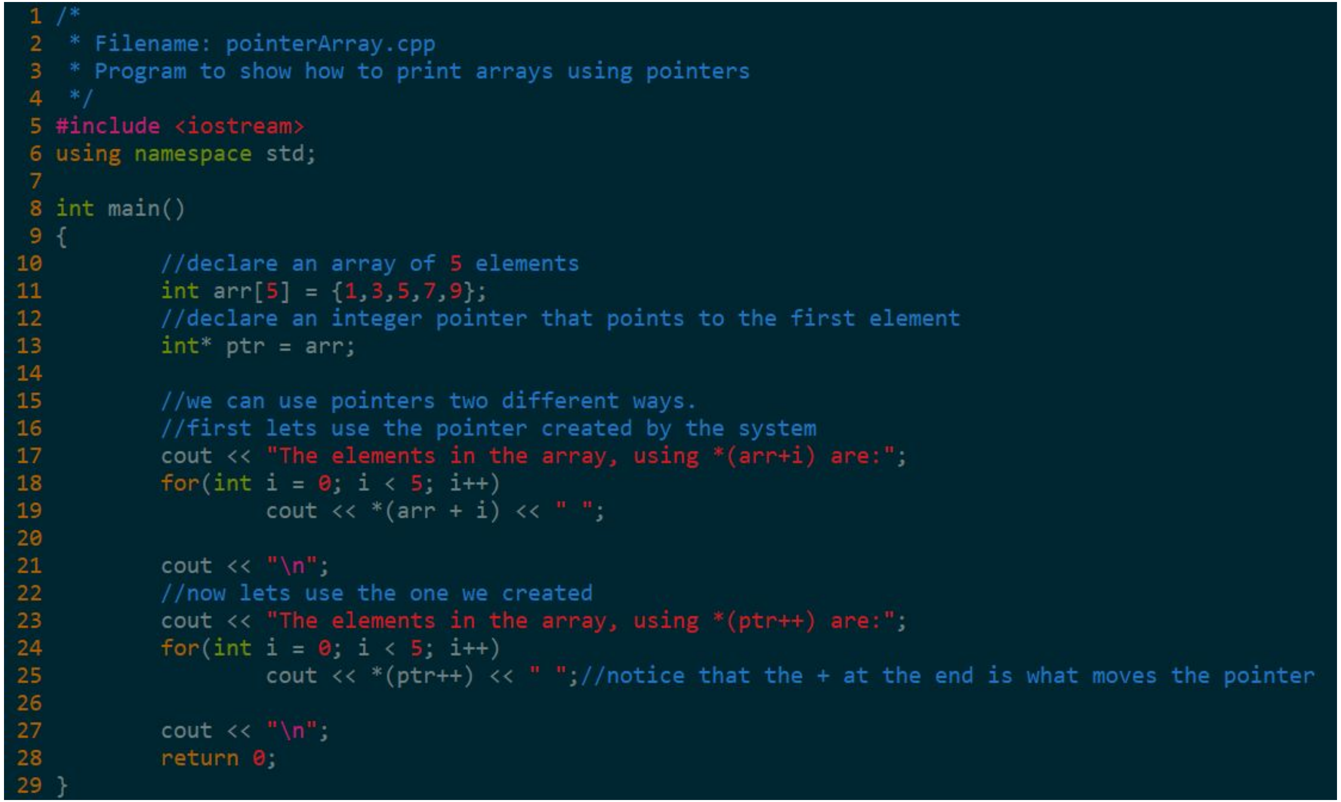
- Addition: +, ++, += (forward)
- Subtraction: - ,--,-= (backward)

When used with pointers, these operators move pointers forward and backward in memory (increase or decrease their address values).

Example:

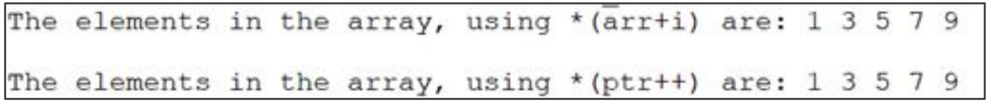
```
int arr[5] = {0,2,4,6,8};
//set ptr equal to the pointer created for the array
int* ptr = arr; // pointing to 0
ptr = ptr + 3; // pointing to 6
ptr--;//pointing to 4
```

Using pointers to step through arrays



```cpp
/*
 * Filename: pointerArray.cpp
 * Program to show how to print arrays using pointers
 */
#include <iostream>
using namespace std;

int main()
{
        //declare an array of 5 elements
        int arr[5] = {1,3,5,7,9};
        //declare an integer pointer that points to the first element
        int* ptr = arr;

        //we can use pointers two different ways.
        //first lets use the pointer created by the system
        cout << "The elements in the array, using *(arr+i) are:";
        for(int i = 0; i < 5; i++)
                cout << *(arr + i) << " ";

        cout << "\n";
        //now lets use the one we created
        cout << "The elements in the array, using *(ptr++) are:";
        for(int i = 0; i < 5; i++)
                cout << *(ptr++) << " ";//notice that the + at the end is what moves the pointer

        cout << "\n";
        return 0;
}
```

Output:

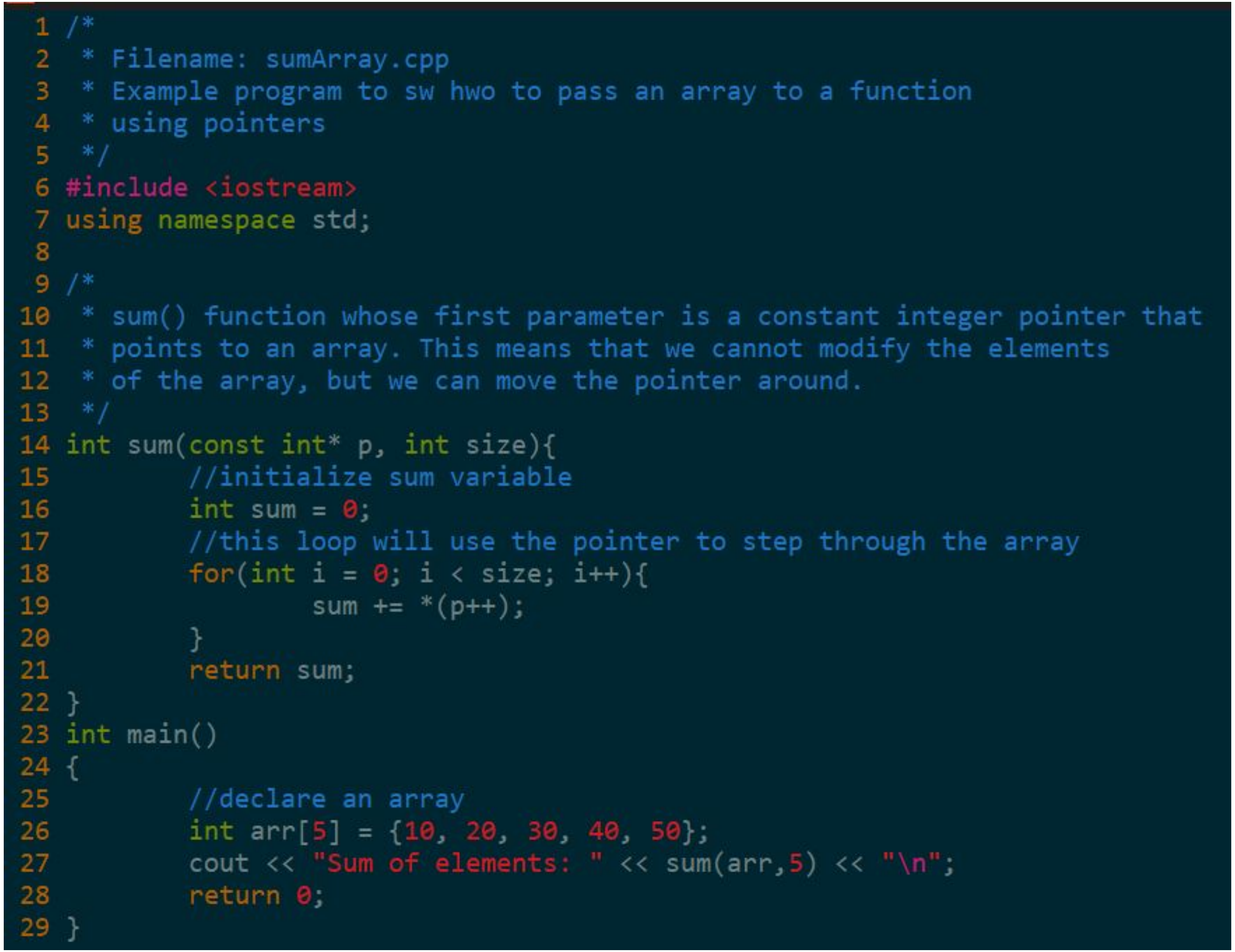


```
The elements in the array, using *(arr+i) are: 1 3 5 7 9

The elements in the array, using *(ptr++) are: 1 3 5 7 9
```

Passing a Pointer to a Function for an Array

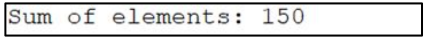
- We can pass a pointer to a function instead of passing the array.
- The following two prototypes are the same:

```
int sum(const int arr[], int size); //passing an array
int sum(const int* p, int size); //passing a pointer for an array
```



```cpp
/*
 * Filename: sumArray.cpp
 * Example program to sw hwo to pass an array to a function
 * using pointers
 */
#include <iostream>
using namespace std;

/*
 * sum() function whose first parameter is a constant integer pointer that
 * points to an array. This means that we cannot modify the elements
 * of the array, but we can move the pointer around.
 */
int sum(const int* p, int size){
        //initialize sum variable
        int sum = 0;
        //this loop will use the pointer to step through the array
        for(int i = 0; i < size; i++){
                sum += *(p++);
        }
        return sum;
}
int main()
{
        //declare an array
        int arr[5] = {10, 20, 30, 40, 50};
        cout << "Sum of elements: " << sum(arr,5) << "\n";
        return 0;
}
```

Output:



```
Sum of elements: 150
```

Iterators

- Iterators are special pointers specifically for STL containers (vector, list, deque, map, set, stack, etc), that allow us to easily step through and access elements in these containers.

- Similarly to pointers, vector iterators have the ability to step through elements in the vector in a forward and backward direction (done with ++ and – operators).

- We can access the value an iterator is pointing to with (*)

- Syntax to declare an iterator:

```
vector<datatype>::iterator iteratorName;
```

- We can initialize iterators with special vector functions:
  - `vectorName.begin()`
    - This function returns an iterator that points to the first element in the vector.
  - `vectorName.end()`
    - This function returns an iterator that **points directly after the last element** in the vector.

Iterator Examples

```
vector <int> vec1{2,4,7,11};
vector<int> :: iterator i = vec1.begin(); //points to the element 2
i++; // the iterator will move to the right
cout << *i << “\n”; // will display the value of 4
//display all elements inside vec1 using an iterator
for(vector<int>::iterator i = vec1.begin(); i != vec1.end(); i++)
{
  cout << *i << “ “;
}
```

Inserting Values into a Vector

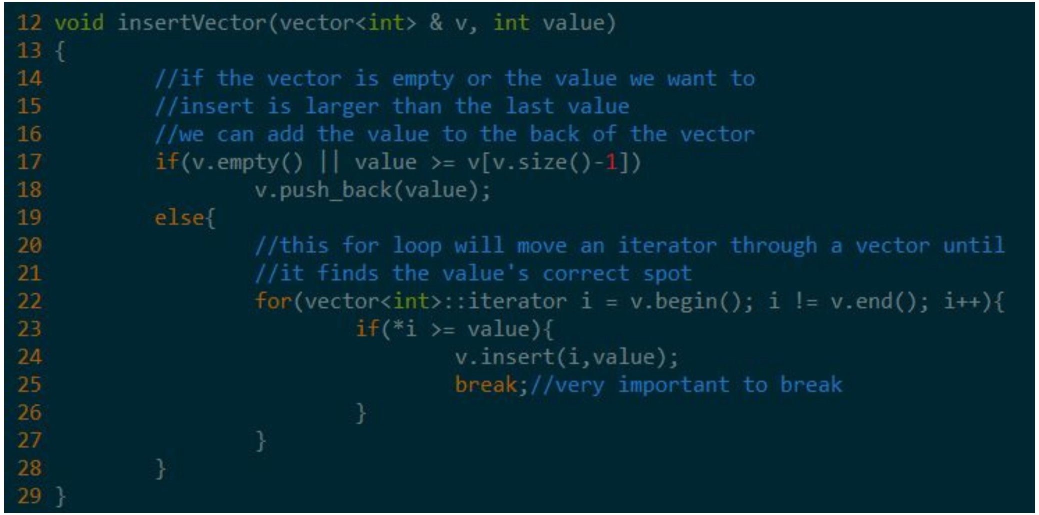
- We can use push_back() to add values to the end of a vector, but if we want to add values to a random location in the vector, we can use insert().
- `vectorName.insert(iterator,value)`
  - This function inserts the value into the position **BEFORE** where the iterator is pointing.
  - It also increases the size of the vector by 1.
- Inserting into an arbitrary place (order does not matter)
  - `vectorName.insert(vectorName.begin() + i,value)`
    - Where i is an index #. The function call will insert the value BEFORE the element at index i, and shift all elements to the right.
- Example:

```
vector <int> vec1{33,10,11,9}; //vec1 = 33 10 11 9

vec1.insert(vec1.begin() + 2, 16); //vec1 = 33 10 16 11 9
```

Inserting into an ordered vector to preserve the order

If you need to insert a value into a SORTED vector and need to make sure you preserve the order, you can use this method:

1. Use an iterator to iterate through each element in a vector.
2. As are are iterating through, check the value of the current element.
    a. If the current element is greater than or equal to the value we want to insert, we call the insert function.
    b. If the current element is less than the value we want to insert, we go to the next element.



```
void insertVector(vector<int> & v, int value)
{
        //if the vector is empty or the value we want to
        //insert is larger than the last value
        //we can add the value to the back of the vector
        if(v.empty() || value >= v[v.size()-1])
                v.push_back(value);
        else{
                //this for loop will move an iterator through a vector until
                //it finds the value's correct spot
                for(vector<int>::iterator i = v.begin(); i != v.end(); i++){
                        if(*i >= value){
                                v.insert(i,value);
                                break;//very important to break
                        }
                }
        }
}
```