Pointers

- Pointers are a special type of variable that contain a memory address as their value.
- The memory address points to another variable.
- A pointer variable uses *indirection* to reference another variable.
- By convention pointer variables begin with ‘p’ or ‘ptr’ to denote them as pointers.
- To declare a pointer place the *indirection* operator (*) after the datatype.

```
int age = 30;
int* pAge;
```

- Use the *unary* operator (&) in front of the variable to get the reference for that variable.
- The unary operator is often referred to as *“address of”* operator.

```
// pAge pointer holds the reference address of age
int* pAge = &age;
```

- It’s worth looking at exactly what these values hold using cout.

```
cout << " pAge: " << pAge << endl;
cout << "*pAge: " << *pAge << endl;
```

The `0x` prefix you see printed for the pointer denotes a hexadecimal memory address.

```
 pAge: 0x7fff5fbff7dc
*pAge: 30
```

- You can assign the contents of what a pointer points to to a variable, like so,

```
// set x to value at pAge
int x = *pAge;
```

- Pointers may only hold a memory addresses, 0, or the `NULL` value.

```
int* ptrNull = NULL;
cout << "ptrNull: " << ptrNull << endl;
```

- Assigning non address values, such as numbers or characters, results in compile time error.
- You may *indirectly* assign value of “what a pointer points to” using the indirection operator!

```
// indirectly set age via the pointer
*pAge = 24;
cout << "age: " << age << endl;
```

Indirection example

- Look very closely at this example. Make sure you understand it.

```
int x = 5;
int y = 10;
int* ptr = NULL;
cout << "ptr points to: " << ptr << endl;

// assign ptr to memory address of y
ptr = &y;
cout << "ptr now points to: " << ptr << endl;
// change the value of x to the value of y
x = *ptr;
cout << "The value of x is now: " << x << endl;
// change the value of y to 15
*ptr = 15;
cout << "The value of y is now: " << y << endl;
```

Arrays are pointers.

- The array name is actually a pointer to the memory location of the zeroth element in the array.
- You can see this by printing out the array pointer using cout.

```
    // arrays are actually pointers
    int arr[] = {5, 4, 3, 2, 1};
    cout << " arr: " << arr << endl;
    cout << "*arr: " << *arr << endl;
```

- It is not necessary to deal with unary (&) or indirection (*) operators when passing arrays to functions, they have an array identifier [].
- Arrays passed to functions are *always* passed by reference.

Passing “by value” or “by reference”

- One of the greatest benefits of pointers is the ability to pass values to function by reference.
- By default, when a value is passed “by value” to a function it is copied for the function to use.
- Depending on the storage size of the variable this may not be the most efficient use of memory.
- When passing by reference you pass a pointer into the function
- Here is an example for adding 5.

```
    void add5ByReference(int* pValue)
    {
        *pValue += 5;
    }

    int add5ByValue(int value)
    {
        return value + 5;
    }

    int main()
    {
        int i;
        cout << "Choose a number: ";
        cin >> i;
        cout << endl;

        cout << "Adding 5 by value" << endl;
        i = add5ByValue(i);
        cout << "i: " << i << endl;

        add5ByReference(&i);
        cout << "i: " << i << endl;
    }
```

const “read only” Qualifier

- Arguments can be passed to functions by values or by reference.
- When passing by value C makes a copy for the receiving function to use, also known as *information hiding*, this prevents direct changing of the incoming content, but creates additional overhead.
- Passing arguments by reference enables C programmers to modify content via pointers.
- There are times when you want the power and speed of passing by reference without the security risk of changing an argument’s content. This is accomplished by using the `const` qualifier. `const` declares a variable to be “read only,” when used in conjunction with pointers in a function you get a read only argument while still passing by reference.

```
void printArray(const int arr[], int size)
{
    for ( int i = 0; i < size; ++i )
    {
        cout << arr[i] << ' ';
    }
    cout << endl;

    // arr[0] = 88;  // read only variable is not assignable
}

void printArgument(const int* pValue)
{
    cout << "*pValue: " << *pValue << " is read only." << endl;

    // *pValue = 3;  // read only variable is not assignable
}

int main()
{
    int i = 5;

    printArgument(&i);

    // arrays are actually pointers
    int arr[] = {5, 4, 3, 2, 1};
    cout << " arr: " << arr << endl;
    cout << "*arr: " << *arr << endl;

    printArray(arr, 5);

    return 0;
}
```

swap() function using Pointers

- The output of this example is the same as the swap() function shown in the Reference Parameters notes.

```
void swap(int* pValue1, int* pValue2)
{
    int temp;
    temp = *pValue1;
    *pValue1 = *pValue2;
    *pValue2 = temp;
}

int main()
{
    int a = 4;
    int b = 7;

    cout << "BEFORE a:" << a << endl;
    cout << "BEFORE b:" << b << endl;

    swap(&a, &b);   // &a is address of a and &b is address of b

    cout << "AFTER a:" << a << endl;
    cout << "AFTER b:" << b << endl;

    return 0;
}
```

- But in this case, the address of variable is passed during function call rather than the variable itself.
  `swap(&a, &b); // &a is address of a and &b is address of b`
- Since the address is passed instead of value, dereference operator (`&`) must be used to access the value stored in that address.
  `void swap(int* pValue1, int* pValue2)`
- `*pValue1` and `*pValue2` gives the value stored at address `pValue1` and `pValue2` respectively.
- Since `pValue1` contains the address of `a`, anything done to `*pValue1` changes the value of `a` in `main()` function as well. Similarly, `b` will have same value as `*pValue2`.