

## Pointers and References

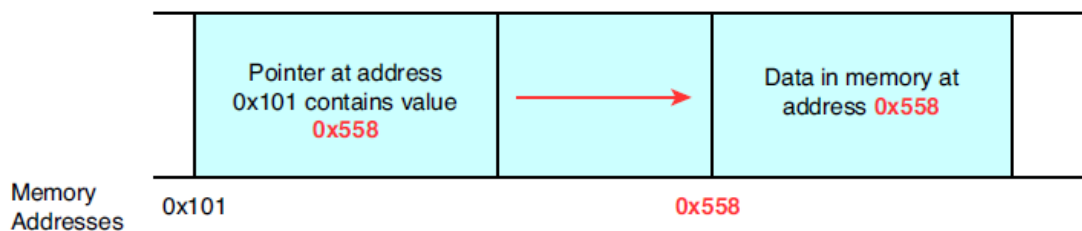
Understanding how pointers and references work is one step toward being able to write programs that are effective in their consumption of system resources.

In this lesson, you find out

- What pointers are
- What the free store is
- How to use operators `new` and `delete` to allocate and free memory
- How to write stable applications using pointers and dynamic allocation
- What references are
- Differences between pointers and references
- When to use a pointer and when to use references

### What Is a Pointer?

Put simply, a pointer is a variable that stores an address in memory. Just the same way as a variable of type `int` is used to contain an integer value, a pointer variable is one that is used to contain a memory address, as illustrated in the following Figure.



Thus, a pointer is a variable, and like all variables a pointer occupies space in memory (in the case of Figure, at address 0x101). What's special about pointers is that the value contained in a pointer (in this case, 0x558) is interpreted as a memory address. So, a pointer is a special variable that points to a location in memory.

### Declaring a Pointer

A pointer being a variable needs to be declared, too. You normally declare a pointer to point to a specific value type (for example, `int`). This would mean that the address contained in the pointer points to a location in the memory that holds an integer. You can also specify a pointer to a block of memory (also called a void pointer).

A pointer being a variable needs to be declared like all variables do:

```
PointedType * PointerVariableName;
```

As is the case with most variables, unless you initialize a pointer it will contain a random value. You don't want a random memory address to be accessed so you initialize a pointer to `NULL`. `NULL` is a value that can be checked against and one that cannot be a memory address:

## OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

*PointedType \* PointerVariableName = NULL; // initializing value*

Thus, declaring a pointer to an integer would be:

*int \*pInteger = NULL; //*

### Determining the Address of a Variable Using the Reference Operator (&)

Variables are tools the language provides for you to work with data in the memory. Pointers are variables, too, but they're a special type that is used exclusively to contain a memory address.

If *VarName* is a variable, *&VarName* gives the address in memory where its value is placed.

So, if you have declared an integer, using the syntax that you're quite well acquainted with, such as:

*int Age = 30;*

*&Age* would be the address in memory where the value (30) is placed. Example below demonstrates the concept of the memory address of an integer variable that is used to hold the value it contains.

Determining the Addresses of an <i>int</i> and a <i>double</i>
<pre>0: #include &lt;iostream&gt; 1: using namespace std; 2: 3: int main() 4: { 5:     int Age = 30; 6:     const double Pi = 3.1416; 7: 8:     // Use &amp; to find the address in memory 9:     cout &lt;&lt; "Integer Age is at: 0x" &lt;&lt; hex &lt;&lt; &amp;Age &lt;&lt; endl; 10:    cout &lt;&lt; "Double Pi is located at: 0x" &lt;&lt; hex &lt;&lt; &amp;Pi &lt;&lt; endl; 11: 12:    return 0; 13: }</pre>
<b>Output</b> Integer Age is at: 0x0045FE00 Double Pi is located at: 0x0045FDF8

### Using Pointers to Store Addresses

You have learned how to declare pointers and how to determine the address of a variable. You also know that pointers are variables that are used to hold memory addresses. It's time to connect these dots and use pointers to store the addresses obtained using the referencing operator (&).

Assume a variable declaration of the types you already know:

*// Declaring a variable*

*Type VariableName = InitialValue;*

# OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

To store the address of this variable in a pointer, you would declare a pointer to the same Type and initialize the pointer to the variable's address using the referencing operator (&):

```
// Declaring a pointer to the same type and initializing to address  
Type* Pointer = &Variable;
```

Thus, if you have declared an integer, using the syntax that you're quite well acquainted with, such as:

```
int Age = 30;
```

You would declare a pointer to the type *int* to hold the actual address where *Age* is stored, like this:

```
int* pInteger = &Age; // Pointer to integer Age
```

In the example below you see how a pointer can be used to store an address fetched using the referencing operator (&).

Demonstrating the Declaration and Initialization of a Pointer
<pre>0: #include &lt;iostream&gt; 1: using namespace std; 2: 3: int main() 4: { 5:     int Age = 30; 6:     int* pInteger = &amp;Age; // pointer to an <i>int</i>, initialized to <i>&amp;Age</i> 7: 8:     // Displaying the value of pointer 9:     cout &lt;&lt; "Integer Age is at: 0x" &lt;&lt; hex &lt;&lt; pInteger &lt;&lt; endl; 10: 11:     return 0; 12: }</pre>
<b>Output</b> Integer Age is at: 0x0045FE00

**Note:** Your output might differ in addresses from those you see in these samples. In fact, the address of a variable might change at every run of the application on the very same computer.

Now that you know how to store an address in a pointer variable, it is easy to imagine that the same pointer variable can be reassigned a different memory address and made to point to a different value, as shown in the example below.

Pointer Reassignment to Another Variable
<pre>0: #include &lt;iostream&gt; 1: using namespace std; 2:</pre>

## OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
3: int main()
4: {
5:     int Age = 30;
6:
7:     int* pInteger = &Age;
8:     cout << "pInteger points to Age now" << endl;
9:
10:    // Displaying the value of pointer
11:    cout << "pInteger = 0x" << hex << pInteger << endl;
12:
13:    int DogsAge = 9;
14:    pInteger = &DogsAge;
15:    cout << "pInteger points to DogsAge now" << endl;
16:
17:    cout << "pInteger = 0x" << hex << pInteger << endl;
18:
19:    return 0;
20: }
```

### Output:

```
pInteger points to Age now
pInteger = 0x002EFB34
pInteger points to DogsAge now
pInteger = 0x002EFB1C
```

### Access Pointed Data Using the Dereference Operator (\*)

You have a pointer to data, containing a valid address. How do you access that location—that is, get or set data at that location? The answer lies in using the dereferencing operator (\*). Essentially, if you have a valid pointer *pData*, use *\*pData* to access the value stored at the address contained in the pointer. Operator (\*) is demonstrated by example below.

#### Demonstrating the Use of the Dereference Operator (\*) to Access Integer Values

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     int Age = 30;
6:     int DogsAge = 9;
7:
8:     cout << "Integer Age = " << Age << endl;
9:     cout << "Integer DogsAge = " << DogsAge << endl;
10:
11:    int* pInteger = &Age;
12:    cout << "pInteger points to Age" << endl;
13:
14:    // Displaying the value of pointer
15:    cout << "pInteger = 0x" << hex << pInteger << endl;
16:
```

## OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
17: // Displaying the value at the pointed location
18: cout << "pInteger = " << dec << *pInteger << endl;
19:
20: pInteger = &DogsAge;
21: cout << "pInteger points to DogsAge now" << endl;
22:
23: cout << "pInteger = 0x" << hex << pInteger << endl;
24: cout << "pInteger = " << dec << *pInteger << endl;
25:
26: return 0;
27: }
```

### Output:

```
Integer Age = 30
Integer DogsAge = 9
pInteger points to Age
pInteger = 0x0025F788
*pInteger = 30
pInteger points to DogsAge now
pInteger = 0x0025F77C
*pInteger = 9
```

## Dynamic Memory Allocation

When you write a program containing an array declaration such as

```
int Numbers[100]; // a static array of 100 integers
```

your program has two problems:

1. You are actually limiting the capacity of your program as it cannot store more than 100 numbers.
2. You are reducing the performance of the system in cases where only 1 number needs to be stored, yet space has been reserved for 100.

These problems exist because the memory allocation in an array as declared earlier is static and fixed by the compiler.

To program an application that is able to optimally consume memory resources on the basis of the needs of the user, you need to use dynamic memory allocation. This enables you to allocate more when you need more memory and release memory that you have in excess. C++ supplies you two operators, *new* and *delete*, to help you better manage the memory consumption of your application. Pointers being variables that are used to contain memory addresses play a critical role in efficient dynamic memory allocation.

### Using Operators *new* and *delete* to Allocate and Release Memory Dynamically

You use *new* to allocate new memory blocks. The most frequently used form of *new* returns a pointer to the requested memory if successful or else throws an exception.

## OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

When using *new*, you need to specify the data type for which the memory is being allocated:

```
Type* Pointer = new Type; // request memory for one element
```

You can also specify the number of elements you want to allocate that memory for (when you need to allocate memory for more than one element):

```
Type* Pointer = new Type[NumElements]; // request memory for NumElements
```

Thus, if you need to allocate integers, you use the following syntax:

```
int* pNumber = new int; // get a pointer to an integer
```

```
int* pNumbers = new int[10]; // get a pointer to a block of 10 integers
```

Every allocation using *new* needs to be eventually released using an equal and opposite de-allocation via *delete*:

```
Type* Pointer = new Type;
```

```
delete Pointer; // release memory allocated above for one instance of Type
```

This rule also applies when you request memory for multiple elements:

```
Type* Pointer = new Type[NumElements];
```

```
delete[] Pointer; // release block allocated above
```

Note the usage of *delete[]* when you allocate a block using *new[...]* and *delete* when you allocate just an element using *new*.

If you don't release allocated memory after you stop using it, this memory remains reserved and allocated for your application. This in turn reduces the amount of system memory available for other applications to consume and possibly even makes the execution of your application slower. This is called a leak and should be avoided at all costs.

Example below demonstrates memory dynamic allocation and de-allocation.

### Accessing Memory Allocated Using new via Operator (\*) and Releasing It Using delete

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     // Request for memory space for an int
6:     int* pAge = new int;
7:
8:     // Use the allocated memory to store a number
9:     cout << "Enter your dog's age: ";
10:    cin >> *pAge;
11:
12:    // use indirection operator* to access value
13:    cout << "Age " << *pAge << " is stored at 0x" << hex << pAge << endl;
14:
15:    delete pAge; // release memory
16:
17:    return 0;
```

## OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
18: }
```

### Output:

```
Enter your dog's age: 9
Age 9 is stored at 0x00338120
```

**CAUTION:** Operator *delete* cannot be invoked on any address contained in a pointer, rather only those that have been returned by *new* and only those that have not already been released by a *delete*.

Thus, the pointers seen in the examples above contain valid addresses, yet should not be released using *delete* because the addresses were not returned by a call to *new*.

Note that when you allocate for a range of elements using *new...[...]*, you would de-allocate using *delete[]* as demonstrated by the example bellow.

### Allocating Using new[...] and Releasing It Using delete[]

```
0: #include <iostream>
1: #include <string>
2: using namespace std;
3:
4: int main()
5: {
6:     cout << "Enter your name: ";
7:     string Name;
8:     cin >> Name;
9:
10:    // Add 1 to reserve space for a terminating null
11:    int CharsToAllocate = Name.length() + 1;
12:
13:    // request for memory to hold copy of input
14:    char* CopyOfName = new char [CharsToAllocate];
15:
16:    // strcpy copies from a null-terminated string
17:    strcpy(CopyOfName, Name.c_str());
18:
19:    // Display the copied string
20:    cout << "Dynamically allocated buffer contains: " << CopyOfName << endl;
21:
22:    // Done using buffer? Delete
23:    delete[] CopyOfName;
24:
25:    return 0;
26: }
```

### Output:

```
Enter your name: Siddhartha
Dynamically allocated buffer contains: Siddhartha
```

## Passing Pointers to Functions

Pointers are an effective way to pass memory space that contains values and can contain the result to a function. When using a pointer with functions, it becomes important to ensure that the called function is only allowed to modify those parameters that you want to modify, but not others. For example, a function that calculates the Area of a Circle given *radius* sent as a pointer should not be allowed to

## OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

modify the *radius*. This is where you use *const* pointers effectively to control what a function is allowed to modify and what it isn't as demonstrated by the example.

Use *const* Keyword in Calculating the Area of a Circle When Radius and *Pi* Are Supplied as Pointers

```
0: #include <iostream>
1: using namespace std;
2:
3: void CalcArea(const double* const pPi, // const pointer to const data
4: const double* const pRadius, // i.e. nothing can be changed
5: double* const pArea) //change pointed value, not address
6: {
7:     // check pointers before using!
8:     if (pPi && pRadius && pArea)
9:         *pArea = (*pPi) * (*pRadius) * (*pRadius);
10: }
11:
12: int main()
13: {
14:     const double Pi = 3.1416;
15:
16:     cout << "Enter radius of circle: ";
17:     double Radius = 0;
18:     cin >> Radius;
19:
20:     double Area = 0;
21:     CalcArea (&Pi, &Radius, &Area);
22:
23:     cout << "Area is = " << Area << endl;
24:
25:     return 0;
26: }
```

**Output:**

Enter radius of circle: 10.5  
Area is = 346.361

### What Is a Reference?

A reference is an alias for a variable. When you declare a reference, you need to initialize it to a variable. Thus, the reference variable is just a different way to access the data stored in the variable being referenced.

You would declare a reference using the reference operator (&) as seen in the following statement:

*VarType Original = Value;*

*VarType& ReferenceVariable = Original;*

To further understand how to declare references and use them, see example bellow.

Demonstrating That References Are Aliases for the Assigned Value

```
0: #include <iostream>
1: using namespace std;
2:
```



## OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
3: int main()
4: {
5:     int Original = 30;
6:     cout << "Original = " << Original << endl;
7:     cout << "Original is at address: " << hex << &Original << endl;
8:
9:     int& Ref = Original;
10:    cout << "Ref is at address: " << hex << &Ref << endl;
11:
12:    int& Ref2 = Ref;
13:    cout << "Ref2 is at address: " << hex << &Ref2 << endl;
14:    cout << "Ref2 gets value, Ref2 = " << dec << Ref2 << endl;
15:
16:    return 0;
17: }
```

### Output:

```
Original = 30
Original is at address: 0044FB5C
Ref is at address: 0044FB5C
Ref2 is at address: 0044FB5C
Ref2 gets value, Ref2 = 30
```

### What Makes References Useful?

References enable you to work with the memory location they are initialized to. This makes references particularly useful when programming functions. A typical function is declared like this:

*ReturnType DoSomething(Type Parameter);*

Function *DoSomething()* is invoked like this:

*ReturnType Result = DoSomething(argument); // function call*

The preceding code would result in the argument being copied into *Parameter*, which is then used by the function *DoSomething()*. This copying step can be quite an overhead if the argument in question consumes a lot of memory. Similarly, when *DoSomething()* returns a value, it is copied again into *Result*. It would be ideal if we could avoid or eliminate the copy steps, enabling the function to work directly on the data in the caller's stack. References enable you to do just that.

A version of the function without the copy step looks like this:

*ReturnType DoSomething(Type& Parameter); // note the reference&*

This function would be invoked as the following:

*ReturnType Result = DoSomething(argument);*

As the argument is being passed by reference, *Parameter* is not a copy of argument rather an alias of the latter. Additionally, a function that accepts a parameter as a reference can optionally return values using reference parameters.

See example to understand how functions can use references instead of return values.

### Function That Calculates Square Returned in a Parameter by Reference

```
0: #include <iostream>
1: using namespace std;
2:
3: void ReturnSquare(int& Number)
4: {
```

## OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
5:   Number *= Number;
6: }
7:
8:
9: int main()
10: {
11:     cout << "Enter a number you wish to square: ";
12:     int Number = 0;
13:     cin >> Number;
14:
15:     ReturnSquare(Number);
16:     cout << "Square is: " << Number << endl;
17:
18:     return 0;
19: }
```

**Output:**

Enter a number you wish to square: 5  
Square is: 25

### Passing Arguments by Reference to Functions

One of the major advantages of references is that they allow a called function to work on parameters that have not been copied from the calling function, resulting in significant performance improvements. However, as the called function works using parameters directly on the stack of the calling function, it is often important to ensure that the called function cannot change the value of the variable at the caller's end. References that are defined as *const* help you do just that, as demonstrated by the example below. A *const* reference parameter cannot be used as an l-value, so any attempt at assigning to it causes a compilation failure.

#### Using const Reference to Ensure That the Calling Function Cannot Modify a Value Sent by Reference

```
0: #include <iostream>
1: using namespace std;
2:
3: void CalculateSquare(const int& Number, int& Result) // note "const"
4: {
5:     Result = Number*Number;
6: }
7:
8: int main()
9: {
10:     cout << "Enter a number you wish to square: ";
11:     int Number = 0;
12:     cin >> Number;
13:
14:     int Square = 0;
15:     CalculateSquare(Number, Square);
16:     cout << Number << "^2 = " << Square << endl;
17:
18:     return 0;
19: }
```

**Output:**

Enter a number you wish to square: 27  
27^2 = 729