CS 161: Pointers – Chapter 10

This week, you will learn about memory addresses and pointers. Pointers provide another option for passing objects by reference and they allow you to access memory allocated dynamically during the execution of your program. Dynamic memory allocation is essential for creating programs that adjust to different storage needs as the program executes.

As you continue to learn more about C++ and write increasingly more powerful programs, I doubt you will ever write a program that does not include pointers somewhere in the code.

Purpose of these notes

These notes provide an overview of the chapter 10 material. Please read the book to see complete code examples and more in depth discussions of the topics.

Notes about these notes

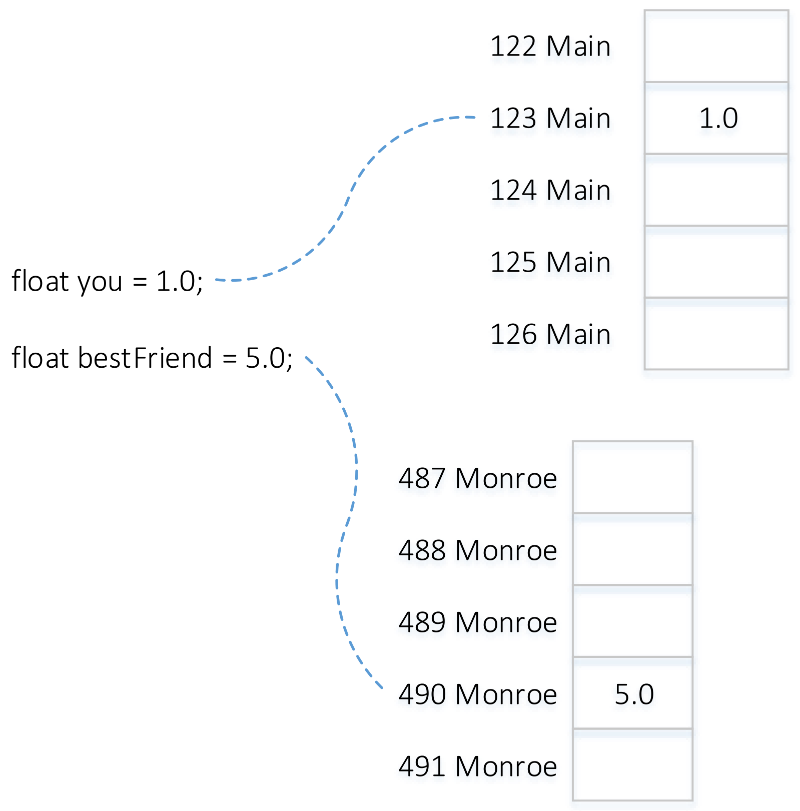
To keep things brief, some lines are left out of the code samples in these notes. The goal is to show what was necessary to illustrate the point. You will sometimes see "…" in the examples which means "some necessary code for a complete program are not shown here".

Additionally, the order of the topics in these notes follow the presentation order in the book for the most part but I did change the order of some sections to provide a better learning flow.

Contents

All things that you create can be referenced by an address

When you create variables such as a float, string, structure, or a class, the computer allocates or reserves a place in memory for the object. The computer tracks the locations by using memory addresses. So just like you live at 123 Main Street and your best friend lives at 490 Monroe Street, any variable you create lives at some address in the computer's memory.



How to see an object's memory address

You use the "&" operator (referred to as the address operator) to find or get to the address of a variable. An object's address is a number. Often, addresses are displayed as hexadecimal numbers such as 3E8 (decimal equivalent = 1000).

```
// Where do you live, bestFriend?

cout << “I live here: “ << &bestFriend << endl;
```

How to use the & operator

Use the & operator by prefacing (placing before) the variable name with the & symbol.

```
cout << &someVariable << endl;  // output address of someVariable
```

Programmer tip

As you learn C++, you are going find multiple cases where the same symbol such as "&" has different meaning based on how it is used within your code. For example, the "&" symbol is also used in function declarations to denote that a parameter is passed by reference. Take the time to really learn the language syntax so that when you see a symbol such as "&" in a line of code, you know what it will do based on the structure of that line of code.

How to think about addresses

There are lots of ways that you can conceptualize addresses.

- An address is the location in memory where the object/variable exists.
- An address is where your variable lives in the computer.
- An address is where your object is stored within the computer while your program is operating.

Pointer variables

So far, I have only shown the output of addresses through cout statements. You can create specific variables to store addresses. These are called pointer variables.

You declare a pointer variable by placing a "*" symbol between the variable type and the variable name. Jennifer in her videos referred to the "*" symbol as a splat symbol; I am going to call them asterisks in these notes.

```
double *myDoublePointer;
double* myDoublePointer2;
double * myDoublePointer3;
```

All of these placements of the * are valid. Any placement of the * between the type and the name are acceptable. However, most programmers either write

```
float* myVariable;
```

or

```
float *myVariable;
```

How to refer to pointer variables

If you created the following

```
string *myString;,
```

you could refer to myString as a pointer variable or simply as a pointer. It is most common to refer to pointer variables as pointers. So, in speech you would say that "myString is a pointer".

Pointer variables store addresses

A pointer variable **always** stores an address.

```
float myBestFriend = 1.0;  // address = 490 Monroe St

float *myBestFriendsAddress;   // create pointer to store myBestFriend's address

myBestFriendsAddress = &myBestFriend;  // assign address to pointer
```

Since pointers store addresses, we use the "&" operator to get the address of the myBestFriend variable.

Pointers point to

Since a pointer contains an objects address, we say that the pointer points to the object.

In the above example, after we execute the following line, we can now say that myBestFriendsAddress "points to" myBestFriend.

```
myBestFriendsAddress = &myBestFriend;  // assign address to pointer
```

The "binky" video (http://cslibrary.stanford.edu/104/) that Jennifer referred you to animates the concept of "pointing to". It is also common to see illustrations with lines ending in an arrow to illustrate "pointing to".

Never forget/Key point

Pointers are always used to store addresses. When writing your code and working with pointers, always ask yourself "am I assigning an address to my pointer variable?"

```
int x = 100;

int* xPointer;

xPointer = x;  // wrong, wrong → you must assign an address to the pointer

xPointer = &x;  // correct, correct→ pointers are always assigned addresses
```

Can I choose the address?

You as the programmer can never choose the address.

```
int *myPointer = 1000;  // never do this
```

Only the operating system interacting with your program can assign addresses.

Pointing to nothing

So, I just said that you should never assign a pointer to an address that you choose. However, you can assign a pointer to what is called the null address. The null address is the value of 0. The operating system will never assign any object created by your program the address of 0 so you can use the value of 0 to denote that my pointer contains no valid address.

```
        double *myDoublePointer = 0;
        // use 0 to denote that you have not assigned an address to myDoublePointer yet
```

The NULL keyword
Many header files including iostream, fstream, and cstdlib define the constant NULL to equal 0.

```
        #include <iostream>

        …

        double *myDoublePointer = NULL;  // set pointer to 0
```

Best practice recommendation
When declaring pointer variables, always initialize them to NULL unless assigning an address during the declaration statement.

```
        double myDouble = 1.23;
        double *myPointer = &myPointer; // assigning address during declaration
                                        // statement

        double *mySecondPointer = NULL; // no initial address assignment so use NULL
```

In terms of "lingo", you can say "mySecondPointer is null", or "mySecondPointer is a null pointer" or "mySecondPointer points to null".

What can I do with a pointer?
So if pointers always store addresses, how do you use a pointer in a program? The cool thing about a pointer is that you can use the indirection operator denoted by "*" (* is another example of multiple meanings for a single symbol) to access the actual object/thing located at the address.

When you use the indirection operator, you do what is called "dereferencing the pointer".

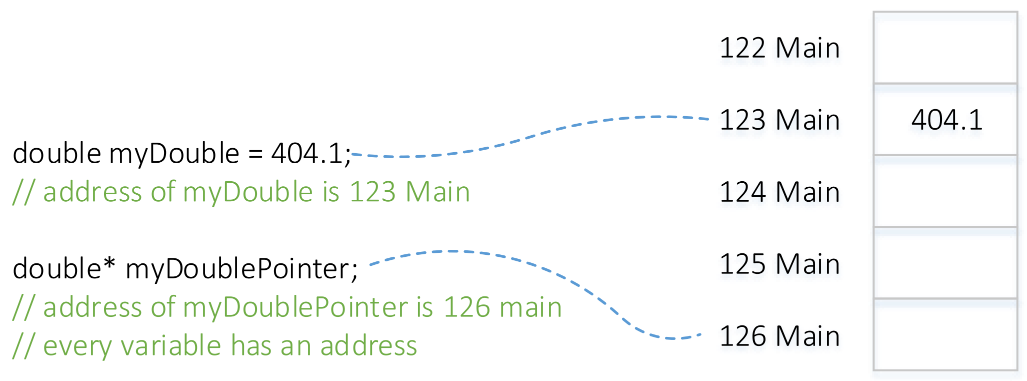
How to conceptualize dereferencing
I like to think of dereferencing as going to the object. If a pointer "points to" an object, dereferencing is "going to where the pointer points".

```
        double myDouble = 404.1;

        double *myDoublePointer; // asterisk used in variable declaration

        myDoublePointer = &myDouble;  // make myDoublePointer to myDouble
```

```
cout << *myDoublePointer << endl; // asterisk used as indirection symbol
// output 404.1
```

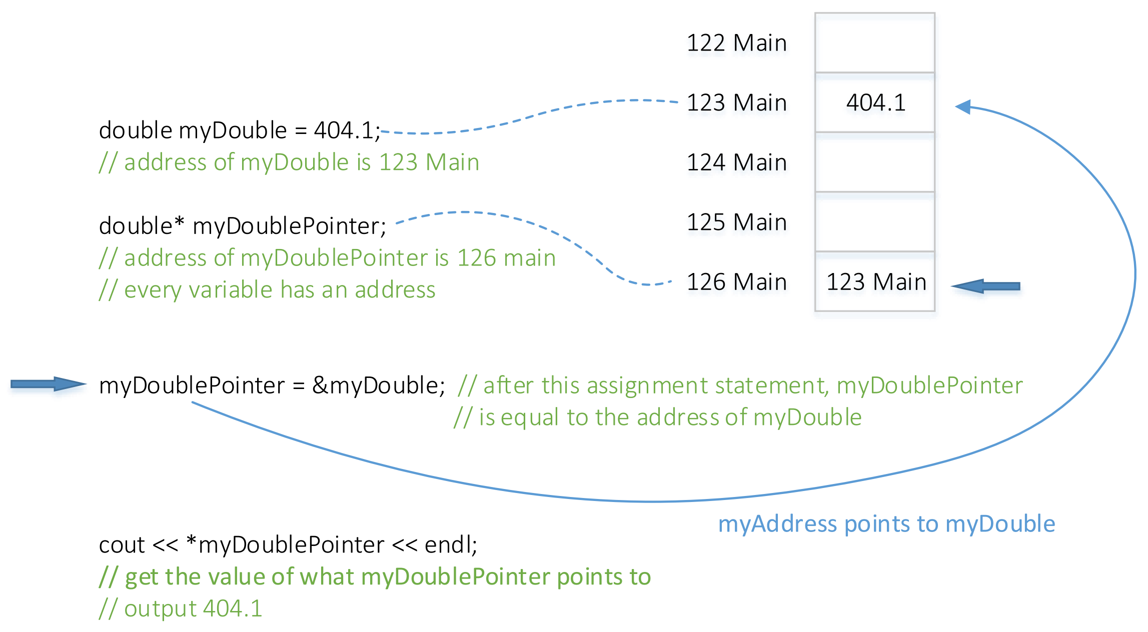
Since myDoublePointer "points to" myDouble, when I dereference myDoublePointer, I get the value of 404.1.

Below is the same code but showing an illustration of the computer memory. First, declare the two variables. Note that a pointer variable is a variable and just like every variable, it is allocated a memory address.



```
double myDouble = 404.1;
// address of myDouble is 123 Main

double* myDoublePointer;
// address of myDoublePointer is 126 main
// every variable has an address
```

Next, assign the address of myDouble to myDoublePointer and then use the indirection operator to deference myDoublePointer.



```
double myDouble = 404.1;
// address of myDouble is 123 Main

double* myDoublePointer;
// address of myDoublePointer is 126 main
// every variable has an address

myDoublePointer = &myDouble;  // after this assignment statement, myDoublePointer
                              // is equal to the address of myDouble

cout << *myDoublePointer << endl;
// get the value of what myDoublePointer points to
// output 404.1
```

Another way you can think of dereferencing is that when you write *myDoublePointer, * myDoublePointer becomes the object it is pointing to.

Working with dereferenced objects

When you deference a pointer, your program gets the actual object at the address, not a copy. Thus, modifying an object through the indirection operator is the same thing as modifying the object through the original variable.

Adding to the above example,

```
double myDouble = 404.1;
double *myDoublePointer; // asterisk used in variable declaration
myDoublePointer = &myDouble;  // make myDoublePointer to myDouble
cout << *myDoublePointer << endl; // asterisk used as indirection symbol
// output 404.1

/*********************************************************
*
* show how dereferencing accesses the same object as myDouble
*
*********************************************************/

// change value of dereferenced myDoublePointer
*myDoublePointer = 200.2;

cout << *myDoublePointer << endl; // *myDoublePointer
// output 200.2

cout << myDouble << endl; // the value of myDouble is also changed
// output 200.2
```

Pointers as function parameters

Declaring function parameters as pointers is another way to pass objects by reference. As discussed above, when you dereference the pointer, you are accessing the actual object that the pointer points and not a copy.

```
void doubleMyParameters(double *x, double *y)
// parameters are both double pointers
{
        *x = x*2;
        *y = y*2;
};

int main() {

        double x = 2;
        double y = 3;
        doubleMyParameters(&x, &y);  // pass addresses as arguments

        cout << x << ", " << y << endl;  // outputs 4, 9

        return 0;
    }
```

Dynamic memory allocation and the new operator

Static memory allocation is when you write a program and declare variables such as

```
int variableOne;
double variableTwo;
float variableThree;
```

This is considered static because the program is going to create these variables and reserve the same amount of space every time the program runs.

However, in most real world problems, it is extremely difficult to define all of the variable types and sizes up front when you write the program. For example, based on user input you might need an array of size 10 one time and the next time you might need an array of size 10000. In your pet lovers app, you sometimes might need to create cat objects and sometimes you might need to create dog objects. When your program needs to create variables on the fly, you need to use dynamic memory allocation.

The new operator

The "new" operator is used for dynamic memory allocation.

```
int *myDynamicIntPointer = new int;  // the new operator returns a pointer
```

When you use the new operator, the program goes out and reserves or allocates space for the new object you have requested.

Then, the new operator returns the memory address of the space it has allocated for the new object. **Thus, the left side of the assignment operator must always be a pointer variable.**

When you allocate memory using the new operator, the memory comes a section of memory called the heap. See Jennifer's videos for further discussion of the heap.

You can even allocate space for multiple objects at once such as

```
int* myDynamicIntPointer = new int[100]; // return pointer variable to 100 ints
```

You might say, "hey, that is starting to look like an array". In a latter section, we will show that arrays are the same as pointer variables.

Using the new operator with structures

Use of the new operator is not restricted to the fundamental C++ types. You can dynamically allocate structures.

```
structure Cat
{
        bool purrsAlot;
        bool hasLongHair;
}

cat* myCat = new Cat;
```

Dereferencing pointers to structures

There are two ways to dereference a pointer to a structure and access the structure variables

Use the * indirection operator

To access a member variable in a structure, we need to first apply the dereferencing operator and then use the "." operator. However, since the "." has a higher operator precedence than the * operator, we need to use parentheses around the dereferencing operator.

```
// continuing above example

bool doesTheCatPur = (*myCat).purrsAlot;
// dereference first, then use . operator
```

Quick review of selected operator precedence related to pointers

Operator precedence goes way beyond mathematical operators. Here is the precedence related to pointers, arrays, and memory allocation.

Precedence	Operator	Description	Associativity
1st	[]	Array subscripting	Left to right
	.	Element selection by reference	
	->	Element selection through pointer	
2nd	*	Indirection	Right to left
	&	Address of	
	new	Dynamic memory allocation	

**When you finish reading this entire document, read section 10.12 in the book and study table 10.1. It contains some great examples about selecting members of objects when pointers are involved.

Use the structure pointer operator

Since typing (*myPointer).structureVariable is annoying, the originators of C created the "->" operator. The "->" is called the structure pointer operator. With this you can write

```
bool doesTheCatPur = myCat->purrsAlot;  // using the -> shortcut
```

Just remember that you can only use the -> operator with structure pointers or class pointers (shown below).

Using the new operator with classes

Let's assume we have defined a class called Dog. In addition to the default constructor, there is a constructor that has two parameters

```
class Dog
{
        public:
                int weight;
                int age;

                Dog(int w, int a)
                {
                        weight = w;
                        age = a;
                }
};
```

To create new Dog objects, we can write

```
Dog* myDog = new Dog(); // default constructor call

Dog* myDog2 = new Dog(50, 6);  // call constructor with weight and age
```

The new operation might not always succeed

If there is not enough available free memory in the heap, a call to the new operator will fail. Specifically, in modern C++ distributions such as we use in the class, the program will throw a bad_alloc exception. You will learn about exceptions in later classes so for now, just remember that there is no guarantee that new can always allocate the memory you are requesting in your program.

Destroying or freeing dynamically created memory

When you allocate memory dynamically through the new operator, that memory is reserved until your program stops or you "free" the memory. The C++ delete operator is used to free memory in a running program. Free in this context means that the memory is returned to the unallocated part of the heap and other new operations in your own program or in other programs can allocate the memory.

Using the delete operator

To free dynamically allocated memory, call the delete operator.

```
        double* myDoublePtr = new double;

        // use myDouble and free when done

        delete myDoublePtr;

        myDoublePtr = NULL;
        // set to null so you know that myDoublePtr now does not point to any
        //allocated memory
```

If you use new to dynamically create an array, you need to place "[]" between delete and the pointer variable.

```
        double* myDynamicArray = new double[100];

        // use then free

        delete [] myDynamicArray;  // don't forget the []

        myDynamicArray = NULL;
```

Memory leaks

A memory leak is where you have finished using a block of memory allocated with new but you have failed to free it with the delete operator.

For example, if we created a function that when called dynamically allocated memory, used it, and then returned without freeing the memory, we would have a memory leak.

```
void myMemoryLeakfunction(int arraySize)
{
        int* myIntArray = new int[arraySize];

        // do a bunch a bunch of work

        // delete [] myIntArray;
        // uncomment the above line to prevent the memory leak
}
```

As stated above, all memory is freed when your program completes but many server programs never end so even small memory leaks add up and cause problems.

Best practices to avoid memory leaks

Follow these two rules to avoid memory leaks:

1. A function that invokes new to allocate storage should also invoke delete to free the storage.
2. A class that requires dynamic storage should call new in its constructors and call delete in its destructors.

Note that the book in Section 10.10 (p. 666) discusses returning pointers from functions. There are situations where it is reasonable to allocate memory with new in a function and to return the pointer to that memory from the function. In that case, the part of the program that calls the function should call delete at the appropriate time.

Dangling pointers

A dangling pointer is a pointer that points to a memory address that has been freed through invoking the delete operator.

Dangling pointers can cause errors that are hard to find because the dangling pointer might continue to work for some time because the freed memory has not been re-allocated to some other program or variable.

One tip to avoid dangling pointers is to always set a pointer to NULL after you call delete on the pointer.

```
delete myPointer;
myPointer = NULL;

// in your code
if(myPointer)  // true of myPointer != 0
{

        // pointer is good and you can use it
}
else
        // pointer is null
```

A second tip is to avoid have multiple pointers pointing to the same dynamically allocated memory. When you call delete on one of them, it will free the memory but your other pointers will still be pointing to the location.

Using dynamic memory allocation to adjust for the number of objects

When we talked about arrays, one of their drawbacks is that you could not define the size of the array during runtime. You had to do things like int myArray[100];

Now that we know the use of the new operator, we can write code so we allocate the exact amount of memory needed to store our data.

```
int numberOfBusRiders;

cout << “How many people are getting on the Bolt Bus? “ ;
// p.s. I am writing this example code while riding back from Seattle
cin >> numberOfBusRiders;

string* riderNames = new string[numberOfBusRiders];
// dynamic memory allocation using a variable to set size
```

Pointers and arrays

As hinted above, an array, specifically the name of an array variable, is a pointer.

```
int myTraditionalArray[100];  // declare an array of 100 ints

int* myPointer;

myPointer = myTraditionalArray;
// the name of an array variable is a pointer of the same type as used in the
// array declaration
```

When a pointer variable points to a sequence of objects, you can use the subscript operator with the pointer.

```
int testArray[]= (10, 20, 30, 40, 50};

int* testPointer = testArray;  // array variable without subscript is a int*

// use array variable
cout << myPointer[0]; << endl; // outputs 10

// use int* variable
cout << testPointer[0] << endl; // outputs 10;
```

If we use new to declare a sequence, we call also use subscripts.

```
int* myInts = new int[5];  // return pointer for 5 ints

myInts[0] = 100;
myInts[1] = 200;
```

Pointer arithmetic

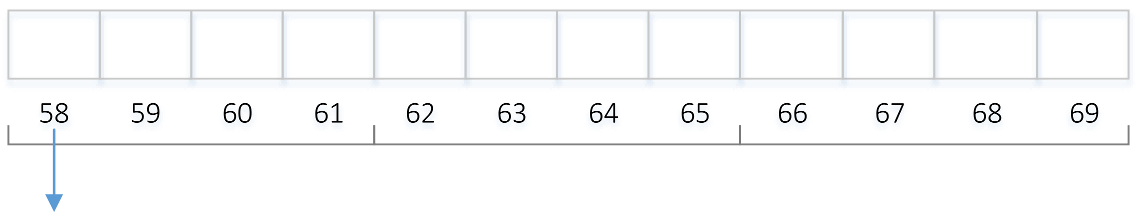
When you use the address operator or you use the new operator, the program returns the address of the first byte of the space allocated. Except for char and bool types, all other types and your own classes and structures require multiple bytes of memory for storage.

Let's consider a variable of type float. In most compilers, a float is a 4 byte long object.

If we execute the following:

```
float* myBigNumbers = new float(3);
```

This returns a pointer to a section of memory with space for 3 type float objects (12 total bytes).



Address (58) returned by new operator

The new operator returns the address of the first byte of this sequence with is 58 in our example.

Adding and subtracting from the pointer

When you add or substract from a pointer, you move the pointer forward or backwards by an amount equal to the size of type of the object that the pointer points to.

Using our myBigNumbers pointer from above, let's add to the pointer and see what happens. We will assume that the new operator returns 58

```
float* myBigNumbers = new float(3);

myBigNumbers [0] = 1.1;
myBigNumbers [1] = 2.2;
myBigNumbers [2] = 3.3;

cout << *myBigNumbers << endl;         // outputs 1.1
cout << long(myBigNumbers) << endl; // outputs 58 as the starting address

// add 1 to pointer
cout << *(myBigNumbers+1) << endl;  // outputs 2.2
cout << long(myBigNumbers+1) << endl; // outputs 62
```

When we added to the pointer, myBigNumber+1, the program did not add 1 but instead it added 4 because a float is of size of 4 bytes.

If our pointer pointed to a sequence of long double objects which are each 8 bytes and we incremented the pointer by one, we would change the value of the pointer by 8.

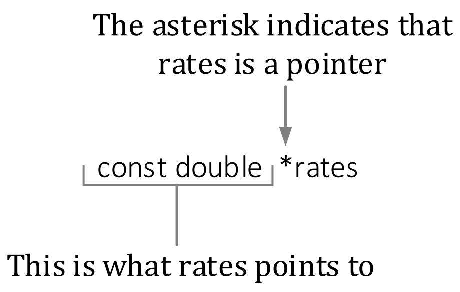
Constants and Pointers

In review, you use the const keyword when you want to define a variable or a value that cannot be changed.

Pointer to a constant

Define a pointer to a constant when you want the pointer to point to an object that cannot or should not be changed.

Syntax


The asterisk indicates that rates is a pointer
const double *rates
This is what rates points to

In words, "const double *rates" means that "rates is a pointer to a constant double" or "rates points to a constant double".

Note than you can place the asterisk next to the type (double above) or the variable name or between.

```
// these are all equivalent
const double* rates;
const double * rates;
const double *rates;
```

Assignment

Assign address of constant variable

The address of a constant variable must be assigned to a pointer to a constant.

```
const double firstRate = 18.0;
const double *rates;

rates = &firstRate;       // since firstRate is constant, double **must** be a
                          // pointer to a constant
```

Assign address of non-constant variable

It is perfectly okay to assign the address of a non-constant to a pointer to a constant

```
double secondRate = 22.0;  // non constant

const  double *rates = &secondRate;  // perfectly okay
```

Remember that you cannot use the indirection operator, *, or the structure pointer indirection operator, ->, on the left side of an assignment statement if the pointer is a pointer to a constant.

```
*rates =  50.0; // wrong, wrong, wrong because of rates is a pointer to a constant
```

Using pointers to constants as function parameters

Pointers to constants are most often used as function parameters. If your function has a pointer parameter and the function code will not modify the value that is pointed to, declare your function like this example.

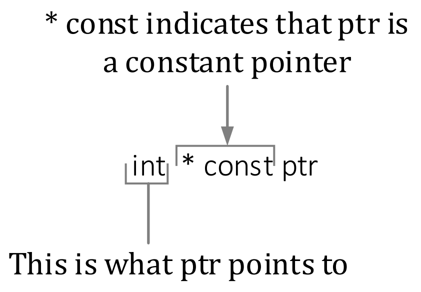
```
// function prototype
void DoSomeMagic(const double*currentRate);
// the function cannot modify what current rate points to

...

// in main

double interestRate = 18.0;
// note that there is no const definition but I am going to pass it to a function
// with a pointer to a const parameter

// call function
DoSomeMagic(&interestRate);
// since function parameter is a pointer, the argument must be an address so
//use the address operator
```

Proper use of const with pointers lets users of your code understand where your program cannot or will not modify the values that are pointed to.

Constant Pointers

A constant pointer is a pointer whose address value is constant. In other words, once assigned an address, it will always point to the same address. So if I assign my pointer the value of “1501 NW Monroe”, it will always point to “1501 NW Monroe” during the program’s runtime. However, unlike the previous pointers to constants, the value of the object at the address can be changed.

Syntax


* const indicates that ptr is a constant pointer
int * const ptr
This is what ptr points to

In words, "int * const ptr" means that "ptr is a constant pointer to an int"

Assignment

Constant pointers must be assigned an initial value when created.

```
Cat * const myFavoriteCat = new Cat();  // reusing Cat object from earlier
                                        // example

myFavoriteCat->purrsAlot = true;  // okay to change value of what the
                                  // pointer points to
```

When used as function parameters

```
void VetCheck(Cat * const cats);  // function prototype with constant pointer

        .. // in main

Cat *herdOfCats = new Cat[50];  // create a pointer to a sequence of 50
                                // cat objects

VetCheck(herdOfCats);  // call function and pass herdOfCats which is the starting
                       // address of my 50 Cat sequence
```

Since the VetCheck function parameter "cats" is a constant pointer, I know that the VetCheck function will not change what "cats" points to within the function body.

Practical Tip

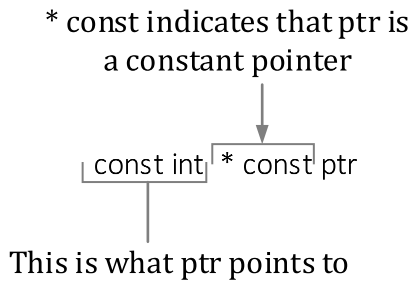
It is usually more beneficial to define a function with pointer to constant parameters than with pointer constant parameters. A pointer to a constant prevents you from changing values that could be used outside of the function while a constant pointer only prevents you from changing the pointers address within the function body.

This is a little bit advanced for now, so this is just for introduction sake, but if you want to change what a pointer points to within a function so that it affects the calling code, you need to pass a pointer to a pointer to the function. Don't get hung up on this now but I don't see much use of pointer constants in functions.

Constant pointers to constants

Constant pointers to constants make the "whole deal" constant. You can make the address in the pointer and what it points to constant as follows:

Syntax


* const indicates that ptr is a constant pointer
const int * const ptr
This is what ptr points to

With this type of declaration, ptr would always point to the address assigned to it when declared and you could not change the value of the object at the pointed to address.