WHAT IS A POINTER?

Pointers are among the most consistently confusing topics for new programmers. Even so, they don't have to be! Once you get some practice you'll see just how uncomplicated they really are and can work on implementing them into your own programs.

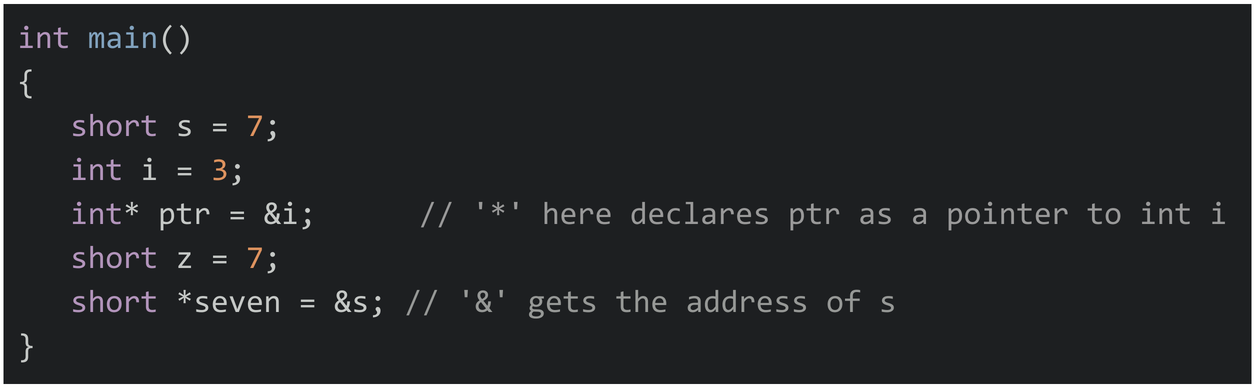
> **Pointers:** are objects whose values are the *addresses* of another object in memory.

HOW ARE THEY USED?

Pointers can be created using the following syntax:

- **(Type) (Star) (Variable Name) = &(Thing Pointing To)**
- `int* ptr = &x;`
- `int *p = &y;`

The "address-of" operator, '&,' gets the address of the variable.
Note that the star can be anywhere between the type and variable name.



```
int main()
{
   short s = 7;
   int i = 3;
   int* ptr = &i;      // '*' here declares ptr as a pointer to int i
   short z = 7;
   short *seven = &s; // '&' gets the address of s
}
```

0x112	0x116	0x120	0x124	0x128
`short s = 7`	`int i = 3`	`int* ptr = 0x116`	`short z = 7`	`short* seven = 0x112`

*This worksheet uses a simplified representation of memory storage for the sake of learning pointers only

TRY IT OUT: POINTER DECLARATION & USE

Declare and initialize an object, `num`, of type `double` and a pointer, `ptr`, to it. Try to use the pointer to:

- Print out the memory address of your double
- Print out the value of your double
- Modify the value of your double

Worksheet by Dylan Gunn, UCLA 2020
Feedback Welcome: dylankgunn@gmail.com

THE NULLPTR

The `nullptr` is a special pointer that does NOT point to an object. Declaration of a pointer type but *failure to initialize* it gives the pointer a garbage value and dereferencing that pointer leads to undefined behavior.

```
int* ptr = nullptr;
```

When we declare a pointer as `nullptr`, we can make a check that our value has not been declared as something else. The `nullptr` may be used to signify the end of something like a list; while traversing down the list, by doing a check for

```
ptr->next == nullptr
```

we can ensure that we know when we've reached the end, assuming we've already set the last value in a list to refer to the `nullptr` if the `next` value is accessed.

Example:

```
struct Node
{
      int val;
      Node* next;     // points to the next node in the list
}

int main()
{
      Node one, two, three;
      one.val = 1;
      one.next = &two;
      two.val = 2;
      two.next = nullptr;

      Node* ptr = &one;
      while (ptr->next != nullptr)
      {
            cout << ptr->val << " ";
            ptr = ptr->next;
      }

      return 0;
}
```

POINTERS & ARRAYS

Arrays and pointers may be similar, but they are NOT the same. However, what does an array *parameter* refer to? When the name of an array is used in expressions, *it is automatically converted to a pointer to the first element of the array*. This means that there is a nearly seamless system of use between arrays and pointers, but remember that arrays are *NOT* pointers!

```
int main()
{
      int arr[] = {4, 3, 2};
      int* ptr1 = arr;
      int *ptr2 = ptr1 + 1;

      *ptr1 = 5;  // '*' here gets the value of (dereferences) arr[0]
      *(ptr2+1) = 6;
}
```

0x123	0x127	0x131	0x135	0x139	0x143
`ptr1 = 0x131`	`-193256328`	`arr[0] = 5`	`arr[1] = 3`	`arr[2] = 6`	`ptr2 = 0x135`

POINTERS TO ARRAYS

Pointers can also be pointed to arrays and used similarly:

```
int arr[5] = {6, 7, 8, 9, 10};
int* ptr = arr;

cout << *(ptr + 2) << endl;  // both print the value '8'
cout << ptr[2] << endl;
```

TRY IT OUT: POINTERS & ARRAYS

Practice these to solidify your pointer-array skills:

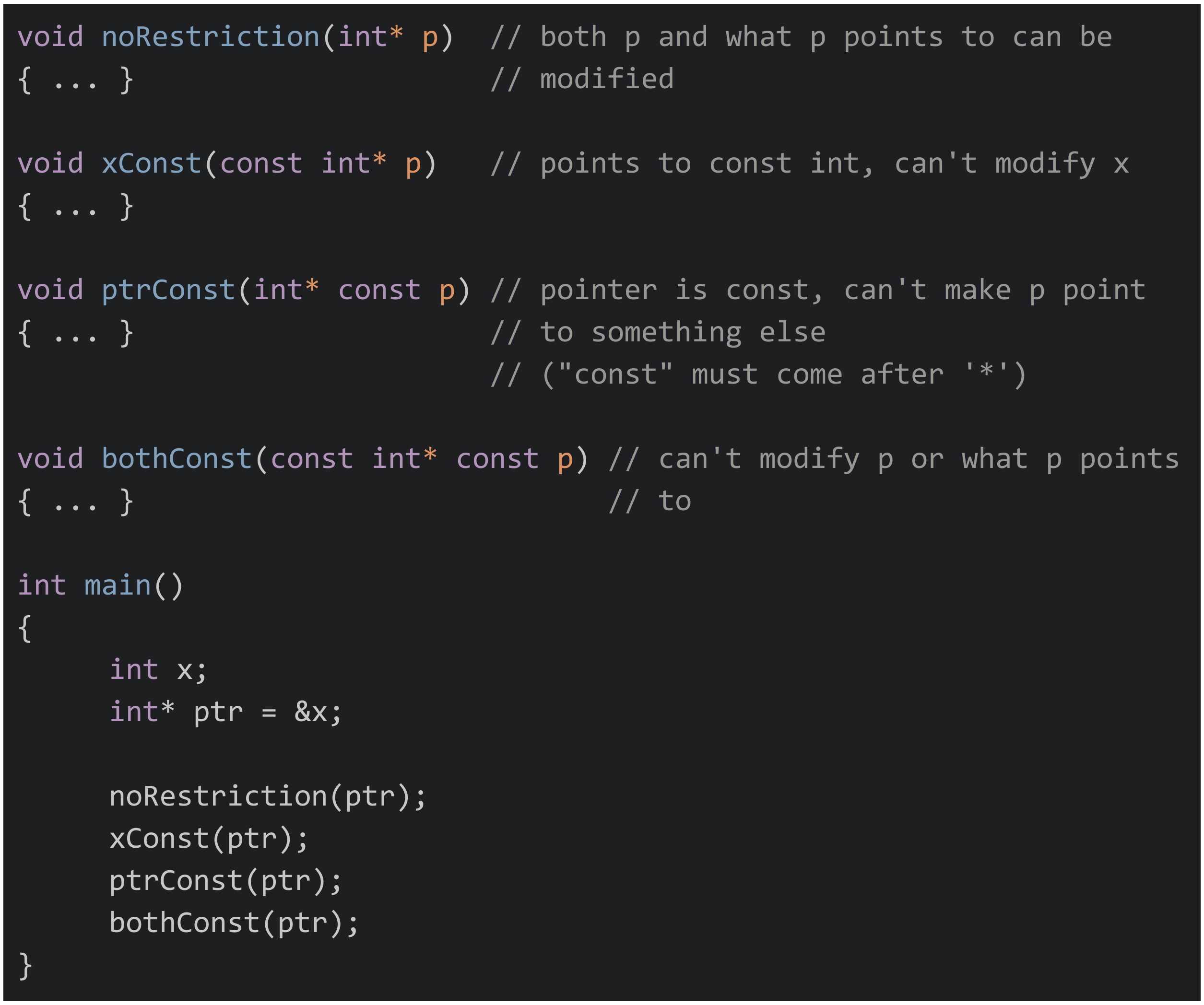
- Pass an integer array and its size to a `void` function `foo` and use a pointer to access every element of the array.
- Try using various pointer operations (*, +, ++, --, +=, etc) with an array variable. What works? What doesn't?

POINTERS & CONST

Many times, we will have to pass pointers to functions with restrictions. Just as we can define const variable types, we can do the same with pointers! However, as a pointer consists of two things (the pointer itself and what is being pointed to), there is some special syntax.

Const Possibilities:

- **(Type) (Star) (Variable Name)**
  - No restriction on modification
- **Const (Type) (Star) (Variable Name)**
  - Pointer to type const: Can't modify value of thing pointed to
- **(Type) (Star) Const (Variable Name)**
  - Pointer is const: Can't change what object is pointed to
- **Const (Type) (Star) Const (Variable Name):**
  - Can't modify pointer or value of thing pointed to



```
void noRestriction(int* p)  // both p and what p points to can be
{ ... }                     // modified

void xConst(const int* p)   // points to const int, can't modify x
{ ... }

void ptrConst(int* const p) // pointer is const, can't make p point
{ ... }                     // to something else
                            // ("const" must come after '*')

void bothConst(const int* const p) // can't modify p or what p points
{ ... }                            // to

int main()
{
     int x;
     int* ptr = &x;

     noRestriction(ptr);
     xConst(ptr);
     ptrConst(ptr);
     bothConst(ptr);
}
```

POINTERS TO POINTERS

Eventually, we must deal with situations involving pointers to pointers.
This can get a little confusing, so it's best to practice as much as possible!
Even so, the syntax is the same as everything we've done so far:

- **(Type) (Star) (Star) (Variable Name)**
- `(int*)* ptrsquared;`
- `double** doubledouble;`

TRY IT OUT: POINTERS TO POINTERS

What would the following program print to standard output?

```
int main()
{
     int a;
     int* b = &a;
     // int** c = &a; would fail to compile
     int** c = &b;
     int d = 100;

     *b = 5;
     **c = 10;

     cout << *b << endl;
     cout << *c << endl;

     b = &d;
     (*b)++;

     cout << **c << endl;
}
```

- Necessary Information:
    - `&a = 0x10, &d = 0x20`
    - `&b = 0x100`
    - `&c = 0x1000`

SOLUTIONS

POINTER DECLARATION & USE

Declare and initialize an object, `num`, of type `double` and a pointer, `ptr`, to it.

```
double num = 3.14;
double* ptr = &num;
```

Try to use the pointer to:

- Print out the memory address of your double
  - `cout << ptr;`
- Print out the value of your double
  - `cout << *ptr;`
- Modify the value of your double
  - `*ptr = 3.1415926;`

POINTERS & ARRAYS

Pass an integer array and its size to a `void` function `foo` and use a pointer to access every element of the array.

```
void foo(int arr[], int size)
{
      int* ptr = arr;
      for(int i = 0; i < size; i++)
            cout << arr[i] << “ “;

      return;
}

int main()
{
      int a[5] = {1, 2, 3, 4, 5};
      foo (a, 5);

      return 0;
}
```

POINTERS TO POINTERS

What would the following program print to standard output?

```
int main()
{
	int a;
	int* b = &a;
	/* int** c = &a; would fail to compile */
	int** c = &b;
	int d = 100;

	*b = 5;
	**c = 10;

	cout << *b << endl;
	cout << *c << endl;

	b = &d;
	(*b)++;

	cout << **c << endl;
}
```

- Necessary Information:
    - `&a = 0x10, &d = 0x20`
    - `&b = 0x100`
    - `&c = 0x1000`

```
10
0x100
101
```