```
//===========================
// Section 12 Pointers and references
//===========================

// What is a pointer?
A variable whose value is an address

What can be at that address?
- Another variable
- a function

Pointers can point to variables or functions
If x is an integer variable and its value is 10, then I can declare a pointer that points to it
To use the data that the pointer is pointing to you need to know its type

Why use pointers?
Pointers can be used to access data outside the function
Those variables might not be in scope so you can't reference them by name
Pointers can be used to operate on arrays efficiently
We can allocate memory dynamically on the heap or the free store
- This memory doesn't even have a variable name
- We can only access it by the pointer, so if we lose the pointer we're fucked

With Object Oriented programming this is how polymorphism works
We can access specific addresses in memory - useful in embedded\systems applications

// Declaring pointers - these will all have garbage data in them
variableType * pointerName;
int * integerPointer;
double * doublePointer;
char * charPointer;
string * stringPointer;

// Initializing pointers - nullptr will tell it to point 'nowhere'
variableType * pointerName{nullptr};

int * intergerPointer{};
double * doublePointer{nullptr};

ALWAYS initialize pointers
nullptr represents 'address 0' which also means nowhere (C++11)
If you don't initialize a pointer to a variable or a function then use nullptr

// Accessing the pointer address and storing the address in a pointer
The address of operator

- Variables are stored in unique addresses
- unary operator
- Evaluates to the address of its operand

Example:
int num{0};

cout << num << endl;                // 10
cout << sizeof num << endl;         // 4 bytes
cout << &num << endl;               // Hexidecimal address of num

sizeof a pointer variable
Don't confuse the size of a pointer and the size of the data
all pointers have the same size - a single memory address
They may be pointing to very small or very large types

Typed pointers
- The compiler will make sure the pointer is pointing to the correct type

Pointers are variables so they can change
They can be null
They can be uninitialized (bad idea);
```

```
// Dereferencing a pointer
Access the data a pointer is pointing to
Access it using the overloaded * indirection operator

int score{100};
int * scorePointer{&score};
cout << *scorePointer << endl; // follow the pointer and display the value

*scorePointer = 200; // Follow the pointer and change the value

NOTE: The dot operator has higher precedence than the dereference operator
To get to data in a vector you must do this:
(*vectorPointer).at(0)

// Dynamic memory allocation
This is allocating storage from the heap at runtime
- We often don't know how much we need until the user tells us
- We can allocate storage for a variable at run time
- We can use pointers to access the stuff we just created

We use the new keyword to allocate storage at run time
int * intPointer{nullptr};      // Create a new pointer to an integer and make it null
intPointer = new int;           // Tell the pointer it now points at a new integer on the heap
NOTE: if you lose the pointer, you lose the only way to get to that data

// Using delete to deallocate
delete intPointer;

Make sure you do this so you don't have a memory leak

// Allocting an entire array
int * arrayPointer{nullptr};
int size{};

cout << "How big do you want the array?";
cin >> size;
arrayPointer = new int[size]; // Allocates this array up on the heap

delete [] arrayPointer; // put the empty brackets between the delete keyword and the arrayPointer name

// The relationship between arrays and pointers
The value of an array name is already the address of the first element in the array
The value of the pointer varialbe is an address
If the pointer points to the same data type as the array element then the pointer and array name can be used interchangeably (almost)

You can dereference the array's variable name like you would a pointer
We can use array subscripting on pointers

int scores[]{98,100,95};
cout << scores << endl;         // hexidecimal address
cout << *scores << endl;        // the value in the first location

int * scorePointer{scores};     // Lets point to it
cout << scorePointer << endl;   // Hexidecimal address
cout << *scorePointer << endl;  // The value in the first location

// Array subscripting works
cout << scorePointer[0] << endl;    // 98
cout << scorePointer[1] << endl;    // 100
cout << scorePointer[2] << endl;    // 95
```

```
// Using in an expression to move around the array (pointer arithmetic)
cout << scorePointer << endl;           // Hexidecimal address
cout << (scorePointer + 1) << endl;     // Adds the size of the data type to the hexidecimal address to get to the next loc
cout << (scorePointer + 2) << endl;     // Same here - this would be the memloc of the third item in our array

// Dereferernce it to see the data itself
cout << *scorePointer << endl;              // 98
cout << *(scorePointer + 1) << endl;        // Do the pointer arithmetic then dereference the location - 100
cout << *(scorePointer + 2) << endl;        // Same as above - 95

Subscript and offset notation equivelence
int arrayName[]{1,2,3,4,5};
int * pointerName{arrayName};

Subscript notation               Offset notation
-------------------              ----------------
arrayName[index]                 *(arrayName + index)
pointerName[index]               *(pointerName + index)

// Pointer Arithmetic
Pointers can be used in:
- Assignment expression
- Arithmetic expressions
- Comparison expressins
 C++ allows pointer arithmetic, but it only makes sense with raw arrays

 (++) increments to the next array element
 intPointer++;
(--) decrements a pointer to the previous array element
intPointer--;

(+) increment by n* sizeof(type)
intPointer += or intPointer = intPointer + n
(-) decrement by n* sizeof(type)
intPointer -= or intPointer = intPointer - n


Subtracting two pointers
Determines the number of elements between the two pointers
NOTE: Both pointers must point to the same data type or you get an error
int n = intPointer1 = intPointer2;

// Comparing pointers - this will not compare the data in the pointers, just the addresses
string s1{"Babs"};
string s2{"Babs"};
string * p1 {&s1};
string * p2 {&s2};
string * p3 {&s1};

cout << (p1 == p2) << endl; // false
cout << (p1 == p3) << endl; // true

// Compare the data inside by dereferencing
cout << (*p1 == *p2) << endl;   // false
cout << (*p1 == *p3) << endl; // true

// Example (with a sentinel value)
int scores[]{100,95,89,68,-1};
// The -1 is a sentinel value - we can loop and stop when we see that
int * scoresPointer{scores};
// We do not need the & when we do this with an array which is already an address
```

```
while(*scorePointer != -1){
    cout << *scorePointer << endl;
    scorePointer++; // Will add 1 'element' to the address, in this case it's an int so probably 4 bytes
}

// condense the while loop above
while(*scorePointer != -1){
    cout << *scorePointer++ endl;   // Precedence and associativity is important here
}                                   // We dereference the pointer, use it, and then increment it

// Const and pointers
Several ways to use it
- Pointers to constants
- Constant pointers
- Constant pointers to constants

The data pointed to cannot be changed
The pointer itself can change and point somewhere else

int highScore{100};
int lowScore{65};
const int * scorePointer{&highScore};   // We cannot change this data
*scorePointer = 86;                     // This will throw an error
scorePointer = &lowScore;               // We can change where it points, however. And now it can't make changes to low score

// Constant pointers cannot change and point to other data
int * const scorePointer(&highScore);       // This one can only point to highScore
*scorePointer = 86;         // Perfectly legal
scorePointer = &lowScore;   // Uh oh. We can't do that.

// Constant pointer to a constant
const int * const scorePointer{&highScore}; // Now we can't change shit
*scorePointer = 86;         // error
scorePointer = &lowScore;   // another error

// Passing pointers into functions
We can use pointers and the dereference operator to achieve pass by reference
The function parameter is a pointer
The actual parameter can be a pointer or the address of a variable

void doubleData(int * pointerTpInteger); // It's expecting a pointer (address) so we can send in an actual pointer or use the & address of operator on a variable

void doubleData(int * pointerToInteger){
    *poinerToInteger *= 2; // We don't need return since we are working with the data directly
}

// How to call the function
int main(){
    int value{20};
    cout << value << endl;  // Value is 20 right now
    doubleData(&value);     // In this case we send in an address using the address of operator
    cout << value << endl;  // Value is now 40 since we directly messed with the variable
}
```

```
// Swapping data using pointers
void swap(int * a, int * b){
    int temp = *a;  // Set the dereferenced value of a into temp
    *a = *b;        // Set the deferenenced value of b into the dereferenced value of a
    *b = temp;      // Set the value of temp into the dereferenced value of b
}

main(){
    int x{100};
    int y{200};

    cout << x << " " << y << endl;
    swap(&x, &y);   // Swap the values at the addresses x and y live at
    cout << x << " " << y << endl;
}

// Using actual pointers
#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <ctime>

void display(const std::vector<std::string>* const v) { // Can't change the vector, and can't change the pointer
    for (auto str : * v) {
        std::cout << str << " ";
    }
    std::cout << std::endl;
}

void display(int* array, signed int sentinel) { // overloaded, we can't use const here because we are updating the pointer
    while (*array != sentinel) {
        std::cout << *array++ << " "; // Don't forget to increment here - I ended up in an endless loop because I missed the ++
    }
    std::cout << std::endl;
}

int main(){

    std::cout << "--------------" << std::endl;
    std::vector<std::string> cats{ "Babs", "Yaya", "Calico", "Rory" };
    display(&cats);

    int scores[]{ 100, 98, 67, 89, 40, -1};
    signed int sentinelValue = -1;
    display(scores, sentinelValue); // Since this is an array we can just toss it in

    return 0;
}

// Returning a pointer from a function
Functions can also return functions
type * function();

Should return pointers to:
- Memory dynamically allocated in the function
- To data that was passed in
NEVER RETURN A POINTER TO A LOCAL FUNCTION VARIABLE

int* largestInt(int* firstInt, int* secondInt) {
    if (*firstInt > * secondInt) {
        return firstInt;
    } else {
        return secondInt;
    }
}
```

```cpp
// Calling that function
int main(){

    int a{ 100 };
    int b{ 200 };
    int* largestPointer{ nullptr };
    largestPointer = largestInt(&a, &b);
    std::cout << *largestPointer << std::endl;
    return 0;
}

// Returning dynamically allocated memory
int* createNewArray(size_t size, int initialValue = 0) {
    int* newStorage{ nullptr }; // Set up a new pointer
    newStorage = new int[size];
    for (size_t i{ 0 }; i < size; ++i) {
        *(newStorage + i) = initialValue; // Add one full element to newStorage and dereference
    }
    return newStorage;
}

// Calling that functions
int main(){
    int* myArray;   // To be allocated by the function
    myArray = createNewArray(100, 20); // Create the array
    // use it

    delete[] myArray; // Be sure to free it back up again
}

DO NOT DO THIS:
int * dontDoThis(){
    int size{};
    return &size; // This is a local variable - do not ever return these
}

int *orThis(){
    int size{};
    int *intPointer{&size};
    return intPointer; // This is also a terrible idea
}

If you do this, the program might work well for a while and then all of a sudden crash
These types of errors are really hard to find

// Potential pointer pitfalls
Uninitialized pointers - can point anywhere
int * intPointer; // Who knows what is in it

Dangling pointers
- Pointing to released memory
- 2 pointers pointing to the same data
- One of the pointers already released it
- Other pointer tries to get to it and crashes

Pointer that points to memory that is invalid
- Like when we return a pointer to a local variable which has gone out of scope

Not checking if new failed
- if new fails an exception is thrown
- We need to use exception handling to deal with this
- Dereferencing a null pointer will cause the program to crash

Memory leak
- Forgetting to release allocated memory with delete
- If you lose the pointer you can't get to it again
- The memory is orphaned or 'leaked'
- One of the most common pointer problems
```

```
// What is a reference
An alias for a variable
Must be initialized when a variable is declared
Cannot be null
Once initialized, cannot be made to refer to something else
Useful as function paramters
Kind of like a constant pointer that is automatically dereferenced

// Using references in a range based loop
std::vector<string> cats {"Yaya", "Babs", "Calico", "Rory"};

for(auto &str: cats){
    str = "Kittens"; // This will change the actual data
}

// Can change to const to prevent changes
for(auto const &str: cats){
    str = "Kittens"; // The compiler will bark
}

// L-Values and R-Values
L-Values have names are are addressable
We can modify them if they are not constants
Generally appears on the left hand side of the assignment statement
Literals are not L-Values

int x{100}; // x is an L-Value
string name{"Josie"}; // name is an L-Value

R-Values can be defined by exclusion: Anything that is not an L-Value is an R-Value
On the right hand side of the assignment expression
A literal
A temporary value that we don't intend to modify

L-Values can appear on both sides of an assignment statement
int x{100};
int y{};

y=100;          // R-Value 100 assigned to L-Value Y
x = x + y;      // R-Value x + y assigned to L-Value x

References
So far we have used all references to L-Values (&source);
int x{100};

int &ref1 = x;          // This is okay, since x is an L-Value we can reference
ref1 = 1000;

int &ref2 = 100;        // This is an error - 100 is a literal not an L-Value and we can't reference a literal

// Same is true when we pass by reference
int square(int &n){     // This function wants a reference to a number
    return n*n;
}

int num{100};
square(num);            // We can do a reference to an L-Value so this works
square(5);              // We cannot do a reference to an R-Value literal so this fails

// When to use which?
Use Pass-by-value when:
- The function does not modify the parameter
- the parameter is small and efficient to copy - char, int, etc

- Use pass-by-reference using a pointer when:
- The function modifies the parameter
- The parameter is expensive to copy
- It's okay for the pointer to contain a null value
```

```
- Use pass-by-reference using a pointer to const
- When the function does NOT modify the parameter
- The parameter is expensive to copy
- It's ok for the pointer to contain a null value

- Use pass-by-reference using a const pointer to const
- When the function does NOT modify the parameter
- The parameter is expensive to copy
- It's ok for the pointer to contain a null value
- You don't want to modify the pointer itself

Pass-by-reference using a reference
- When the function DOES modify the parameter
- The parameter is expensive to copy
- The parameter will never be nullptr

Pass-by-reference using a const reference
- When the function does NOT modify the parameter
- The parameter is expensive to copy
- The parameter will never be nullptr

//======================
// Section 12 Challenge
//======================
```

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <ctime>

// Write a print function that expects a pointer to an array of integers and display the elements in the array
void print(const int * const arrayName, int arraySize) {
    for (size_t i{ 0 }; i < arraySize; ++i) {
        std::cout << arrayName[i] << std::endl;
    }
}

// Write a function that expects two arrays of integers and their sizes - we can use pointers since arrays are already addresses
int* applyAll(const int * const firstArray, size_t firstArraySize, const int * const secondArray, size_t secondArraySize) {

    size_t sizeOfNewArray{ firstArraySize * secondArraySize };
    int * combinedArray{ nullptr }; // Create our pointer to our new storage
    combinedArray = new int[sizeOfNewArray]; // Allocate this on the heap
    int counter{ 0 };
    // Loop over the second array and multiply each element of array 1 and store the product in the new array
    for (size_t i{ 0 }; i < secondArraySize; ++i) {
        for (size_t j{ 0 }; j < firstArraySize; ++j) {
            *(combinedArray + (counter++)) = firstArray[j] * secondArray[i];
        }
    }
    return combinedArray;
}

int main(){
    const size_t firstArraySize{ 5 };
    const size_t secondArraySize{ 3 };
    int firstArray[]{1,2,3,4,5};
    int secondArray[]{10,20,30};

    std::cout << "array 1: " << std::endl;
    print(firstArray, firstArraySize);

    std::cout << "array 2: " << std::endl;
    print(secondArray, secondArraySize);

    int* results = applyAll(firstArray, firstArraySize, secondArray, secondArraySize);
    constexpr size_t resultsSize{ firstArraySize * secondArraySize };

    std::cout << "Results: " << std::endl;
    print(results, resultsSize);
    std::cout << std::endl;

    delete [] results;

    return 0;
}
```