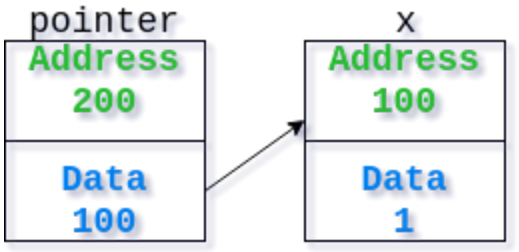
Pointers



A pointer contains the address of a variable. For example, if **int x = 1;** then the value of **x** is being stored in some location in memory. Let's say this memory location is at **address 100**. If you have a **pointer** pointing to **x**, that is another location in memory. Now, the data being stored in the pointer will be the **address of x**. Since the address of **x** is **100**, the pointer contains the value **100**. Since a pointer is also a location in memory, it will have its own address. For now, focus on the data portion of the pointer. To declare and initialize a pointer to an **int**:

```
int x = 1; //Normal variable
int* pointer; //* indicates this variable is a pointer
pointer = &x; //Assigning pointer to the address of (&) x
```

The first line is just declaring and initializing a normal **int** variable. The second line is declaring an integer pointer. This is done with the ***** (asterisk) symbol. The asterisk may come directly after **int** or be attached to your variable name, such as **int *pointer;**. The next line actually initializes the pointer to the **address of x**. Remember that the **&** (ampersand) symbol can be read as **address of**. The variable **pointer** now points to **x**.

An important note about declaring pointers is that you have to include the ***** before every variable or else the compiler will think you are declaring a normal variable of whatever type you are working with. For example, **int *ptr, *ptr2, *ptr3;** will produce three pointer variables. However, **int* ptr, a, b;** will produce one pointer variable and two normal variables (such as **x** in the above example). It does not matter if the ***** is attached to **int**. This only applies for the first variable. One way to get around this is to use **typedef**. For example, This will declare three pointers of type **int**:

```
typedef int* intPtrs;
intPtrs ptr, ptr2, ptr3;
```

You are able to use both **x** and **pointer** to access and alter the data in **x**. The tricky part is that the same symbol ***** is used to **dereference** the pointer. **Dereferencing** a pointer means to access whatever it is pointing to. Here is how you use the **dereference operator**:

```
*pointer = 23; //Changed x to 23
cout << "x is now: " << x << endl; //Displays 23
```

This snippet of code **dereferenced pointer** such that it acts like the variable **x**. The ***** always comes before the variable as shown.

Be careful not to omit the ***** as the program would error. You are not allowed to assign an **int** to an **int***.

If you were to **cout << *pointer;**, you would display whatever **x** holds. However, if you **cout << pointer;**, you would see the literal address of whatever **pointer** is pointing to:

```
cout << pointer << " is the same as outputting " << &a << endl;
```

In our example, the above statement would display 100 for both **pointer** and **&a**.

You are able to have more than one pointer pointing to the same variable. For example:

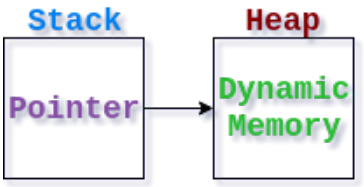
```
int x = 100;
int *ptr1, *ptr2;
ptr1 = &x;
ptr2 = ptr1; //Both ptrs point to x and can be used for manipulation
```

In order to pass a pointer to a function, the formal parameters of the function will include the *****, just like declaring a pointer. The function invocation will be the same as any invocation. For example:

```
void function(double *ptr); //Function declaration
function(pointer); //Function invocation elsewhere in program
```

Please note that passing a pointer this way will allow the function to alter the contents of the passed-in pointer. Use **const** if you want to read the contents of the pointer like: **void function(const int *ptr)**.

Dynamically Allocated Pointers



Dynamic allocation puts data onto the **heap** versus the **stack**. Data in the **stack** is destroyed only when it goes out of scope, such as a function ending. Data in the **heap** may be created and destroyed at any given time. The created pointer will be placed into the **stack**, but will point to data in the **heap**. This is beneficial if you need to free up memory as the program runs. An example of this would be to delete an array that is too small and create a bigger one during run-time. This also allows for flexibility versus hard coding values.

Since pointers point to some location in memory, dynamic allocation omits the step of declaring and initializing a variable, such as **int x = 1**. In order to dynamically allocate memory, you use the **new operator**. This takes the place of declaring and initializing a variable. For example, here is how to dynamically allocate a **char**:

```
char *pointer = new char('M');
```

The declaration portion is the exact same as you've seen. After the equals sign, the keyword **new** is put followed by the type you want to allocate followed by the value in parenthesis. This means that the only way to access the data is via the pointer. Instead of having a variable, we only have a pointer to some location in memory that happens to store the character M. In order to access the **char**, you use the **dereference operator**. For example:

```
*pointer = 'K'; //The contents that pointer points to changes to K
cout << *pointer << endl; //Displays K
```

Now M has been replaced by K. This is akin to re-assigning a regular variable to a different value.

One important aspect of declaring a pointer to be used dynamically is to initialize it to **nullptr**. This signifies that the pointer is pointing to nowhere. You are not able to dereference a **nullptr**, and your program will error if you try. You may see pointers being initialized or set to **NULL**. This is deprecated as there is no distinction between the **int 0** and **NULL**. They are the same and are ambiguous in certain cases, such as an overloaded function call.

In order to free up memory once your need for dynamic allocation is complete, you use the **delete operator**. This deletes the data the pointer is pointing to. The pointer will still exist. For example, if you wanted to delete what the above **char pointer** is pointing to, you would write this in your code:

```
delete pointer;
```

Now, **pointer** will still point to the same address in memory, but that address has been freed and can be used by other programs. Therefore, dereferencing a deleted pointer yields unexpected results (may segmentation fault your program). This is called a **dangling pointer**. A **dangling pointer** is a pointer that points to deallocated memory. For safe practice, it is wise to re-initialize your deleted pointer to **nullptr** if you want to reuse the same pointer later in the program.

Another thing to be cautious of is two pointers pointing to the same address in memory. If one of the two pointers is deleted, both pointers will be a **dangling pointer** and both should be re-initialized to **nullptr**, or some other valid location in memory.

Furthermore, a **memory leak** is where there is allocated memory in the heap, but no pointer pointing to it. There is no way to access this memory again. This happens when you allocate memory with a pointer and then let that pointer point to some other address in memory without using the **delete operator** first. This takes up valuable memory space. Be careful when dealing with pointers.

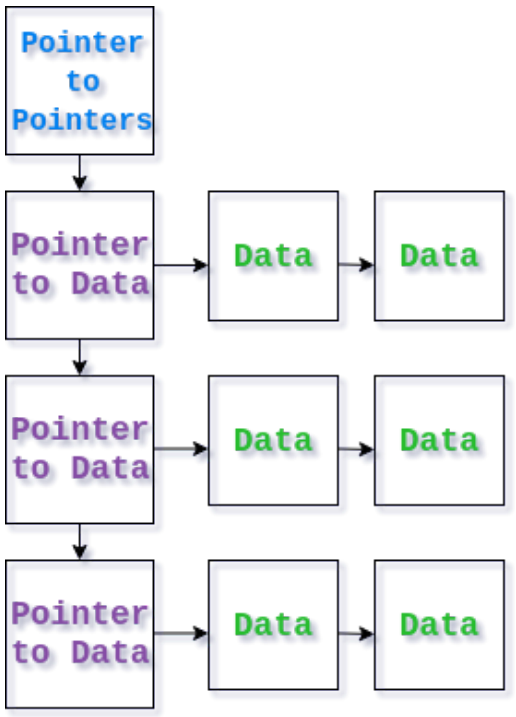
Dynamically Allocated Arrays

To dynamically allocate a 1D array requires square brackets during the creation and deletion of the dynamic array. Here is an example:

```
int* myArray = new int [SIZE]; //SIZE is size you want array to be
delete [] myArray; //Square brackets between delete & pointer variable
```

After you initialize your dynamic array, you are able to use the pointer just how you would a normal array. For example, to access the second element in the dynamic array, you would **myArray[1]**.

Dynamically allocating a 2D array requires further steps. A dynamic 2D array is a pointer to an array of pointers. Each of the pointers in the array point to the actual data. This is what creates the rows and columns. Here is how to allocate a 2D array:




```
int* *myArray = new int* [ROWS];
for(int i=0; i<ROWS; i++)
    myArray[i] = new int [COLUMNS];
```

The first line is declaring and initializing the **myArray** pointer to point to an array of integer pointers. The square brackets will house the number of ROWS you would like. Since this is an array of integer pointers, don't forget there are two ***** after **int** on the left hand side of the equals sign (one to denote a pointer and one to denote that pointer is pointing to more pointers) and one ***** after **int** on the right hand side (to properly allocate memory for pointers versus solely the type of data).

If you have the same number of columns, you can then use a **for** loop to initialize each pointer to data to point to an array of data with the size COLUMNS. This is akin to the one-dimensional array example above, except you do it for however many rows of pointers you have. For example, if ROWS = 3 and COLUMNS = 2, you would have what the picture shows. This is a 3 by 2 2D array. Now, each element can be accessed just as any other 2D array by using the double square bracket on the end of the 2D array variable, such as **myArray[ROW][COLUMN]**.

If you want to initialize each array to have a different number of data elements, you would explicitly do so, such as **myArray[0] = new int [UNIQUE NUMBER OF COLUMNS]**.

Deleting a dynamic 2D array can be thought of as working the initialization process backward. First, you need to delete each pointer to data array and then delete the pointer to pointers. Here is how to delete **myArray**:

```
for(int i=0; i<ROW; i++)
    delete [] myArray[i];
delete [] myArray;
```

Arrow Operator

The arrow operator is syntactic sugar to allow for the **dereferencing of a pointer** and the use of the **dot operator** all in one statement. This is used when dealing with classes, structs, and the Standard Template Library. For example, if you have a dynamically allocated vector of type **char**:

```
vector<char>* charVector  = new vector<char>;
(*charVector).push_back('J');
```

In order to access the vector, since it is dynamically allocated, you need to **dereference** it. Since vector is part of the Standard Template Library, it is a class with member functions. This is akin to how type string is a class with member functions. In order to access these member functions, you need to use the **dot operator**. The above statement adding J to **charVector** looks ugly. It can be cleaned up with the **arrow operator**. This statement can now be equally expressed as:

```
charVector->push_back('J');
```

This syntax is the same no matter if you use a struct, class, or the Standard Template Library (which are templated classes). Here is an example using a struct:

```
struct Node
{
    int data;
    Node* link;
};

int main()
{
    Node* head = new Node;
    head->data = 800;
    cout << head->data << endl;
}
```