### What is an array?

The name of the array is actually a memory address. You can prove this by trying to print out the name of the array – you will get a funky value. This is a number in hex that represents where the array starts in memory:

```cpp
int main() {
    int nums[10];
    cout << nums << endl;
    return 0;
}
```

Thus, the array name is actually a variable that identifies a memory address. Variables that hold memory addresses are called **pointers**.

### Pointers

A pointer variable is a variable that holds the address of some other memory location. In that way, it refers to, or points to, that other memory location. You declare a pointer by putting a `*` in front of its name. For example,

```cpp
int *num; // num is a pointer to an int
double *val; // val is a pointer to a double
```

The actual int or double in these cases doesn’t exist yet. Two options:

1. You can assign the address of an already existing value to a pointer variable using the `&` symbol. Here’s an example:

   ```cpp
   int *num;
   int x = 7;
   num = &x; // num’s value is the address of x (& literally means address of
   ```

2. You use the `new` keyword to set aside a brand new space in memory for holding the target data type:

   ```cpp
   int *num;
   num = new int;
   double *val;
   val = new double;
   int *numbers;
   numbers = new int[10];
   ```

Once you establish the value of the pointer, you can use the pointer’s value to access the value of the cells referred to by the pointer. If it’s a single number (i.e. not an array), just put a `*` in front of the pointer variable’s name to dereference it (which means that you’ll be able to play around with the referred-to cell’s actually value). If it’s an array, you can either **pointer arithmetic** (which we’ll discuss) or use the normal array notation. Examples:

```cpp
*num = 5;
*val = 3.14;
numbers[2] = 3;
```

### Memory issues

When you declare a variable statically, it comes off the system stack. Such memory is managed by the operating system; you don’t have to worry about cleaning it up. When you declare a variable dynamically, it comes of the heap; you do have to worry about cleaning it up (i.e. returning it to the heap). You do so by using the `delete` keyword:

```cpp
delete ptrVarName; // delete a single value
delete [] ptrVarName; // delete a whole array of values
```

To be on the safe side, it’s usually wise to not only delete the dynamic variable, but then set it to point to nothing. The C++ word for nothing is NULL:

```cpp
ptrVarName=NULL;
```

### Pointer Arithmetic

Usually just printing out the value of a pointer – i.e. the memory address – isn’t that helpful. But you can use it as a way to get at the value stored at the referred-to memory location. All you have to do is put a `*` in front of it. This is called dereferencing the pointer.

When you put a `*` in front of the variable name, you are actually dereferencing the value at slot #0 in the array. If you want to reference slot #1 in the array, just add 1 to the name of the array.

Here’s the example:

```cpp
int main() {
    int nums[10];
    nums[0] = 5;
    nums[1] = 17;
    cout << *nums << endl;
    cout << *nums << endl;
    cout << *(nums+1) << endl;
}
```

This is called pointer arithmetic. It is dangerous, but it drives home an important point: arrays are actually pointers.
Arrays are passed by reference because of this pointer stuff

The fact that an array name is really a pointer (memory address) explains why arrays always seem to be passed by reference. Recall that when we pass an array to a function, the function can change values in the array, and those changes stick when we return from the function, even though we don’t seem to be passing the array by reference (after all, we don’t append an & to the data type of the array). The explanation is this: when an array name appears in a function declaration, that array name represents a variable whose value is a memory address. That memory address is passed by value – changes to the memory address will not stick, since it is passed by value. However, the values that are included in the array can change.

OK. Is that it?

No. The fact that arrays are actually memory addresses are used to our advantage when we want to (1) resize an array because we don’t know ahead of time exactly how big to make it (2) return a new array from a function.

Declaring an array whose size can change

We don’t always know how big to make an array. Suppose we are writing software for an MP3 player. We might ultimately need space for up to 10,000 songs. However, when the player ships, there are no songs, and the user only gradually builds up his or her collection. In the meantime, there would be all that wasted space if we declared the array statically.

Static declaration

Suppose we declare an array of 10 ints:
```c
int nums[10];
```
Suppose we start filling it up and find that we need room for more. We can’t redefine nums like this:
```c
int nums[15];
```
We also can’t do something like this:
```c
cout << “How many numbers will you store? “;
cin >> capacity;
in nums[capacity];
```
Why? As we have discussed, when you declare an array, the size must be a constant.
So, static declaration is very limiting: you must use a constant for the size, and, once the size is set, you can’t resize.

Dynamic declaration

However, we can take advantage of the fact that an array is actually a memory address – a pointer.

You declare the array as a pointer variable instead – specifically, in this example, a pointer to an int. To declare a pointer, you use the * symbol as part of its name.

```c
int *nums;
```
The nums pointer has a garbage initial value, just like any variable would. So, you need to initialize it to a memory address – namely, the memory address of the first cell of the new array you are creating. You set aside memory space and return the address of the first cell of it by using the keyword new:

```c
nums = new int[someSizeValue];
```
In this case someSizeValue could be a constant or a variable – variable size indications are ok now.

The new keyword is the great space allocator. Again, it sets aside space and returns the address of the first cell of the newly allocated space. It grabs the space from a part of system memory called the heap.

You refer to the slots of a dynamically allocated array in exactly the same way as a statically allocated array – using a numeric index in []

Cleaning up

When you allocate an array dynamically using new, you cannot forget to clean up that memory. The reason is that the space for the array isn’t allocated by the operating system – you allocate it. So, you can’t rely on the operating system to clean up – you have to.

To return the memory for the array back to the heap, you use the syntax
```c
delete [] nameOfArray;
```
The allocated space will be returned to the heap.

Example

Write an application that asks the user for the number of double values he wants to store, loads the array with that many double values, and then prints those values in reverse order.
```c
#include <iostream>
using namespace std;
int main() {
    int numCount;
    ```
```cpp
double *nums;
cout << "How many numbers? ";
cin >> numCount;
nums = new double[numCount];
for (int i = 0; i < numCount; i++) {
    cout << "Enter number " << i+1 << " ";
cin >> nums[i];
}
cout << "Here are the numbers in reverse: ";
for (int i = numCount-1; i >=0; i--) {
    cout << nums[i] << " ";
}
delte [] nums;
return 0;
}
```

**Passing a pointer variable to a function**
You can pass arrays to functions – the name of the parameter will be proceeded by a *. Nothing new. This holds true regardless of whether the pointer points to one value or a whole array of values.

Example:
```cpp
void printReverse(int *nums, int count) {
    for (int i = count-1; i >=0 ; i--)
    {
        cout << nums*i+ << " ";
    }
}
```

```cpp
int main() {
    int numbers[3];
    numbers[0] = 4; numbers[1] = 3; numbers[2] = 5;
    printReverse(numbers,3);
    return 0;
}
```

**Adding a new value to an array**
Suppose we want to write a function that will append a new value to an array. Let’s call it append. Suppose for now we want to abide by the limits – in other words, once an array reaches capacity, we can’t add more data to it. We’ll return a false if there is no more space. Here’s the function:
```cpp
bool append(int val, int numbers[], int count&, int cap) {
    if (count == cap) {
        return false;
    } else {
        numbers[count] = val;
        count++;
        return true;
    }
}
```

Note that we could have also written it this way, since arrays and pointers are identical:
```cpp
bool append(int val, int *numbers, int count, int cap) {
    //same code
}
```

**Now, let’s revise it to resize if the array is full**
So now we want to resize the array if it is full. If the array is full, we need to resize the array to create space at the end. The way to do this is as follows:
1. dynamically allocate new space that has enough slots
2. copy all the values from the old space to the new
3. delete the old space
4. set the pointer that had been pointing to the old space to point instead to the new space:

Suppose we have an array called currentArray that has 10 slots in it. We need to expand it to 20. Here’s the code:
```cpp
int *newArray;
newArray = new int[20];
for (int i = 0; i < 10; i++) {
    newArray[i] = currentArray[i];
}
delete [] currentArray;
currentArray = newArray
```

If you want this to go in a function instead, you simply have to pass the array pointer by reference rather than by value, since the address to which the pointer refers will change. Here’s the code:
```cpp
void append(int val, int *&currentArray, int& count, int& cap) {
    int* newArrays;
```
if (count == cap) {
    newArray = new int[cap*2];
    for (int i = 0; i < cap; i++) {
        newArray[i] = currentArray[i];
    }
    cap = cap*2;
    delete [] currentArray;
    currentArray = newArray;
}
currentArray[count] = val;
count++;
}  

Could instead return an array from a function

Instead of returning the array from the function as a by-reference parameter, we could have instead returned the array from the function via a return statement. However, when you do so, it must be returned as a pointer; not as an array. Here’s an example of a function called resize:

```c
int* resize(int *array, int count, int newCap) {
    int *newArray = new int[newCap];
    for (int i = 0; i < count; i++) {
        newArray[i] = array[i];
    }
    return newArray;
}
```

Here’s how to call it:
```
int* values;
values = resize(currArray, 5,10);
```

As always, you must remember to clean up any dynamically allocated memory you set up.

Summary

So far in this lecture, we have done this:
1. Review: how to dynamically declare a single variable
2. Review: how to dynamically declare an array
3. Review: how to clean up dynamically allocated memory
4. How to pass a pointer to a function
5. How to pass a pointer to a function by reference (so that the address being referred to can change
6. How to resize an array
7. How to return an array from a function via a return statement

Pointers and struct values

As with “normal” variables, you can declare pointers and use the new keyword.
```
Company *myCompany;
myCompany = new Company;
```
You can then dereference myCompany to gain access to its fields:
```
(*myCompany).name = “Mac’s Bait and Tackle”;
```
You have to be very careful to include the * within the parentheses; otherwise, the compiler will try to dereference the wrong thing.
To avoid this subtle point altogether, use a different notation when dereferencing a struct pointer: the -> notation.
```
myCompany->name = “Mac’s Bait and Tackle”;
The syntax myCompany->name means the same as (*myCompany).name but is less error prone.
```
As always with pointers, you must remember to return the memory you allocated using delete:
```
delete myCompany;
```

Arrays of Structs

As you know, you can have an array of struct-type values. Suppose again we have a struct called Company. We can declare an array of 10 of them, like this:
```
Company companies[10];  // companies is an array of 10 Company objects
companies[0] is the very first of these companies;
companies[1] is the second, etc.
Since companies[0] is a Company, it has fields built into it that you can pick off using the dot notation:
companies[0].name, for example, companies[1].revenue, for example.
```

A dynamically allocated array of struct values

If you don’t know ahead of time how many values are going to be in a collection, you can dynamically allocate the array, like this:
```
Company *companies;
Later, when you find out how many companies you need, do this:
companies = new Company[numberOfCompaniesYouWant];
Because arrays and pointers are synonymous, you access individual Company values within this array in the same way you did before:
cout << companies[0].name << endl; ← for example

| Can you have a dynamically allocated array of dynamic companies | Yeah, but it’s not for the timid, at least not yet. First, here is a statically declared array of pointers to companies: Company* comps[10]; ← ten pointers to Company values You could fill them will all nulls: for (int i = 0; i < 10; i++) { comps[i] = NULL; ← remember, each cell of the array is a pointer } To create a brand new company at location 0, for example: comps[0] = new Company; So, comps[0] is a brand new, dynamically allocated Company. Then, you can play around with it, like this: comps[0]->name = "Confusing, Inc."; |
| Freeing dynamically allocated memory: simple to more complicated | Let’s run through a few examples. 1. If you have dynamically allocated a Company like this: Company *comp; comp = new Company; then you must delete it like this: delete comp; comp = NULL; You use delete, not delete [], because there is just one of them. 2. If you have dynamically allocated an array of Company values, like this: Company *comp; comp = new Company[10]; then you get rid of that array like this: delete [] comp; comp = NULL; 3. If you have dynamically allocated an array of pointers to Company values and then dynamically allocated each of the Company objects to which those pointers point, you have two rounds of deletes: first, delete the Company objects pointed to by the pointers, and then delete the array itself: Company** comps; comps = new Company*[10]; for (int i = 0; i < 10; i++) { comps[i] = new Company; } //later: for (int i = 0; i < 10; i++) { |
delete comps[i]; ← not delete [], because each pointer points to a single object
comps[i] = NULL;
}
delete [] comps; ← delete [], because here we delete the array
comps = NULL;

Worksheet on Pointers and Arrays

1. Declare a struct called `Point` that consists of two doubles, `x` and `y`, and models a point in space.

2. Declare a variable called `line` that is a dynamic array of 10 `Point` values.

3. Assuming you declared `line` correctly in #2, write a statement that sets the `x` and `y` for the point at location 3 in the array to random values.

4. Now declare a variable called `line` that is a static array of 10 pointers to `Point` values.

5. Assuming you have done #4 correctly, write a statement that will create the `Point` value stored at location 3 in the line array.

6. Now that you have done #5, write a statement that will set the `x` and `y` of the `Point` you just created at location 3 in the line array to random values.

7. Now shift gears and create a dynamic array called `line` of 10 pointers to `Point` objects.

8. Write a statement to create the `Point` at location 3 in the array you just created.

9. Now that you have done #8, write a statement to set the `x` and `y` for the point at location 3 to random values.

10. Write code to completely remove the line array you created in #7 from memory.
The Program Stack

- In a modern computer data and instructions are stored on the program stack.
- Every time a function is called, a new record is added to the stack that contains
  - Space for the newly declared variables
  - Address where the new stack record began (i.e. the frame pointer)
  - Address to which to return when the function is done (i.e. ret)

This will be useful to us in a few minutes when we talk about buffer overruns.

Character Arrays

Toward the beginning of the course, we learned about a particularly convenient way to store strings of characters – C++’s string class. Now that we understand pointers and their relationship to arrays, we can revisit the idea of saving strings of characters. This time, we will implement strings as character arrays.

It is possible to declare a string as an array of characters, like this:

```cpp
char* name = new char[10];
```

name is a pointer to a set of 10 contiguous memory locations, each one of which is large enough to store a char value. Such an array can store up to 9 characters. Why 9 characters instead of 10? Because at least one slot must be reserved to store null character, which marks the end of the array and helps C++ figure where the end of the character array is located. If you don’t leave space for that null character, your string stands a great risk of becoming corrupt. As we’ll see this is a prime pathway for a buffer overrun to take place!

You can do lots of things with these character arrays, because several functions have been written for you by the C / C++ gurus. These string-related functions are defined in the <cstring> library, which means you must include the following line at the top of your code:

```cpp
#include <cstring>
```

Once you do, you can invoke any of the following functions:

- `strcat(char* dest, char* src)` – copies the characters from src into dest
- `strlen(char* str)` – tells you how many characters are in str, excluding the null terminating character
- `strcat(char* dest, char* src)` – copies the characters from src onto the end of dest, placing a null-terminator at the end of dest
- `strstr(char* str1, char* str2)` – returns a pointer to the first occurrence of str2 in str1, or null if str2 is not in str1

Here’s a code sample to demonstrate these ideas:

```cpp
#include <iostream>
#include <cstring>
using namespace std;

int main() {
    char* name = new char[10];
    strcpy(name,"Angus");
    cout << name << endl;
    char* fullName = new char[20];
    strcpy(fullName,name);
    strcat(fullName," Young");
    cout << fullName << endl;
    cout << strlen(fullName) << endl;
    return 0;
}
```
Strings are precarious.

- A null character marks the end of the characters
- It is way too easy to write beyond the null character and try to copy more characters into a space than it can handle
- When you do that, you get a buffer overflow.
Buffer Overflow Example

Take a look at this source code:

```c
bool IsPasswordOkay(void) {
    char Password[12];
    gets(Password);
    if (strcmp(Password, "goodpass"))
        return(true);
    else return(false);
}

void main(void) {
    bool PwStatus;
    puts("Enter password:");
    PwStatus = IsPasswordOkay();
    if (PwStatus == false) {
        puts("Access denied");
        exit(-1);
    }
    else puts("Access granted");
}
```

Program stack before call to isPasswordOkay():

```
Stack pointer →

| Storage for PwStatus (4 bytes) |
| Caller ebp-Frame pointer OS (4 bytes) |
| Return address of main()-OS (4 Bytes) |
| . . . |
```

Program stack during program execution

```
Stack pointer →

| Storage for Password (12 bytes) |
| Caller ebp-Frame pointer main() (4 bytes) |
| Return address-main() (4 Bytes) |
| Storage for PwStatus (4 bytes) |
| Caller ebp-Frame pointer OS (4 bytes) |
| Return address of main()-OS (4 Bytes) |
| . . . |
```
Now, if 20 characters are entered, we overflow the password array by 9 bytes. Suppose for example, we entered this for the password:

12345678901234567890

Those extra 9 bytes overwrite the frame pointer, return address, and part of the storage space for PWStatus, as shown here:

The end result is that the program will crash – and crash hard. The program crashes because the return address is altered by the buffer overflow and the new address is either invalid, or memory at that address (a) does not contain a valid CPU instruction; (b) does contain a valid instruction, but the CPU registers are not set up for proper execution of the instruction; or (c) is not executable.