# C and C++ Concepts:

## Arrays

An array is a collection of similar objects.

* int A[5]; will declare an array of ints called A with five integer locations.
* Arrays can be accessed by using the index like A[3] = 29;
* We can also declare arrays with initial values with int B[5]={2,4,6,8,10};

We can iterate through an array using a for loop:

for(int i=0; i<5; i++){

printf(“%d”, B[i]);

}

We can pass arrays as parameters to functions by using the syntax void func(int A[]){}. Arrays cannot be passed by value ever, and are default passed by address. The variable in the function will be a pointer to an array in memory. We could also use the syntax func(int \*A){} to have it point to an integer, but using the bracket syntax makes the code clearer. Since every array is passed by address, we can make changes to the value of the array in memory from inside of the function. We can also return a variable of type array by returning a pointer to an address of an array in heap memory. The function signature could look like int[] func(int n){} or int\* func(int n){} depending on the compiler. Some compilers will not work with the bracket syntax however.

## Structures

A collection of data members that are related under one name. They could be similar or dissimilar types. A structure is also used for user defined datatypes. For example, a rectangle would have two properties length and height. We can make a structure:

struct Rectangle{

int length;

int height;

};

The memory a struct will take is equal to the sum of all of its members. Just declaring the structure will not actually take any space, but each variable of a structure type will be the size of the struct. We can initialize variables of type structure by:

int main(){

struct Rectangle r; //Declaration

struct Rectangle r={10, 5} //Declaration + Initialization

r.length = 15; //Will access the length variable and set it to 15

printf(“Area of rectangle is %d”, r.length \* r.height);

}

The variable r will take 8 bytes of memory because length and height are both ints, and one int is 4 bytes. The Dot Operator will access the members of a structure.

A few examples of structures are:

1. Complex Numbers
   1. A complex number is defined as a+bi where i is an imaginary number
   2. struct Complex{ int real; int imaginary;};
   3. A variable of type Complex will take 8 bytes (4+4)
2. Student
   1. A student structure will contain the details of the student
   2. struct Student{int roll; char name[25]; char dept[10]; char address[50];};
   3. A variable of type Student will take 79 bytes (4+25+10+50)
   4. Example:

struct Student s;

s.roll = 10;

s.name = “John”;

1. Playing Cards
   1. A playing card will have a face value (1 through 13), shape (heart=4, diamond=3, spade=2, club=1), and color (red=1, black=0)
   2. struct Card{int face; int shape; int color;};
   3. Example:

int main(){

struct Card c;

c.face = 1;

c.shape = 1;

c.color = 0;

struct Card c1 = {2, 3, 1};

//C is a black Ace of Clubs

//C1 is a red 2 of Diamonds

}

* 1. Example 2:

int main(){

struct Card deck[52]; //This will be an array of 52 structures,

taking up 12 bytes for a total of 52\*12 = 624 bytes.

deck = { {1,0,0}, {2,0,0}, … , {1, 1,0}, …}

printf(“%d”, deck[0].face); //print out the face of the first card

printf(“%d”, deck[0].shape); //print out the shape of the first card

}

We can also access structures with pointers. For more information on pointers see the section below.

int main(){

struct Rectangle r={10,5};

struct Rectangle \*p = &r;

//p.length = 20; //this doesn’t make sense because p is just a pointer

(\*p).length = 20; //this works because the dot operator takes precedence

p->length=20; //this works much cleaner than above

}

We can also dynamically allocate memory in the heap for structures

int main(){

struct Rectangle \*p; //this is stored in the stack

p = (struct Rectangle\*)malloc(sizeof(struct Rectangle)); //this is stored in the heap

p->length = 10;

p->height = 5;

}

We can also pass the entire structure to functions as a parameter. In the example above, if we had struct Rectangle r, we can pass r to a function with a signature like int area(struct Rectangle r1){}. This will pass the struct r to area by value. The new r1 will be a separate object with its own members. We can use pass by reference with structures so that we can directly modify the values of r from the function, but again this is not the best practice for more complex code. We can pass structures by address as well, using the pointer syntax and the arrow syntax above.

If we have an array as an element of a structure, we can still pass the structure by value. Inside of the structure, the array gets copied into the new element. This way, we can pass an array by value, but it is inefficient when arrays get larger because the compiler will manually loop through each element and copy it into the new array. Since the array was passed by value, it will not modify the actual parameter outside of the function.

## Pointers

A pointer is an address variable, that stores the address of some data. Normal variables are data variables, and pointers can indirectly access that data.

Why do we need pointers?

* In memory, there is the code section, stack, and heap. The code can directly access anything on the stack or in the code section, but the heap is outside of the main program. We can use pointers to access memory on the heap. Likewise, the main program can’t directly reference, or access files stored on the disk, so we would need to have a pointer to that file to read. Additionally, stuff like the keyboard, monitor, internet, etc. can only be accessed through pointers. Pointers give us access to heap memory, resources, and parameter passing.

We can declare variables the following ways:

* Data variable:
  + int a = 10;
  + This is a data variable that is stored in the main code section. It is two bytes, so it takes two address spots in memory, for example memory location 200 and 201.
* Pointer variable:
  + int \*p;
  + p=&a;
  + Since we declared p to be a pointer to a, p will contain the address 200 since that is where a is stored

We can use the pointer in the following ways:

* printf(“%d”, a); //will print 10
* printf(“%d”, \*p); //will print 10 using dereferencing

Every variable we declare is stored in the stack. To access and store variables in heap, we can do the following:

#include <stdlib.h>

int main(){

int \*p;

//C language

p = (int\*) malloc(5\*sizeof(int)); //5\*4=20 so it will allocate 20 bytes in the heap

//lets assume that malloc stores the first address at

5000. Then, when we cast malloc to an int pointer using (int\*), p will have the address 5000 in it

//C++ language

p = new int[5]; //“new” is an operator in C++ and other languages

}

## Reference

This is a feature that is only available in C++. We can give an alias to a variable in C++. For example:

int main(){

int a = 10;

int &r = a;

cout<<a; //10

r++; //will increment a

cout<<r; //11

cout<<a; 11

}

The variable a will get stored in the main code memory. The variable r will be a reference to a. Let’s say that a is taking up 2 bytes at address 200 and 201. Both a and r will be looking at the address 200 in memory. Why would we do this? This seems convoluted. This is useful in parameter passing. We can write small functions using references instead of pointers. In this example program, nothing is gained using references.

## Functions

In C, we can only pass by value or pass by address, but in C++ we can also pass by reference. A function is a piece of code that can perform a specific task. A structure is a group of similar data, a function is a group of similar instructions. They can also be called modules or procedures. Functions can help us to break down larger tasks into smaller repeatable tasks. Putting everything inside of a single function is called Monolithic programming but breaking everything up into functions is called Modular (or Procedural) programming. An example of a function is:

int add(int a, int b){

int c;

c = a + b;

return(c);

}

int main(){

int x, y, z;

x = 10;

y = 5;

z = add(x, y);

printf(“Sum of y: %d”, z);

}

int add(int a, int b) is called the prototype or signature of a function, and the body is the definition of the function. add(x,y) is calling the function. In this case, x and y are “Actual Parameters” and then int a and int b are called “Formal Parameters”. The Actual Parameters in our case live inside of the main memory stack, and then the Formal Parameters exist only inside of the add memory stack. When we return from the function, that stack is deleted from the overall stack. Memory cannot be accessed on any stack layer that wasn’t the one that created it.

## Parameter Passing

There are three main ways of passing parameters.

* Pass by Value
  + Values are passed by the actual parameter values. The actual value is copied into the function formal parameters. Any change to the formal parameters in the function will not affect the actual parameters. We should use Pass by Value when we do not need to modify the actual parameters.
* Pass by Address
  + The address of the parameter is passed to the formal parameters of a function, and the formal parameters must be pointers. Additionally, we must treat the formal parameters appropriately. Since they are pointers, we must change the syntax to work with pointers instead of values. We can use pointers to indirectly access and modify the values of actual parameters in different functions.
* Pass by Reference
  + Pass by Reference is only supported in C++, not in C. First, to change code to be passing by reference, we do not modify the code from Pass by Value. In the function signature we can add an & before each formal parameter to signify that we are passing the reference. References do not take any extra memory, so we are literally just renaming the actual parameters to whatever names are specified in the function signature.

Pass by reference and pass by address are fundamentally different even though they accomplish the same thing. When we use pass/call by reference, the code in the function that we are calling by reference gets added to the first function like in monolithic programming. We can then directly access the actual parameters instead of indirectly with pointers. It is not advised to use pass by reference on complex functions because this will significantly impact the performance of the code. It is a powerful feature but should not be used just because. Passing by address is much more common.

## Classes

The highest level of programming in C code is using structures and functions that operate on those structures. We can easily move from structures and functions to C++ classes. We can change the keyword struct to class, and then put all the functions that operate on the data members inside of the curly brackets. We can then remove all the references to the struct in each of the functions, because we have direct access to the data members now. We can then declare all the members as private or public, and we can do the same with each function.

Inside the main function, we can declare a new function with the type of the class directly. We do not need to say struct or class inside of the declaration.

## Constructor

## Templates