Stanley Wei
CS31 Notes
Fall 2022

Table of Contents

Overview

Background

- C family - programming languages descended from the C programming language
    - C++, Objective C, Java, C#, Swift
        - Python, Perl, Ruby are not C-descended but have similarities
- Compilation/syntax error - program is not “grammatically”/syntactically correct and thus can’t be compiled
    - Runtime/logic error - program fails to accomplish the intended task during execution

C++ Syntax

- C++ ignores all white space, including line breaks
- Hanging C++ expressions (e.g. “3;”) will still compile, even if they don’t do anything

Variables

Primitives

- i++ vs ++i
    - int a = i++; assigns a value of i to a, then adds 1 to i only
    - int b = ++j; adds 1 to j, then assigns a value of j+1 to b
- Assigning a double variable the result of dividing two integers will still round down - one of the integers needs to become a double first

Strings

- Char and string are different types with no inherent defined (automatic) means of converting between the two in the standard library
    - Char-to-string concatenation provided by standard C++ library, not conversion
- Chars can be cast to integers, and will return whatever the integer encoding of the character in question is
    - Integer encoding (e.g. ASCII) varies depending on machine
    - Integers can also be cast to chars, and will return the character corresponding to the integer in the character encoding
- The integer values of various characters vary between encodings
    - Encodings for integers are guaranteed to be consecutive (e.g. int('0') will always be one less than int('1'), even if the actual value of int('0') is unknown
        - Does not hold for letters: 'a' will always be less than 'b', but it may not be consecutive ('a' may not necessarily be exactly one less than 'b', nor does 'A' to 'B', etc.)
            - Letters are consecutive for ASCII (e.g. int('a') + 1 = int('b'), though 'z' and 'A' are notably not consecutive (i.e. lowercase and uppercase letters are adjacent within their own series, but not to each other)
    - Lowercase and uppercase letters may be more or less than each other in different encodings, or even interweaved
- Comparing strings will do a character-by-character comparison of ASCII values until a difference is found

Arrays

- Accessing array indices out of bounds may not terminate the program
    - May instead return whatever unknown variable is located at memory address given by the out-of-bounds index
    - Computes memory address via address @ start + index * size of each element
- ***Array initialization***
    - An array, if provided an initializer of insufficient length, will provide a default value for any unspecified values (e.g. 0 for ints)
        - Providing an empty array as an initializer will fill the entire array
        - Not specifying all values for a const array will result in unchangeable default values being placed in all unspecified items
    - Arrays cannot be reassigned (e.g. array_var = *other array*) after it is initially assigned
- ***Array sizing***
    - By default, C++ does not allow assigning an array a variable (i.e. not specified at compile time) length
    - Once declared, an array cannot be resized
        - Any "resizing" will necessarily allocate memory for an entire second array, cannot modify memory allocation for first array
        - If length was not specified on declaration, an array cannot be resized after it is initialized
    - Arrays of 0 elements are illegal
        - Initializing an array of unspecified length with an empty array will fail
- ***C++ Arrays***
    - There are no inherent subarray, array size/length functions in C++
    - Declaring a multidimensional array in C++ simply declares a one-dimensional array, that happens to contain arrays as elements
    - Passing an array as a function parameter will pass a reference to the original array, not the values
        - To pass in a const array, the array parameter must itself be explicitly declared as const in the function declaration
        - Cout-ing an array will only print the memory address of the array, with the exception of C strings (which are printed as strings)

- Cout-ing a C string with no null byte will result in cout indexing out of bounds until a null byte is found

C Strings

- **C string**: array of char objects, where a '\0\ (null)' char marks the end of the string
    - Can be initialized with string literals
        - A string literal (and, by extension, string object) is actually represented as a C-string in memory
    - An array of C strings is equivalent to a 2D array of characters
- When looping through C strings, the stop condition should be when the character of interest (e.g. string[i]) == '\0'
- Inputting a C string via cin: cin.getline(C string variable, size of C string)
- ***Notable C string functions:***
    - *strcpy*(dest, source) - copies source C string to dest C string
        - Will copy until end of source, even if it goes out of bounds of dest
        - strncpy(dest, source, # chars) - copies limited number of chars
            - Does not automatically append a null byte - copying part of a string w/o having or adding a null byte will result in an invalid C string
    - *strcmp*(string1, string2) - Compares two C strings and returns how the strings compare (greater, equal, less)
        - Returns negative if string1 < string2, 0 if the strings are equal, and positive if string1 > string2
    - *strlen*() - counts number of characters in a C string (up to, but not including, the null byte)
    - *strcat*(dest, source) - copies source C string to end of dest C string (i.e. copies first character of source to null byte in dest, and then continues)
        - strncat(dest, source, # chars) - copies limited number of chars
        - Will throw an error if source is a char instead of a string
    - *strchr*(source, searchChar) - searches for first instance of searchChar in source C string (NULL if none)

Pointers

- ***Pointers*** are variables that store memory addresses, not actual objects
    - Are likely smaller than actual objects
- Pointer operations
    - The &[variable name] operator returns the memory address of a variable
        - Generates a *pointer* to the variable in question
        - [type] *[pointer name] = &[variable];
            - Type is the type of the variable (e.g. a double variable, can only have a double pointer)
                - Defining a pointer with a different type, that can be automatically converted to the variable type (e.g. int -> double), will still fail
            - The & is necessary to generate a pointer; omitting it would cause an error
        - In contrast, [type]& [variable name] generates a *reference* to the variable
    - The *[pointer name] operator creates a reference to the variable the pointer is pointing to
- A **null pointer** (***nullptr***) signifies a pointer not pointing to any value
    - Results in undefined behavior and crashes if dereferenced with *
    - NULL is null value for C++ (before C++11), base C
        - Integer constant 0, where a pointer is expected, is treated as the null pointer value
- *Pointers and references*
    - *References vs pointers:*
        - *References* directly point to the original variable and can immediately be used to modify/access the variable's value
        - *Pointers* only store the memory address (need to be dereferenced to actually modify/access the variable's value)
    - Passing a reference (e.g. as a function parameter) is identical to passing a pointer at a low level - will run equally efficiently
        - References and pointers are different at a C++ language level, but identical at a machine code level
- *Pointer errors*

    - Providing a non-variable value (e.g. a raw number instead of an int object) where a memory reference (e.g. int& instead of int) is requested will cause a compile error
        - Includes variable + raw number
    - Following an uninitialized pointer will result in nonsense behavior or a crash
- *C++ Pointers*
    - Any pointer to a const variable must itself be declared as a pointer to a const, though the pointer itself does not have to be const
        - const [var type]* [pointer_name] declares a (non-constant) pointer to a const [var type]; [var type]* const [pointer_name] declares a constant pointer to [var type]
    - Pointers are also passed by value in function parameters, not by reference
        - To pass pointers by reference, the syntax is [type]* &[parameter name]
    - Both variables and pointers can be defined in the same line (e.g. int i, *p will define integer i, pointer p)
        - The pointer * must be specified for each individual variable in the line
    - Using the * operator between two variables will always result in multiplication, even if one of the objects is a pointer
        - If doing variable * (value of pointer), the syntax is variable**pointer
    - Comparing pointers without dereferencing will compare the memory addresses of the pointers, not the values of the variables they point to

Pointers & Arrays

- Adding an integer to a pointer in an array is akin to incrementing through the array
    - i.e. &arr[i] + j = &arr[i + j]
        - int* ptr = &arr[0] -> (ptr+1) = &arr[1];
        - &arr[i] - &arr[j] = i - j (*Distance between two array elements*)
    - e.g. Adding one to a pointer (pointing at an single element in an array)) moves the pointer to the next element in the array
    - *ptr++; will evaluate to {ptr++; return *(ptr - 1)}, *not* *ptr += 1 (++ has more precedence than *)
        - *ptr = *ptr + 1 *will* evaluate to *ptr += 1;
- Adding one to a pointer in an array moves the [memory address indicated by the] pointer forward by one <u>variable</u>, not one byte

    - e.g. doubles are 8 bytes; so adding 1 to a pointer in a double array, adds 8 to the memory address of the pointer, *not* 1
        - Adding 1 is equivalent to saying “move the memory address by one double variable [however large that is]”
- C++ allows for generating pointers to any of the elements in an array, and a pointer immediately after the array
- Comparing pointers in an array will compare the subscripts of each pointer
    - i.e. Given ptr = &arr[i] and ptr2 = &arr[j], ptr > ptr2 is equivalent to the inequality i > j
- Any array “variable” is actually a pointer to the 1st element (index = 0) of the array, not the array itself, and can take pointer operations
    - e.g. Given double arr[], double* ptr = arr is the same as stating double* ptr = &arr[0];
        - double* ptr = arr + 5 is the same as saying double* ptr = &arr[0] + 5
    - The same is true for arrays passed into functions - in a function parameter, double arr[] is the same as double* arr
- Using a square bracket operator on a pointer in an array will return the element given by moving the pointer forward by however many elements are passed in the square brackets
    - i.e. Given double *ptr = &arr[i], ptr[j] = arr[i + j]
    - (This is why arrays passed as function parameters can be accessed + modified with square brackets, despite being passed as pointers)

Control Structures

Ifs and Booleans

- else if is very literally separated into else and if
    - Given if() {} else if() {} else {}, the second if and second else are a single statement encapsulated in the first else, a la if() {} else { if() {} else {} }
    - if() {} else if() {} else {} is effectively equal to if() {} else { if() {} else {} }
- if statements only accept a single statement
    - Multiple commands -> {}
- Assigning a variable a value in a boolean statement will return the value for the purposes of the if
- Putting a raw value without assigning a variable in a boolean statement will cause the value to be interpreted as a pointer and be interpreted as true
    - Will compile and pass even if the value type has no automatic conversion to bool

Loops and Switches

- Expressions in for(expression1; expression2 ; expression3) can be omitted
    - for(; ;) will evaluate as always true
    - Will not compile if the value type has no boolean conversion (e.g. Strings cannot be automatically converted to a boolean and thus will not compile)
- Failing to add a break to a switch statement will automatically run the next case, irregardless of provided variable value
    - Will run through the entire switch until either the next break or the end of the loop

Functions

- Functions can only return one value
- Function parameters by default copy only the value of a provided variable
    - Modifying the parameter variable will only modify the local value within the function, not the actual variable passed into the function
    - Reference parameter - placing “&” after the type declaration of a parameter (e.g. void function (int& x) {}) will create a reference to the passed-in variable
        - Essentially provides the memory address of the passed-in variable

        - Modifying the value within the function will also modify it for the original passed-in variable outside of the function
- Passing a function parameter as a different variable type will only compile if there is an *automatic conversion* between the given and defined types
- Passing an array as a function parameter will only pass the memory address of the first element of the array
    - Will not indicate the length of the array, etc.
    - If passing multidimensional arrays, must explicitly pass in the size of all dimensions (save for the first), e.g. int x[][10][20] (the last two dimensions must be specified)
- ***Default parameters*** - parameters can be specified to take a certain value, if the value of the parameter is not explicitly defined by the function call
    - e.g. Given function foo(int i, int j = 0), calling foo(10) will run the function with j = 0
    - Said default value can depend on runtime computation (i.e. not be known at compile time), but cannot depend on other parameters
    - In instances where a single function call to an overloaded function, could potentially be a call to multiple functions (due to default parameters), the compiler will throw a compilation error
- Functions can be **overloaded** (have multiple functions declared with the same name)
    - Overloaded functions must have different parameters (e.g. void foo(string foo2) & void foo(int foo2) can coexist, but void foo(int foo2) and void foo(int foo3) cannot)
        - If a parameter has a default value specified, its parameters must not overlap with those of another function (with or without specifying the parameters with default values)
    - Given functions that have different parameter types with automatic conversions (e.g. void foo(int foo2) & void foo(double foo2)), the compiler will run whichever function has the least amount of conversions needed (from the provided parameters in the function call, to the function-demanded types)
        - Having equal numbers of automatic conversions between two functions will result in an error
    - Overloaded functions are allowed to have different return types
        - Overloaded functions cannot differ *only* in return types (must have some other difference e.g. in parameters, even if the return types are completely different)

Misc

- cctype functions: isalpha, isdigit, islower/tolower, isupper/toupper

Objects

Structs/Classes

- **Structs** are essentially custom variable objects
    - Declared with struct/class StructName {};
        - Can also make an incomplete type declaration (to implement later) with struct/class StructName;
    - Can have custom attributes - e.g. declare [type] *variable name*; in the struct declaration
    - Can create variables of type [struct], e.g. [struct name] *struct variable name*;
        - Access variable with *structvariablename.variablename*;
    - Terminology:
        - Member function/method/operation - functions of a struct
        - Data member/instance variable/attribute/field - variables of a struct
- **C++: Structs and classes are identical**
    - Have identical declarations, except for beginning with “struct”/”class”
        - Only functional difference - methods/attributes default as public for structs, private for classes
    - Convention:
        - Structs mainly used for simple collections of data without interesting behavior, where there is no reason to hide attributes/methods
        - Classes are used when dynamic/interesting behavior where implementation might benefit from being hidden/abstracted
- Structs (i.e. variables of type struct) can also be passed in function parameters
    - Struct variables are default passed by value, *not* by reference
- Structs can also be declared as const, just like regular variables
    - Const structs are not allowed to run non-const methods; will throw an error if any non-const methods are run
        - For a method to be run, must be explicitly defined as const (by writing “const” after the parameter parentheses), e.g. void foo() **const**;
            - Applies both for declarations in the struct declaration and implementations outside the struct declaration

- When accessing properties of a struct object via a pointer to the struct object, the syntax is *pointername->propertyname*, e.g. ptr->id
    - By default, the . operator has greater precedence than the * operator, so *ptr.id will evaluate as *(ptr.id) and give an error
- Rather than allowing programs to modify struct attributes directly, having changes to attributes be done via struct **methods** can reduce frequency of bugs
- Declaring methods - declare a function/function prototype within the closed curly braces of a struct declaration
    - Special syntax is involved in implementing a struct method prototype outside of the struct declaration's curly braces
        - Instead of simply writing, e.g. void func(), the struct the function belongs to needs to be specified, i.e. void FooStruct::func()
- Within a struct method, the keyword ***this*** represents **a pointer to the specific struct variable that called the function**
    - Is needed in cases where there is ambiguity between different variables
        - Is unnecessary and can be omitted in cases where there is no ambiguity
    - e.g. given void FooStruct::func() { this->id = 0 }, FooStructVariable.func() will set the attribute id of FooStructVariable to 0
- When dealing with pointers to structs, **the -> operator can be used to access the methods/attributes of the struct being pointed to**
- Access modifiers: struct methods and attributes can be classified as **public** (viewable outside of the struct) or **private** (only viewable by the struct)
    - Similar formatting to a switch and its cases, i.e. struct FooStruct { **public:** void f(); int v; **private:** void g(); int w;};
    - Attributes and methods not explicitly defined as public/private are public by default for structs, private by default for classes
- **Constructor** - a struct method that is automatically called by any newly-created variables with type of that struct
    - Has the same name as the struct and no return type, e.g. struct Foo { Foo(); };
    - Constructor can also have function parameters
        - Creating new variables of type struct: FooStruct FooVariable([*parameters*]);
        - **Parameters after variable name are only included when the constructor takes >1 parameters**

            - Including parentheses when creating variables of structs with no parameters will result in creating a function instead of a variable
    - **Default constructor** - constructor with no arguments
        - Is used for initialization when creating an array of the struct
        - Is not required - if a struct has constructors defined, but only constructors with parameters, said parameters must then be included when creating any new variables of the struct type
            - Prevents creating an array of the struct
    - If a constructor is not provided by a struct declaration, the C++ compiler will write a default constructor in place of one
        - Compiler-written default constructor leaves attributes of built-in types (e.g. int, double) uninitialized, call default constructors of any attributes that are classes/structs
        - If a constructor is provided, the compiler will not write a constructor - only the custom-written constructor may be used
    - Constructors will perform automatic conversions, if passed a variable that would require one (e.g. constructor asks for int, program calls with double -> C++ will perform an automatic double-to-int conversion and continue)
- When creating arrays of structs, the constructor will be run for *every element*, even if not every element is actually needed/used
    - May result in unnecessary processing/time spent
    - Alternative - creating arrays of type pointer to struct (default uninitialized) and creating new structs variables + adding them to the array when needed
        - FooStructPtrArray[index] = new FooStruct;
- Changing the struct a pointer is assigned to will **not** delete the struct that the pointer was previously being pointed to (the struct will remain in memory, even if it's not in use)
    - Creates **garbage** - memory that has been allocated but is no longer accessible (taking up memory without being used)
    - Deleting a struct in memory: delete PtrToFooStruct;
        - Keyword delete takes a pointer, will delete whatever object is at the address given by the pointer
- **Destructor** - a struct method that is automatically called by the struct whenever the struct is deleted with the delete keyword or goes out of scope

    - Defined as **~StructName()** {} (no return type, name is the same as the struct itself + a tilde beforehand)
        - Cannot take parameters
    - Generally used for performing any necessary garbage cleanup before deleting a struct
- When creating a new object, the member variables are initialized before the constructor is run (i.e. given a struct with an int member variable, the int is first initialized, and only then is the struct variable initialized [around it])
    - Destructors occur in the opposite order - first the [outer] struct has its destructor run, then the [inner] member variables have their destructors run

Programming

Memory Allocation

- **Garbage** - memory that has been allocated but is no longer accessible (taking up memory without being used)
- **Memory leak** - accumulation of garbage in a program over [a long] time
    - May potentially result in the computer running out of memory despite not having too many variables in use
- Keyword **delete** takes a pointer, will delete the variable at the address given by the pointer
    - The pointer becomes a *dangling pointer* - no longer points to a variable
        - Will result in undefined behavior if dereferenced
    - delete can be called on null pointers without issue
- Where objects live:
    - Named local variables/automatic variables (e.g. in a function, as a function parameter, etc.) live on “the stack”
        - Are no longer available once the function exits
    - Variables declared outside of any function live in “global storage area”/”static storage area”, exist until the program ends
- **Dynamically allocated objects** (e.g. structs created with “new”) live on “the heap”, only go away once the program ends
    - Must be explicitly deleted with the delete keyword to avoid becoming garbage
        - Destructors can be used to ensure garbage is cleaned