CS 16 Notes: Intro to C++

Chapter 1.1: Computer Systems

- Software: programs used by a computer
    - Operating System: a program that allocates the computer's resources to different tasks the computer must accomplish; the master program that spearheads other programs
    - Program: set of instructions for the computer to follow
    - Data: input to the program
    - High-level languages: languages similar to human languages; must be translated into machine language before the computer can understand
    - Low-level (Assembly) languages: languages the computer can understand
    - Machine languages: written in binary
    - Compiler: takes source code and translates into machine language object code
    - Linker: combines object code from compiler and links it with other object code to produce a complete program
- Hardware: physical machines that make up a computer
    - PC: Personal Computer designed to be used by 1 person at a time; contains 5 main components
        - Input Devices: communicate information to the computer (i.e., keyboard)
        - Output Devices: computer communicates information to user (i.e., monitor)
        - Processor (Central Processing Unit | CPU; chip): follows instructions in a program and performs specified calculations
        - Main Memory (Random Access Memory | RAM): consist of memory locations (strings of binary) split into groups of bytes associated with an address; the first byte in the group is used as the address (usually grouped together to form a larger memory location); used while following instructions of a program; information can be easily accessed
        - Secondary Memory: used to keep a permanent record of information; requires sequential access to find information
            - Files: stores various units of information
            - Hard Disks: fixed in-place and cannot be removed from the disk drive
            - Flash Drives: stores memory without using power
    - Workstations: industrial-strength PC
    - Mainframe: larger computer generally shared by more than 1 user
    - Network: connected computers to share resources and information

- Binary: numeric base 2 system with only 0s and 1s
    - Bit: a binary digit; either a 0 or a 1
    - Byte: 8 bits of memory
    - Address: number that identifies a byte

Chapter 1.3: Introduction to C++

- Qualities of high and low level language
    - can directly manipulate memory (low-level)
- C is a subset of C++; C programs are also C++ programs
- (Executable) Statements end with a semicolon; they are instructions for the computer to follow
- Compiling and running a C++ file: g++

Basic Skeleton:

```
#include <iostream>                 //every main program starts with
using namespace std;                //these two lines of code
int main() {                        //and must have a main function
        return 0;                   //returns 0 to end execution of the program
}
```

```
#include <iostream>                 //an include directive; library of code handling input & output
using namespace std;                //provides convenient usage of standard functions
```

Variable Declarations and Initialization:

```
VAR_TYPE VAR_NAME1 = VAL, VAR_NAME2 = VAL, etc.; //initializes VAR_NAMES with VAL as VAR_TYPE variables
VAR_TYPE VAR NAME1(VAL), VAR_NAME2(VAL), etc.;   //alternative for initializing variables in declarations
```

Input & Output Statements:

```
cin >> VARIABLE;                    //asks for user input and stores it in the VARIABLE
cout << STATEMENT;                  //prints out the STATEMENT
```

Note: pretend c is the keyboard/monitor; the arrows point in the direction of input/output from them

Chapter 1.4: Testing and Debugging

- Syntax Errors: violation of programming syntax
    - Error Message: given for direct violation of syntax rules
    - Warning Message: indicates a likely mistake, but not a violation of syntax
- Run-time Errors: errors that occur during program execution
- Logic Error: semantic mistake in translation to a programming algorithm

Chapter 2.1: Variables and Assignments

- Variables are memory locations
- Identifier: name of a variable (case sensitive) that must start with a letter or underscore and only contain letters, digits, or underscores in its name; they are case sensitive
    - Reserved/Keywords are special identifiers which cannot be used as variable names
- Declaration: telling the compiler the data type of a variable
- Assignment Statement: a statement assigning the value of a variable
    - Initialization: the first time a variable is assigned a value
    - Uninitialized variables have garbage values (whatever is left in memory)

Chapter 2.2: Input and Output

- Input Stream: stream of input fed to the compiler for the program to use
    - does not matter where the source comes from
    - program does not read input until user presses return key
- Output Stream: stream of output generated by the program
    - Insertion Operator: `<<`
    - cout does not add spaces nor new lines
- Include Directive: imports a library
- Header File: file imported using include directive `<header_file>`
- Using Directive: using a specific collection of names to prevent confusion with names; also, includes definitions of functions, etc.
- Escape Sequence: characters with special meanings
    - Include a newline character in the last line of a program
- Raw String Literals: entire strings in between the quotes of R"(STRING)"

Special Characters:

\n	new line	\t	horizontal tab
\a	makes a “beep” sound	\\	backslash
\”	double quote	\0	null character

Chapter 2.3: Data Types and Expressions

- short (2 bytes), int (4 bytes), long (4 bytes)
    - result of two integers combined using an arithmetic operator is an integer
    - modulo (%) can give different results when working with negative integers
- float (4 bytes), double (8 bytes), long double (10 bytes)
    - doubles have at least 14 digits of accuracy
    - integers (and bools) can be stored as doubles
    - doubles are printed as integers without floating point precision
- char (1 byte): single character enclosed in single quotes
    - small integers can be stored as chars and vice versa
- bool (1 byte)
    - 0 is false; 1 (or any nonzero number) is true
    - integers can be stored as booleans and vice versa
- string
    - must be included from the string library using #include <string>
    - cin only reads until a whitespace character for strings
- change: (VAR %= VAL) changes VAR by VAL%
- If cin receives the wrong data type, no error is thrown; instead, the variable keeps its past value (if any)

Chapter 2.4: Simple Flow of Structure

- When no curly brackets are present following the control structure, the following statement (end marked by a semicolon) is the enclosed statement
- And Operator: &&
- Or Operator: ||

Conditional Statements:

```
if(BOOL_EXPRESSION) {
        CODE BLOCK
}
else if(BOOL_EXPRESSION) {
        CODE BLOCK
}
else {
        CODE BLOCK
}
```

Loops:

- do-while loops execute body before checking whether the boolean expression is true

```
while(BOOL_EXPRESSION) {
        CODE BLOCK
}

do  {
        CODE BLOCK
}
while(BOOL_EXPRESSION)
```

Chapter 2.5: Program Style

- To prevent a variable from changing after initialization, declare const
    - (e.g., const double PI = 3.14)

Chapter 3.1: Using Boolean Expressions

- Order of Precedence
  - Unary Operators: +, -, ++, --, !
  - Binary Arithmetic Operators: *, /, % followed by +, -
  - Relational Operators: <, >, <=, >=
  - Boolean Operator: !=, == followed by && followed by ||
  - Variable Assignment: =
- Short-Circuit Evaluation: used in C++, if the first operand of the && or || operator gives enough information to evaluate the boolean expression completely, then the second operand is not evaluated
- Complete Evaluation: evaluates both operands of the boolean operators && and ||
- Enumeration Type (enum): declared list of integer constants
  - Doesn't support any operations to save memory
  - Default value of the first constant is 0, subsequent constants are incremented by 1
  - Constants can be initialized with non-default value

Chapter 3.2: Multiway Branches

- Dangling Else Problem: the compiler pairs a dangling else statement with the nearest if statement without an else statement
- Nested Code Scope: variables can be declared in both an outer and inner block of the nest with the same name; the outer declaration is ignored in the inner nest and vice versa

Switch Statements:

- only works with integral data types (short, byte, int, char)
- case CONS must be constants
- compares the VAR with each of the cases' CONs using ==
- If true, code executes until the compiler breaks out of the switch statement (or the entire statement is executed)
- Default statement executes if VAR doesn't match any of the cases

```
switch(VAR) {
        case CONS1:
                CODE BLOCK
                break;
        case CONS2:
                CODE BLOCK
        case CONS3:
        case CONS4:
                CODE BLOCK
                break;
        default:
                CODE BLOCK
}
```

Chapter 3.3: More on Loops

- Increment and Decrement Operators: ++, --
    - var++ returns the original value, then increments the variable by 1
    - ++var increments the variable by 1, then returns the new value
- `break` statement exits the loop

For Loops:

- can initialize and declare a variable for VAR_INIT; these are local to the for loop
- semicolon immediately after a for loop creates a null statement
- can leave parts of the for loop as empty; beware of infinite loop

```
for(VAR_INIT; BOOL_EXPRESSION; UPDATE) {
        CODE BLOCK
}
```

For-Each Loop (C++11):

- iterates through the iterable
- given the ith iteration, ITERABLE[i] is stored in VAR

```
for(VAR: ITERABLE){
    CODE BLOCK
}
```

Chapter 3.4: Designing Loops

- Sentinel Value: throwaway value distinct from values that are plausible
- Flag: a variable to track when an event has taken place

Chapter 4.1: Top-Down Design

- breaking programs down into subcomponents (functions)

Chapter 4.2: Predefined Functions

- can randomize the pseudorandom numbers using srand( ) from <cstdlib>
- Type Casting: changing a variable type using static_type<VAR_TYPE>(VAR_CHANGE)

Chapter 4.3: Programmer Defined Functions

- return type void = no return value
    - can still use return; command but not necessary
- arguments passed into functions are values, not the variables itself
- functions must be declared before the main( ) structure if they are called in main( )

Function Prototype:

- declaration of a function to describe how it's called
- no need to give argument types names, just declaration of types

```
RET_TYPE FUNCT_NAME(ARG_TYPE1, ARG_TYPE2, etc);
```

Function Definitions:

```
RET_TYPE FUNCT_NAME(ARG_TYPE1 ARG_NAME1, etc) {
        CODE BLOCK
}
```

Chapter 4.4: Procedural Abstraction

- use comments and declarations to explain how the code works to prevent the necessity to dissect code

Chapter 4.6: Overloading Function Names

- Overloading: providing two (or more) definitions to the same function name (presumably due to a different amount of arguments passed in via various scenarios)
    - must not contain the same amount of parameters of the same data type
- Automatic Type Conversion: to fit specifications of function declaration, the formal parameters passed in a function will automatically be casted to the declared data type (e.g., int -> double)
    - will not work in the event of overloading

Chapter 5.2: Call By Reference Parameters

- Call By Value Functions: copies of the arguments passed into the function are used as parameters
- Call By Reference Functions: formal/reference parameters are passed into the function instead of copies
    - more efficient use of memory since no need to make a copy
- `&` indicates that parameters passed in are formal; Ex: `void function_example(double &number);`
- Initializing references to variables: VAR_TYPE &VAR_NAME = VAR;
    - VAR and VAR_NAME are the same memory location; cannot be unbound

Chapter 5.3: Using Procedural Abstraction

- Precondition: what is assumed to be true when the function is called (to ensure proper functionality)
- Postcondition: effect of the function call
- test all possible flows of the possible function to ensure all branches work correctly
- test edge cases: (values in which the program is supposed to change behavior)

Chapter 5.4: Testing and Debugging Functions

- Driver Program: a special program to test each function individually outside the main program
    - only one untested function should be in a driver program
- Stubs: simplified versions of functions that do not perform correct procedures but validates the program to test the other components

Chapter 5.5: General Debugging Techniques

- Localizing Errors: pinpointing the location of a bug
- Debugger: program that allows user to step through specific lines of code to pinpoint bugs

Assert Macro:

- must import the cassert directive
- when the BOOL_EXPRESSION is false, the program will throw an error

```
#include <cassert>
assert(BOOL_EXPRESSION);
```

Chapter 6.1: Steams and Basic File I/O

- Classes defined in <fstream> directive
- Stream: flow of data
- Output Stream uses the ofstream object; Input Stream uses the ifstream object
    - .open("FILE_NAME") opens FILE_NAME
        - pass argument ios::app to append to a file
    - .close( ) closes the file and prevent issues
    - .fail( ) returns true if object did not successfully open file
        - opening files that don't exist does not throw an error
    - OUTFILE_OBJ << DATA; //writes DATA to corresponding outfile
    - INFILE_OBJ >> VAR; //reads some data of appropriate data type into VAR
    - INFILE_OBJ will become false when the file is read over
- Member function: function associated with an object
- Dot operator: "." to reference member variables and functions
- Calling object: object using the dot operator
- exit(BOOL): function that ends the program immediately

- exists in the <cstdlib> directive, using namespace std;
- BOOL = 1 if function exits due to error
- BOOL = 0 if the function exits for any other reason

Chapter 6.2: Tools for Stream I/O

- Flags: used to format output for the .setf( ) function
  - ios::fixed //numbers not written in scientific notation
  - ios::scientific //numbers written in scientific notation
  - ios::showpoint //shows decimal and trailing zeroes
  - ios::showpos //plus sign shown for positive numbers
  - ios::right //right align given a set field width
  - ios::left //left align given a set field width
- Manipulator: used to format output in a nontraditional way after << operator
  - setw(NUM) //sets the next arg with exactly NUM characters
    - under <iomanip> directive
  - setprecision(NUM) //sets the next arg with NUM decimal precision

Floating Point Numbers Formatting:

- formats decimals with INT_VAL number of decimal places for specific OUTSTEAM_OBJ

```
OUTSTEAM_OBJ.setf(ios::fixed);          // fixed-point representation
OUTSTEAM_OBJ.setf(ios::showpoint);      // show the decimal point
OUTSTEAM_OBJ.precision(INT_VAL);        // decimal precision
```

Chapter 6.3: Character I/O

- .get(CHAR_VAR); //reads the next char and stores in CHAR_VAR
- .put(CHAR); //writes CHAR into corresponding output stream
- .putback(CHAR); //puts CHAR back into input stream to be read
- string directive functions:
  - getline(IFSTREAM_VAR, STR_VAR); //reads next line of stream and stores it in STR_VAR
- cctype directive functions:
  - toupper(CHAR); //returns uppercase letter of CHAR
  - tolower(CHAR); //returns lowercase letter of CHAR
  - isupper(CHAR); islower(CHAR); //returns BOOL value associated with function
  - isalpha(CHAR); isdigit(CHAR); isspace(CHAR); //returns BOOL value associated with function

Chapter 7.1: Introduction to Arrays

- Array Declaration: varType varName[ArraySize];
  - stores enough memory such that varName occupies ArraySize * size of VarType
  - varName is a reference to the memory location of the 0th index of varName
- Array Initialization: varType varName[ArraySize] = { element0, element1, etc… };
  - ArraySize can be omitted when arrays are initialized; size will be # of elements passed in
  - If # of elements passed in is less than ArraySize, the rest of the array will be initialized with 0 (or equivalent)
- Indexed Variables = Subscripted Variables = Array Elements
- Array elements are stored next to each other in memory
- Random Access: ability to access any element in O(1) time (true for arrays)
- Declared size and base (data) type of array cannot be changed
- Array out of bounds errors may or may not throw an error (due to memory access permissions)
  - Has permission: may access and return value (but not change)
  - Doesn't have permission: segmentation fault
- Segmentation fault: program crash due to permission issues accessing memory or accessing memory locations that don't exist
- auto Data Type: automatically determines the data type
- Array variables cannot be reassigned memory locations

Chapter 7.2: Arrays in Functions

- Array Parameter: behaves like a call by reference parameter
    - Difference between call by reference parameter is only the memory address is passed as the parameter, not the data type size nor size of the array data type size can be deduced but the size is not known
- Constant Array Parameter: declare array parameter of a function constant using const to prevent accidental changes of the array

Chapter 8.1 Array Type for Strings

- C String: array of characters
    - Length of string is 1 less than the size of the array since the last element is "\0"
    - Null character distinguishes C String from array of chars
    - Cannot use = or == for assignment or equality checks
- <cstring> Library
    - strcpy(CSTR_VAR, str); //sets the value of CSTR_VAR to str
    - strcmp(CSTR_VAR1, CSTR_VAR2); //returns 0 if equal, a positive number if the first different character in CSTR_VAR1 has a larger ASCII than that of the same index in CSTR_VAR2 (and negative for vice versa).
    - strcat(CSTR_VAR1, CSTR_VAR2); //concatenates CSTR_VAR2 after CSTR_VAR1
    - strlen(CSTR_VAR); //returns the length of the CSTR_VAR
- <cstdlib> Library
    - atoi(C_STR); //converts C_STR to an integer
    - atol(C_STR); //converts C_STR to a long
    - atof(C_STR); //converts C_STR to a float

Chapter 8.2 String Class

- <string> library (using namespace std;)
    - allows strings to be treated as a basic type
    - valid operators: =, +
    - two constructors: one to initialize an empty string, and one that takes an argument;
    - getline(cin, strVar, sep); //reads line until reading the sep character; default is "\n"
    - STR.length( ); //returns length of STR
    - STR.at(i); //checks to see whether i is an illegal index
    - STR.substr(position, length); //returns substring starting at position of that length
    - STR.erase(position, length); //removes substring of size length starting at position
    - STR.insert(position, string); //inserts string at position of STR
    - STR.find(string); //returns index of first occurrence of string
    - stoi(string); //returns integer of string
    - stod(string); //returns double of string
    - to_string(numeric); //returns string of argument
- C++ will automatically convert cstrings to strings when needed

Chapter 8.3 Vectors

- arrays that can grow and shrink in length
- declaration: vector<type> v;
- template class: can plug in any data type to create a vector v of that type
- vector.push_back(arg); //appends arg to vector
- vector[i] = arg; //changes the ith element of vector to arg
- vector.size( ); //returns the size of the vector in an unsigned int
- unsigned int: values that are only nonnegative
- can work with vector capacity to manage memory efficiency; vectors, by default, double in size when the capacity cap is hit

Chapter 9.1 Pointers

- Pointer: memory address of a variable
- Dangling pointers: undefined (nonnull) pointers

- Pointer Declaration: varType *varName;
- & operator (&varName): refers to the memory address of varName
- * operator (*varName): dereferences the pointer and works with the value of varName
- Dynamic variables: created and destroyed while the program runs
    - new operator: returns a pointer to a new, nameless variable of a specified data type
    - Declaring dynamic variables: varType *pointerName = new varType;
    - terminates program if there is not enough memory
- Freestore: special area of memory in the heap reserved for dynamic variables
    - delete operator: eliminates dynamic variable and frees memory
- Memory leak: when pointers are removed, dynamic variables cannot be be accessed/deleted
- Automatic (ordinary) variables: dynamic properties automatically controlled (local variables destroyed when function calls are complete)
- Global variables: declared outside all function definitions
- Type Define: typedef varType varName; \\varName can be used as a data type (that is of varType)
- Pass-by-address function: passing a pointer into a function

Chapter 9.2 Dynamic Arrays

- Dynamic Arrays: size is not specified when the program is written; determined when the program is run
    - created using the new operator
    - when deleting, square brackets appear before varName; delete[ ] varName;
- Pointer Arithmetic: arithmetic of addresses
    - pointerName + num; calculates the address of pointerName + num * size of data type of pointer
        - num can be positive or negative
    - pointerName1 - pointerName2; calculates the number of size of data types the pointers are away from each other

Chapter 10.1: Structures

- objects without member functions
- a structure variable holds structure values with each individual value being called a member value
- initializing structures: structName varName = {value for each member var};

Structure Definition:

- structure tag: legal identifier of a struct (structName)
- member names: names of member variables
- before ending semicolon, can declare vars of structName type

```
struct structName{
  member vars;
} varsOfstructNameType;
```

Chapter 10.2: Classes

- Scope Resolution operator "::": class::function utilizes said function within that class
- Scope vs Dot operator: dot operator is associated with objects, scope is associated with class names
- Private: definitions within the private section is only allowed to be used within the class, whereas public definitions are allowed publicly
- Encapsulation: combining a number of items into a single package
- Assignment operator makes a copy of one object to another
- Constructor with no arguments: do not pass empty parenthesis to signify no arguments

Class Definition:

```
class className{
 public:
   CODE BLOCK
 private:
   CODE BLOCK
};
```

- Constructor Initialization Section: an optional section to initialize member variables before the constructor body

Ex: ClassName(type1 param1, type2 param2) : member1(param1), member2(param2) { CODE BLOCK }

Chapter 10.3: Abstract Data Types

- Abstract Data Types: data types which the programmer does not have access to the implementation of values and operators
- Interface/Specification: public member functions of a class + usage comments
- Implementation: private members + implementation of all members

Chapter 10.4: Inheritance

- Derived Classes: a class is derived from another if it inherits all its parent features and contains additional specific features

Class Definitions:

```
class derivedClassName : parentClassName{
   CODE BLOCK;
};
```

Chapter 13.1: Nodes & Linked Lists

- Node: data and pointer to next node
- Arrow operator: "->" combines dereferencing operator and dot operator to specific member variable
- NULL, nullptr has address 0
- Linked List elements are not stored next to each other in memory
- Linked List: list of nodes
- Doubly Linked List: list of nodes which also contain a pointer to the previous node
- Circular Linked List: the tail node points back to the head node

Chapter 14.1 Recursive Functions

- stack frame = activation frame
- variables last created on a stackframe are the first variables to be deleted
- stack overflow: exceeds size limit of stack
- stacks: last in first out
- queues: first in first out

Chapter 15.1, 15.2 Inheritance

- Derived (Child) Classes inherit from a Base (Parent) Class
    - Child classes has all the functionalities of a parent class that are not private
    - An object of type DerivedClass is also an object of type BaseClass
    - The access level declared on the base class in the derived class declaration denotes the level of protection (or better) the inherited members will have
        - public denotes that public parent members stay public, and protected parent members stay protected
    - A derived class does not inherit any constructors, destructors, or assignment operator definitions of the base class
        - Constructors and assignment operator overloading should all make a call to the parent class; a call to the parent destructor is done automatically
    - Derived classes must call the base class constructor in the initialization part of the constructor
        - If not explicitly called, the default constructor will be called on the parent class
- Redefinition: overriding a definition in a parent class with a new definition in the child class
    - not the same as overloading a function
- Function Signature: function name with the sequence of data types of parameters
    - excludes const and &
- Protected: members defined with the protected access level are inherited by child classes and not accessible outside these classes

Derived Class Example:

```
class DerivedClass : accessLevel BaseClass{
   public:
      DerivedClass(paramTypes paramVars) : BaseClass(baseParams){
         //CODE BLOCK
      }
}
```

Chapter 15.3 Polymorphism

- Polymorphism: associate multiple definitions to a single function
- Virtual Functions: allowing the compiler to undergo late binding
    - The virtual keyword must be added to the function declaration (and not function definition) of the base class (before return type)
    - Destructors should be virtual to ensure no memory leaks when deleting from a pointer of type Base Class pointing to a type of Derived Class.
    - Overriding: virtual functions that are defined in a child class (they are not redefined)
- Late Binding: determine the specific implementation of a function at run time

Chapter 16.1, 16.2 Exceptions

- Exceptions: any data type can be thrown as an exception
    - Classes are often trivial
- Catch Block (exception handler) executes when an error is thrown from the Try Block
    - Can have multiple catch blocks to handle different possible exceptions thrown; the first catch block that matches the parameter will be executed
    - For the case of inherited catch blocks, the error may be caught by a parent type (earlier than expected)
    -

try-catch block Example:

```
try{
  \\CODE BLOCK
  throw someException();
}
catch(someException){
}
```

Chapter 17.1 Templates

Type Parameter: the parameter of the template (i.e., “Item” for <class Item>)

Algorithm Abstraction: express general algorithms not specific to particular data types of parameters

Chapter 18.1 Iterators

Using Keyword: using defines the namespace that is to be worked with

Chapter 18.2 Containers

Set: stores elements without repetition; order is specified in some way

Map: stores elements in key-value pairs; keys must be unique