Unit 2 Study Guide and Review: Dealing with Data and Compound Types

2a. List and use the operators in the C++ language

1. What are the operators and the order of precedence?

Precedence	Operator	Description	Associativity
1 **highest**	`::`	Scope resolution (C++ only)	None
2	`++`	Postfix increment	Left-to-right
	`--`	Postfix decrement	
	`()`	Function call	
	`[]`	Array subscripting	
	`.`	Element selection by reference	
	`->`	Element selection through pointer	
	`typeid()`	Run-time type information (C++ only) (see typeid)	
	`const_cast`	Type cast (C++ only) (see const_cast)	
	`dynamic_cast`	Type cast (C++ only) (see dynamic cast)	
	`reinterpret_cast`	Type cast (C++ only) (see reinterpret_cast)	
	`static_cast`	Type cast (C++ only) (see static_cast)	

3	`++`
	
	`--`
	
	`+`
	
	`-`
	
	`!`
	
	`~`
	
	`(`*`type`*`)`
	
	`*`
	
	`&`
	
	`sizeof`
	
	`new`, `new[]`
	
	`delete`, `delete[]`
		Prefix increment
		
		Prefix decrement
		
		Unary plus
		
		Unary minus
		
		Logical NOT
		
		Bitwise NOT (One's Complement)
		
		Type cast
		
		Indirection (dereference)
		
		Address-of
		
		Size-of
		
		Dynamic memory allocation (C++ only)
		
		Dynamic memory deallocation (C++ only)
			Right-to-left

4	`.*`
	
	`->*`
		Pointer to member (C++ only)
		
		Pointer to member (C++ only)
			Left-to-right

5	`*`
	
	`/`
	
	`%`
		Multiplication
		
		Division
		
		Modulo (remainder)
			Left-to-right

6	`+`
	
	`-`
		Addition
		
		Subtraction
			Left-to-right

7	`<<`	Bitwise left shift	Left-to-right

	>>	Bitwise right shift	
8	< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left-to-right
9	== !=	Equal to Not equal to	Left-to-right
10	&	Bitwise AND	Left-to-right
11	^	Bitwise XOR (exclusive or)	Left-to-right
12	|	Bitwise OR (inclusive or)	Left-to-right
13	&&	Logical AND	Left-to-right
14	||	Logical OR	Left-to-right
15	?:	Ternary conditional (see ?:)	Right-to-left
16	= += -= *= /= %= <<= >>=	Direct assignment Assignment by sum Assignment by difference Assignment by product Assignment by quotient Assignment by remainder Assignment by bitwise left shift Assignment by bitwise right shift	Right-to-left

	`&=`
	
	`^=`
	
	`|=`
		Assignment by bitwise AND
		
		Assignment by bitwise XOR
		
		Assignment by bitwise OR	

17	`throw`	Throw operator (exceptions throwing, C++ only)	Right-to-left
18 **lowest**	`,`	Comma	

Operators are an essential component of computer programming, as it allows the program to make decisions. Understanding the order of precedence allows us to know how a program will interpret instructions and carry out these instructions. For a comprehensive list of operators, review this article.

2b. Define and use arrays, struct, unions, and enumerations

1. What is the purpose of a struct and how is it used?

A struct is a user-defined type that is used to create an array with multiple data types stored in a single element. To create a struct, use the following format:

```
typedef struct person_s
{
    char *name;
    int age;
} person_t;
```

(`person_t` is the name of the struct.)
To create an instance of the struct, you can declare it in main:

```
person_t molly;
```

To review structs, review this page.

2. What is the purpose of an enumeration type and how is it used?

An enum is a user-defined type that consists of a set of named constants. It essentially represents list of possible values. Only these values listed in the enum list are considered appropriate values.

```
typedef enum { GREET, ASK_OK, ASK_STUDY, CELEBRATE } action_type;
```
You can review enumeration types by reading this page

3. What is the purpose of a union type and how is it used?

Unions are similar to structures but only one member within the union can be used at a time, due to it having a shared memory size for all members.
Unions are used when just one condition will be applied and only one variable is required.
Review this page for more information on the use of a union.

4.What is the purpose of an array and how is it used?

Arrays are a container that holds a sequence of content of the same data type. To declare an array, you provide a datatype, name and size.
```
char a[50];
```
After the array has been declared, you can refer to the array by referencing its index. To review arrays, review Chapter 4 of Programming in C++.

2c. Use pointers

1. How do you declare a pointer?

A pointer is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is –

```
type *var-name;
```

Here, type is the pointer's base type; it must be a valid C++ type and var-name is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer.

2. How do you de-reference a pointer?

An interesting property of pointers is that they can be used to access the variable they point to directly. This is done by preceding the pointer name with the *dereference operator* (*). The operator itself can be read as "value pointed to by".

Therefore, following with the values of the previous example, the following statement:

```
baz = *foo;
```

This could be read as: "baz equal to value pointed to by foo", and the statement would actually assign the value 25 to baz.

3. How do you pass a pointer?

When declaring the parameters for your function you use type* varName. So

returnType functionName (type *varName){

// statements

}

When calling the function in main add the address operator before arguments. So in main call it with functionName(&varName);

```
// C++ program to swap two numbers using

// pass by pointer.

#include <iostream>

using namespace std;


void swap(int* x, int* y)

{
```

```
    int z = *x;
    *x = *y;
    *y = z;
}

int main()
{
    int a = 45, b = 35;
    cout << "Before Swap\n";
    cout << "a = " << a << " b = " << b << "\n";

    swap(&a, &b);

    cout << "After Swap with pass by pointer\n";
    cout << "a = " << a << " b = " << b << "\n";
}
```

Pointers are a way to store data. Instead of referring to the actual value of a dataset, you refer to its address. You then access the value by referring to the pointer to value. For additional information on pointers, refer to this page.

2d. Use the functions of the string class

1. How do you use string functions to manipulate string variables?

Requires the #include <string> preprocessor directive.

Part of the std namespace, and therefore also requires the using namespace std ; preprocessor directive.

Here are some of the more common string methods:

+, +=	Joins / Concatenates strings
<<	Output Stream
==, !=, < <=, >, >=	Compare strings
>>	Input Stream
myString.append()	Append string/characters
myString.assign("string-goes-here")	Same as: myString = "string-goes-here" ;
myString.at(pos)	Returns character at pos
myString.begin()	Returns iterator positioned at first character of string. Start pointer
myString.c_str()	Returns C string pointer. Null terminated
myString.capacity()	Returns size of allocated storage for string
myString.compare(string 2)	Returns 0 if strings compare equal
constructors	string strOne() = "string goes here" ; string strTwo("string goes here") ; string strThree ; strThree = "Anorak" ; string strFour (strOne) ; char *testStr = "testing" ; string strFive(testStr) ; string strSix(testStr, 4) ; //first 4 characters of testStr

	string strSeven(strOne, pos, len) ; //copy strOne into StrSeven at pos, for len
	string strEight(len, 'x') ; //len number of 'x' characters

myString.copy(ptrArr, len, pos)	Copies substring of len from myString from pos, into array at ptrArr Doesn not append a NULL to ptrArr, needs to be carried out separately
myString.data()	Returns C string pointer. NOT NULL terminated
myString.empty()	Returns 1 if empty string, or 0 if not empty string
myString.end()	Returns iterator positioned at last character of string. End pointer
myString.erase()	Erases the string. No parameters required
myString.find(string2)	Returns position of FIRST occurrence of string2
myString.find_first_not_of ("char/string")	Returns position of FIRST char/string NOT defined in the "char/string" parameter list
myString.find_first_of("ch ar/string")	Returns position of FIRST char/string defined in the "char/string" parameter list
myString.find_last_not_of ("char/string")	Returns position of LAST char/string NOT defined in the "char/string" parameter list
myString.find_last_of("ch ar/string")	Returns position of LAST char/string defined in the "char/string" parameter list
getline(input_stream, string, delim)	Extracts chars from input stream and stores in string until .max_size(), EOF or delim
myString.insert(pos, string)	Insert a copy of string into myString at position pos

myString.length()	Returns int length of string. No parameters required
myString.max_size()	Returns integer defining maximum size of string possible
myString.rbegin()	reverse begin
myString.rend()	reverse end
myString.rfind(string2)	Returns position of LAST occurrence of string2
myString.replace(pos, len, string2)	Replace len characters in myString at pos, with string2
myString.resize(len, 'char')	Resizes myString to len. Truncates if less. Expands if smaller. If 'character' is missing, filled with blanks
myString.rfind("char/string")	Returns position of LAST char/string defined in the "char/string" parameter list
myString.size()	Returns int size of string. Same as above. No parameters required
myString.substr(pos,len)	Returns substring within myString from pos, for len characters
myString.swap(string2)	Swaps string2 with myString

This article lists many of the string functions that can be used. Make sure you understand how to read and interpret this information and correctly apply the information. The substring function, in particular, is very important. The substring retrieves a portion of the string based on the starting index and the length of the the substring.

myString.substr(pos,len)	Returns substring within myString from position, for length characters

Unit 2 Vocabulary

This vocabulary list includes terms that might help you with the review items above and some terms you should be familiar with to be successful in completing the final exam for the course.
Try to think of the reason why each term is included.

- Array
    - An **array** is a collection of elements of the same type placed in contiguous memory locations that can be individually referenced by using an index to a unique identifier.
- Data structure
    - A **data structure** is a group of **data** elements grouped together under one name. These **data** elements, known as members, can have different types and different lengths.
- De-reference
    - **Dereferencing** a pointer means getting the value that is stored in the memory location pointed by the pointer. The operator * is used to do this, and is called the **dereferencing** operator.
- Enumeration
    - An enumeration is a distinct type whose value is restricted to a range of values, which may include several explicitly named constants ("enumerators"). The values of the constants are values of an integral type known as the underlying type of the enumeration.
- Operator
    - An **operator** is a symbol that tells the compiler to perform specific mathematical or logical manipulations.
- Order of precedence
    - Operator **precedence** specifies the **order** of operations in expressions that contain more than one operator. Operator associativity specifies whether, in an expression that contains multiple operators with the same **precedence**, an operand is grouped with the one on its left or the one on its right.
- Pointer
    - a **pointer** is a programming language object that stores the memory address of another value located in computer memory. A pointer *references* a location in memory, and obtaining the value stored at that location is known as *dereferencing* the pointer. As an analogy, a page number in a book's index could be considered a pointer to the corresponding page; dereferencing such a pointer would be done by flipping to the page with the given page number and reading the text found on that page.
- Self-reference

    - Self-reference occurs in reflection, where a program can read or modify its own instructions like any other data.[2] Numerous programming languages support reflection to some extent with varying degrees of expressiveness. Additionally, self-reference is seen in recursion (related to the mathematical recurrence relation) in functional programming, where a code structure refers back to itself during computation.
- Struct
    - An **aggregate data type** is a data type that groups multiple individual variables together. One of the simplest aggregate data types is the struct. A **struct** (short for structure) allows us to group variables of mixed data types together into a single unit.
- Typedef
    - The *typedef declaration* provides a way to declare an identifier as a type alias, to be used to replace a possibly complex type name
- Union
    - A **union** is a user-**defined** type in which all members share the same memory location. This **means** that at any given time a **union** can contain no more than one object from its list of members. It also **means** that no matter how many members a **union** has, it always uses only enough memory to store the largest member.