1) Look at the chart in Section 3.1 What additional datatypes are listed here that we did not reference yet in class in our list of built-in datatypes.

enum, array, struct, union, class, pointer, reference, long double

2) How do we write Avogadro's Constant as a C++ literal value? If you do not know Avogadro's constant by heart, it's available through the Encyclopedia Britannica. It is an important value in Chemistry and Physics.

```
const double avogadro = 6.02214076E23;
```

3) Read the Software Engineering Tip on Page 99. Why is it a good idea to use named constants in place of numeric literals?

Two main reasons:
1) Named constants make the program easier to read compared to literal values.
2) Named constants make it easier to change the value in a single spot in the program rather than searching in multiple places for all the values to change.

4) What is the difference between the unary – operator and the binary – operator? (The two minus operators).

The unary - operates on one value (negating a value) while the binary - operates on two values (subtracting one value from another).

5) What is the difference between the two groups of statements below? If you are unsure, put them both in a program and print the result of x and y.

```
int  x = 3;
int  y = x++ + 3;
```

```
int  x = 3;
int  y = ++x + 3;
```

The prefix notation (++x) increments x first and then uses the new value. The postfix notation (x++) uses the value of x first and then increments x.

x = 4, y = 6

x = 4, y = 7

6) List the order of precedence table for the arithmetic operators:

highest order: unary +, unary –
medium order: %, /, * (from left to right)
lowest order: binary +, binary – (from left to right)

7) What is the difference between type coercion and type casting? Give an example c++ statement of each kind.

Type coercion is an implicit assignment of a value of one type to a variable of a different type. The value must be coerced into the new type.
Type casting is an explicit changing of the type of a value.

Type coercion: int x = 4.5;
Type casting: int x = (int) 4.5;

8) Explain what each of the following string operations does.

```
string word = "Happy";
```

(a) `cout << word.length() << endl;`
prints length of string (5)
(b) `cout << word.find("p") << endl;`
prints first location of "p" in word (2)
(c) `cout << word.substr(3,2) << endl;`
prints substring starting at index 3, going for 2 chars ("py")
(d) `cout << word.at(0) << endl;`
prints character at index 0 ('H')

1) cin and cout are both objects. What type are they and where are they declared?
cin is of type istream while cout is of type ostream. They are declared in the iostream header file. These are "streams" that can be thought of as streams of characters coming into or out of our program. The operating system connects these streams to useful computer hardware (like typed input from the keyboard or characters printed to the screen).

insertion operator (<<) No difference between cin >> int1 >> int2; and cin >> int1;
extraction operator (>>) cin >> int2;

2) Explain the purpose of the "reading marker" and its role in parsing an input stream.
The reading marker keeps track of the current place in the stream where we are reading. When we process the next input with cin, the marker tells us where in the input stream to look for the appropriate value for the use in cin.

3) What role does the ignore function play in parsing input streams?
The ignore function allows us to skip over specified input in the stream to look further in the stream for values that we want.

4) When we normally read a string value using cin we get the group of characters separated by spaces. How do we read an entire line of text into a single string variable?

When reading characters into a string variable, the >> operator skips leading whitespace characters. It then reads successive characters into the variable, stopping at the first trailing whitespace character (which is not consumed, but remains as the first character waiting in the input stream).

We use the getline() function which reads an entire line from the stream. It reads a sequence of characters until it encounters a newline.

Note that with getline, the newline character is consumed (but is not stored into the string variable).

5) Write a C++ program that opens a file called `temperatures.dat`. The program should read six temperatures (floats) from the file and output the average temperature. Write or copy & paste your solution here.

```
//=========================================================
// Flannery Currin
// read_data.cpp
// February 2025
// This program reads from a file called  temperatures.dat
// It will read 6 temperatures from the file and output
// the average temperature.
//=========================================================
#include <iostream>
#include <fstream>
using namespace std;

int main ( void )
{
  // variables to hold the temperature values
  float temp1;
  float temp2;
  float temp3;
  float temp4;
  float temp5;
  float temp6;

  // use an input stream to open temperatures.dat
```

```
  string filename = "temperatures.dat";
  ifstream temps;   // Declare the file streams we will use.
  temps.open(filename);  // Prepare each file for reading or
writing.

  // read the temperatures in and store them
  temps >> temp1;
  temps >> temp2;
  temps >> temp3;
  temps >> temp4;
  temps >> temp5;
  temps >> temp6;

  float sum = temp1 + temp2 + temp3 + temp4 + temp5 + temp6;
  float mean = sum / 6.0;

  cout << mean << endl;

  return 0;
}
```

6) Briefly explain the design philosophies of functional decomposition and object oriented programming (be specific about what each approach entails:

Functional decomposition is a design philosophy that breaks up large computation problems into smaller parts and then generally codes each of those smaller parts into separate functions. The main() function acts like a "general contractor" that calls and coordinates the "specialist" functions to do different parts of a job.

Object oriented programming is a design philosophy that encapsulates data and the operations on that data into an entity called an object. These objects maintain information about specific attributes (the object's state) and those attributes can be accessed or changed through special functions called methods.

FD is focused on the actions. OOP is focused on the data.

1) In the principle of *functional decomposition* explain the role of the *interface* vs the *encapsulation*.

Functional decomposition is a top-down design strategy where we take larger problems and break them up into smaller pieces but placing each piece in a function. We want functions to be tasks that are large enough to be "interesting" (we typically avoid functions that are one or two simple statements). We want our functions to be ONE SINGLE task (don't make a function that does two or three things).

The interface for a function describes WHAT it does. It is the external message to the users of the function – like a simple user manual. Encapsulation is the idea of hiding the details of HOW the function does its work. A user of a function typically cares about the interface (WHAT it does) but not its detailed implementation (the HOW part).

2) What is the difference between a function's *arguments* and its *parameters*?

The arguments are the expressions that are used in a function call. They provide the values that will get passed to the function.
The parameters are the identifiers (variable names) used in a function definition.

```
result = myFunction(x, 5);
// here x and 5 are the arguments in the function call

int myFunction( int n, int total )
{
     ....
}
// here n and total are the function parameters
```

3) Some people think it is ok to write function definitions without function declarations. In what specific situation is it impossible to write function definitions without using at least one function declaration?

The compiler starts at the top of our cpp file and scans downward. It must know about a function before it sees a function call. This is why we put function declarations at the top of our programs. You can "get away" without a function declaration if you put the function definition in your source file before the function call. So it might be tempting to put your function definitions at the top of your cpp file and the main function near the bottom.

But this is not considered good programming practice, especially for C++ programming. We put function declarations at the top, then we have our main function definition next and then following are all the other function definitions. The function declaration for main() is provided automatically by the compiler since every program we write must have a main function. That is why we don't declare main.

But there are at least two situations where we must have function declarations.
1) If we have a pair of functions which call each other. Consider function A which calls function B. And function B which calls function A. It would be impossible to put both A's defintion before B and also B's definition before A. So we must resort to at least one function declaration.
2) If we define a function in another source file then we must declare the function in the file where we are calling it. This is what happens with all the built-in functions we use like sqrt() and getline(). These function declarations are in the header files that we copy into our code

using the preprocessor. The definitions of these functions are in separate files and are combined with our code during the linking phase of compilation.

4) What is a *local variable* and why is it referred to as local? Answer both parts!

A local variable is declared and used within a block. Most often this block is a function. We call it local because its scope is restricted to the block it is declared in.

A parameter acts like a local variable to a function except that it gets an initial value assigned to it whenever the function is called.

5) What is a good style choice for naming functions?

Since most functions do something useful, it is often best to pick a verb for a function name. For example:

decode(), computeMean(), cleanData()

If we have a boolean function (with a boolean return value), then we often have a "property name" like this:

isEven(), isComplete(), isCorrect()

If we have a void function, which are often used to display some result, then we might pick names like (Adding the verb Print)

printTable(), showChart(), drawFigure()

When picking a name for a void function, try to find one that sounds like a command to the computer.

6) Explain the difference between a parameter that is passed-by-value from one that is passed-by-reference. Explain both the syntactic differences and also the semantic ones.

Syntactically, the difference is an ampersand:

```
int  myFunction  (int  x )    <- pass by value
int  myFunction  (int& x )    <- pass by reference
```

Semantically, in a pass-by-value situation, the argument and the parameter occupy two different memory locations. At the time of the function call, the value of the argument is passed and copied into the memory location of the parameter. If the function changes the

value of its parameter, then only the local copy of memory is changed; the memory location of the argument is not affected. As you can see, using value parameters helps us avoid unintentional changes to arguments.

In a pass-by-reference situation, both the argument and the parameter refer to the same piece of memory. Thus any changes to the function parameter will result in a change to the argument variable. This kind of thing can be helpful in two situations:
1) Where you intentionally want to change the argument. This is how we can overcome the problem of only one return variable. If we pass-by-reference, then we can, in effect, have each of our & variables act like a return channel.
2) If the thing you are passing in is very large, then pass-by-reference saves memory so that we don't have two copies of the large data item.

7) What is the purpose of the *assert()* statement in a c++ program? Also give an example of how it might be used in the following function:

```
#include <cassert>
int compute ( int array[], int size )
{
      // we expect size to have a value > 0
      ...
      assert(size>0);

}
```

The assert statement can be used to verify an assumption about a variable value. If the assumption is not correct, the assert statement will cause the program to stop executing and report an error about the invalid value.

#define NDEBUG

Limitation
As useful as the assert function is, it has two limitations. First, the argument to the function must be expressed as a C++ logical expression.
The second limitation is that the assert function is appropriate only for testing a program under development. A production program (one that has been released to the public) must give helpful error messages to the user.

1) Define the *scope* of a variable and list the three different kinds of variable scope.

The scope of a variable is the part of the program source code where the variable is visible – where it can be read and assigned.

1) Class scope: a class variable can be accessed inside any class function.

2) Local scope: a variable is local to a particular block of code, usually a function.
3) Global scope: a variable is accessible throughout the whole program.

4) Namespace scope: The scope of an identifier declared in a namespace definition extends from the point of declaration to the end of the namespace body, and its scope includes the scope of any using directive specifying that namespace.

There is also namespace scope: the scope of an identifier declared in a namespace definition includes the point of declaration to the end of the namespace body and the scope of any using directive specifying that namespace.

2) List the five scope rules below.

1. Function names have global scope (function definitions cannot be nested).
2. A function parameter's scope is identical to the scope of a local variable declared in the outermost block of the function body.
3. A global variable's (or constant's) scope extends from its declaration to the end of the file, except as noted in rule 5.
4. A local variable's (or constant's) scope extends from its declaration to the end of the block in which it is declared, including any nested blocks, except as noted in rule 5.
5. An identifier's scope does not include any nested block that contains a locally declared identifier with the same name (local identifiers have name precedence).

3) Describe how a static variable in a function works.

Normally, when a block of code stops executing (for example, when we return from a function), any variables that are local within that block cease to exist. Their memory contents are no longer retained. Declaring a local variable to be static causes that variable to remain permanently in memory for the duration of our program. Furthermore, a static variable retains a value between instances of when its scope is active. This can be helpful if we want a local variable to "remember" its value between function calls.

4) Define *stub* and *driver* and explain their role in software testing.

When we are building a larger program, we often want to build it in pieces and test each piece as we go along. This is better than coding a huge bit of code and then testing it all at once.

To facilitate this, we often define a function with a stub – a short piece of code that will compile and do something reasonable. For example, we might temporarily define isEven() like this:

```
bool   isEven ( int array[], int size )
{
     return true;
```

```
}
```

This is not the correct intended execution of this function, but we can insert this stub code temporarily so that our program compiles and we can test the operation of other parts of our code. When we have time, we will come back and remove the stub code and replace it with the intended logic.

Similarly, we might have a function that we want to test so we can write a main() function (often in a different file) that is designed only to test that function. This version of main() is called a driver. We can then copy our working and tested function into our intended program where a different main exists that does the intended functionality of our program.

Driver: A simple main function that is used to call a function being tested. The use of a driver permits direct control of the testing process.

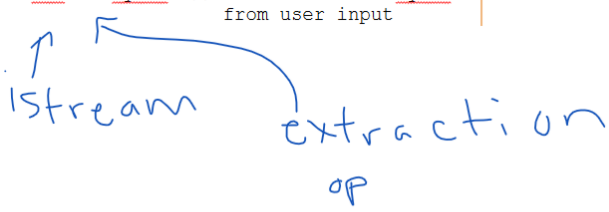
```
int myInt; // declaration only
```


```
bool someBool = true; // combo declaration &
                      // assignment -- initialization
```

```
myInt = 5; // assignment only
```


```
#include <iostream>
...
cout << myInt << endl; // print 5
```


```
#include <iostream>
...
cin >> myInt; // store value in myInt
              from user input
```




```
if (booleanExpression)
{
  ...
}
```

- Logical operators
  - Python: not, and, or
  - C++: !, &&, ||
- Arithmetic operators
  - +, -
  - *, /, %
    - Why not //
    - Behavior of / based on type of operands
    - Automatic type coercion
    - int / int --> int (5 / 2 --> 2)
    - float / float --> float (5.0 / 2.0 --> 2.5)
    - float / int --> float (5.0 / 2 --> 2.5)
    - int / float --> float (5 / 2.0 --> 2.5) <- Automatic type coercion

We can cast to types explicitly:

- `int x = 10;`
- `float y = 5.8;`
  - `x / int(y)`
  - `X / (int)y`

We can force narrowing based on type of variable result is stored in:

- `int x = 10;`
- `float y = 5.8;`
- `int z = x / y; // z == 1; the fractional part is truncated!`

- Assignment operators
  - =
    - We can chain assignments!
    - x = y = z;
      - 1st: y is assigned the value of z
      - 2nd: righthand = returns the value of y
      - 3rd: x is assigned the value of y (which is the value of z)
- Combo assignment & arithmetic operators:
  - +=
    - x= 10;
    - y= 5;
    - x+= y;
      - Changes the value of x to 15
  - -=
  - *=
  - /=
  - ++
    - x= 10;
    - y= 5;
    - x++; // postfix notation
      - Returns value of x
      - Changes value of x to 11
    - ++y; // prefix notation
      - Changes value of y to 6
      - Returns value of y
  - --
- X = 9;
- Y = X++;
- // what's the value of X? 10
- // what's the value of Y? 9

- X = 9;
- Y = ++X;
- // what's the value of X? 10
- // what's the value of Y? 10

- String methods
  - s.length()
    - Parameters? Not needed because we call with a specific string object
    - Returns: number of characters in string integral number (special type – string::size_type)
  - s.find()
    - Parameters?
      - Yes, takes a
      - target string
    - Returns: index of start of substring if found; string::npos value otherwise
  - s.substr()
    - Parameters: 2 ints
      - Start: Start index
      - Len: The **length** of the substring
    - Return: string representing the Len characters starting at Start in s
  - s.at()
    - Parameter: 1 int
      - Index/position
    - Return: char at specified index in s

- Other helpful functions
  - toupper()
    - Takes a character
    - Result needs to be cast back to a char
  - tolower()
  - to_string()
    - Defined for built-in and many user-defined types
    - Converts from original type to string
  - stoi()
    - Converts a string to an int
  - stof()
    - Converts a string to a float

:: Scope resolution operator – says where an identifier is defined

- While loops
  - while ([boolean expression]){
    ...
    }
- For loops
  - for (int i = 0; i < 10; i++){
    ...
    }

For (Declaration & initialization; Conditional check;
How we move toward condition being false)

- User input we've already seen:
  - cin
    - istream object included in <iostream>
    - Extraction operator: >>
      - stream >> var;
      - Can be used with other istreams!
      - Doesn't include any leading whitespace
      - Stops before trailing whitespace
      - Other types:
        - Can read ints, floats
        - Stops at first inappropriate character for type of variable

- Other ways to read from an istream:
  - cin.get(var)
    - Reads a single char, including whitespace characters.
    - Inputs the next character waiting in the stream—even if it is a whitespace character—and store it into the variable somechar.
  - cin.ignore(num, character)
    - Skips over the next num characters or until we reach character, whichever comes FIRST
  - getline(cin, someString)
    - Reads everything until a newline character
    - Stores that in someString
    - Consumes the newline character!
- Reading from/writing to files:
  - #include <fstream>
  - Declare file streams
    - Ifstream // represents a stream of characters coming from an input file.
    - Ofstream // represents a stream of characters going to an output file.
  - Open the files
  - When we use istream/ostream operators or methods, we need to specify which stream object we are reading from or writing to.
  - With these data types, you cannot read from and write to the same file.
  - In C++, however, a file is automatically closed when program control leaves the block (compound statement) in which the stream variable is declared.
  - When control leaves the block, a special function associated with each stream object, called a

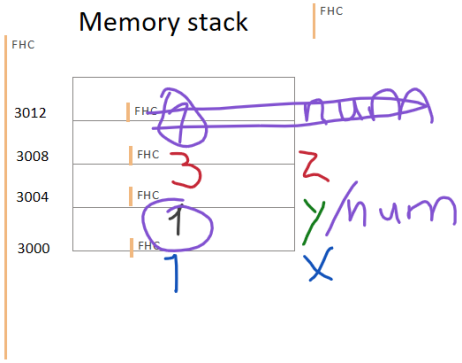
destructor, is implicitly called to close the file.

- What if something goes wrong when we try to read from or write to a file?
  - Stream objects can enter a FAIL STATE
  - Won't give an error message
  - Doesn't stop the program
  - Program will ignore any functions or operators reading to or writing from that file
  - Another way to make a stream enter the fail state is to try to open an input file that doesn't exist.
  - We can test for this!
    - if (inStream){...} //only do this if inStream is not in a fail state
    - if (!inStream){...} //only do this if inStream is in a fail state
    - while(inStream){...} //do this until inStream is in a fail state
- Function prototypes
  - Declarations
  - string determineHopSuccess(int, int, int, int, int);

    return type function name types of params

- Before we can call a function, the compiler needs to have seen:
  - Declaration/prototype
  - OR Definition

```
int main(){
    int x = 1;
    int y;
    int z = 3;
    y = 1;
    increment(y);
    return 0;
}
```



Pass by Value vs. Pass by Reference

Pass by value:

- Makes a copy of the argument, stores the copy in a new location
- Any change the function makes to its copy of the parameter does not transfer to the original copy

Pass by reference:

- Passes along the address in memory of the argument
- Any change the function makes to the parameter is visible/affects the original

Local vs. Global Scope

- Scope: where in our program we can refer to an identifier
- Local Scope
  - Identifiers declared within a block (function definition, loop, conditional)
  - Extends to the end of the block from wherever it was declared
- Global Scope
  - Identifiers declared outside of any block
  - Extends to end of file/program from wherever it was declared

Scope Rules

1. Function names have global scope (function definitions cannot be nested).
2. A function parameter's scope == the scope of a local variable declared in the outermost block of the function body.
3. A global variable's (or constant's) scope extends from its declaration to the end of the file, except as noted in rule 5.
4. A local variable's (or constant's) scope extends from its declaration to the end of the block in which it is declared, including any nested blocks, except as noted in rule 5.
5. An identifier's scope does not include any nested block that contains a locally declared identifier with the same name (local identifiers have name precedence).

Using Namespace Identifiers (Identifiers declared within the namespace body cannot be accessed outside the body except by using one of three methods.)

Namespace: a mechanism by which the programmer can create a named scope.

- Write a using directive (locally or globally):
  - using namespace std;
  - alpha = abs(beta);
- Write a using declaration:
  - using std::abs;
  - alpha = abs(beta);

  These using declarations allow the identifiers abs to be used throughout the body of main as synonyms for the longer std::abs.

- Use a qualified name consisting of the namespace, the scope resolution operator :: and the desired the identifier
  - alpha = std::abs(beta);

Lifetime of a Variable

- Lifetime: The time during program execution in which a variable/identifier actually has memory allocated to it
- Local automatic identifiers(= Automatic variable)
  - Memory gets allocated when control enters the function
  - Local variables are "alive" as long as the function is executing
  - Memory is deallocated for those identifiers when control exits the function
- Global identifiers(= Static variable)
  - Alive for as long as the program is running, after being declared

Static Variables

- To make a local variable's lifetime persist between calls to a function, you must use the reserved word `static` in its declaration

```
int  popularSquare(int  n)
{
  static int timesCalled = 0; // Initialized only once
  int result =  n * n; // Initialized each time

  timesCalled++;
  cout  << "Call # "  << timesCalled  << endl;
  return  result;
}
```