When to use struct and when to use enums, topics on class that was covered before break.
Difference between structured datatype and simple datatype
Project 7 (what does constuctors, desctructers do)

1) What kind of information is available in #include<climits>?

The climits header file contains upper and lower limit values for different numeric datatypes. This can be useful to test or assign a numeric variable to the largest or smallest value possible.

2) What does the sizeof( ) function do?

The sizeof() function returns the number of bytes that a datatype or variable uses. There are a few useful circumstances where you will want to use this in a program.

3) What does typedef allow you to do?

typedef creates a synonym of an existing datatype. Note that it does not create a new datatype but rather creates a new name for an existing datatype. This can be useful when the datatype name is long and cumbersome, often with structs or multidimensional arrays.

4) What does enum allow you to do? Give an example.

enum allows you to create a synonym datatype for counting values.

```
enum PEOPLE { ALICE, BOB, CAROL, DAVE };
PEOPLE person1 = ALICE;
```

In reality, PEOPLE is a synonym for int and ALICE=0, BOB=1, .. automatically. It allows the programmer to create symbolic names for ordinal values.

5) What distinguishes a structured datatype from a simple datatype? List the four different kinds of structured datatypes.

A simple datatype is atomic, meaning it cannot be decomposed into smaller values. int and float are both examples.

A structured datatype is comprised of multiple values which can either be simple datatypes or structured datatypes. Examples of structured datatypes are arrays, structs, unions, and classes. Examples:

```
    int   x;                    // simple
    int   array[10];            // structured
    struct Personal p;          // structured
```

6) Consider the C++ code below.

```
const int NUM_QUIZZES = 10;

struct StudentRecord
{
     string    name;
     int       quizzes[NUM_QUIZZES];
     int       final_exam;
     float     average;
};

StudentRecord  s1;
s1.name = "Sally";
```

(a) Write a statement below which will assign the grade of 80 to Sally's third quiz grade:

```
s1.quizzes[2] = 80;
```

Notice we use the . to access quizzes and then index into the array.

(b) Write a statement below which will assign the grade of 90 to Sally's final exam score:

```
s1.final_exam = 90;
```

(c) Write a series of statements below which will compute Sally's overall course average and assign it to "average" in her struct record as

50% of the average quiz grade (over all 10 quizzes) + 50% of the final exam score

```
// first compute the average of the quiz grades
float quiz_average = 0;
for ( int i = 0; i < NUM_QUIZZES; i++ )
   quiz_average += s1.quizzes[i];
quiz_average = quiz_average / NUM_QUIZZES;
// notice that quiz_average is a float so that proper floating
// point division is invoked in the above statement.

// now compute the overall class average for Sally
s1.average = 0.5 * s1.final_exam + 0.5 * quiz_average;
```

1) Define Abstract Data Type (ADT).

An abstract data type is an implementation independent specification for a datatype. It is like a blueprint for a datatype.

2) Which is most important in the ADT: *how* or *what*? Explain.

In an ADT the WHAT question is most important. The specification should provide information for *what* the datatype does. The *how* it works is hidden and is dependent on the implementation of the datatype.

Note that a single ADT might have different implementations on different machines, in different programming language, or even in the same programming language using different data structures. For example, there is more than one way to build a LIST ADT in c++ each with its own advantages and disadvantages. And yet both implementations will provide the exact same functionality to the user which is specified by the ADT design.

3) What role does the *data representation* play in an ADT implementation?

The data representation is the specific details on how the data in the ADT is actually stored.

4) There are generally five categories of methods associated with an ADT. List the five different kinds of methods and give a brief description of each kind (see blue box on Page 682).

There are five different kinds of methods associated with most ADTs.
Constructors allow you to create and properly initialize a new variable of that type.
Destructors allow you to properly delete a variable. This is most important when the ADT creates dynamic memory that should be deleted to prevent a memory leak.
Observers usually return some kind of information about the value(s) in the ADT. They generally do not change the variable's value.
Transformers generally allow the user to change the value in the ADT variable. They are usually carefully constructed so as to permit the user to properly change the value and not allow erroneous kinds of changes.
Iterators allow you to access all the values in an ADT that might have multiple values, such as in a list. These can be either observers or transformers, but the key idea is that they access all the values instead of just one part of a value.

5) How is a C++ class different from an ADT? How are they related?

An ADT is an abstract design, separate from an actual implementation.
A class is a specific implementation of an ADT in some programming language.

6) How is a C++ class different from a C struct?

A c struct allows you to group data members together in a single datatype. Structs do not have methods built-in to them. A c++ class has data members like a struct but it also contains the methods used to operate on those data members.

7) What do the *private* and *public* access modifiers do to class methods and data? When do we generally use public and when do we generally use private in creating a C++ class?

The private and public access modifiers allow and prevent users of your class to access data members and/or method members of your class. Generally most data is private and most methods are public, but there are exceptions to both.

8) How is a C++ class different from an object of that class? How are they related?

As stated in answer 6, an ADT is an abstract design, separate from an actual implementation.
A class is a specific implementation of an ADT in some programming language.
An object is a specific instance of that class in an actual program.

ADT describes the abstract interface.
Class provides the implementation.
Object is an actual variable of that new type.

9) Discuss how *information hiding* plays a role in class design and implementation?

Information hiding is an important aspect of class designs. You want to make public the *what* aspect of your datatype – what does your datatype do? These are usually public methods that operate on your datatype. The *how* is generally hidden from the user of your datatype by keeping the actual data representation as private data members. This is an important part of abstraction and encapsulation and is the main concept of object oriented programming!

10) What is the difference between the *specifications file* and the *implementation file* in class design? Why have two different files for a single class?

It is common to split up the class implementation into two files. The class specification (class declaration) is generally put in a header file that the user can #include into their program. And the user can read this file too. The implementation (the class method definitions) are generally put into a separate file (a cpp file). These are often compiled separately by the class designers and then only the object code is provided to the user so they can link to it during the last step when they compile their own programs. This way, clients do not need to be concerned about implementation details as long as underlying implementation changes do not alter the specifications.

Polymorphism

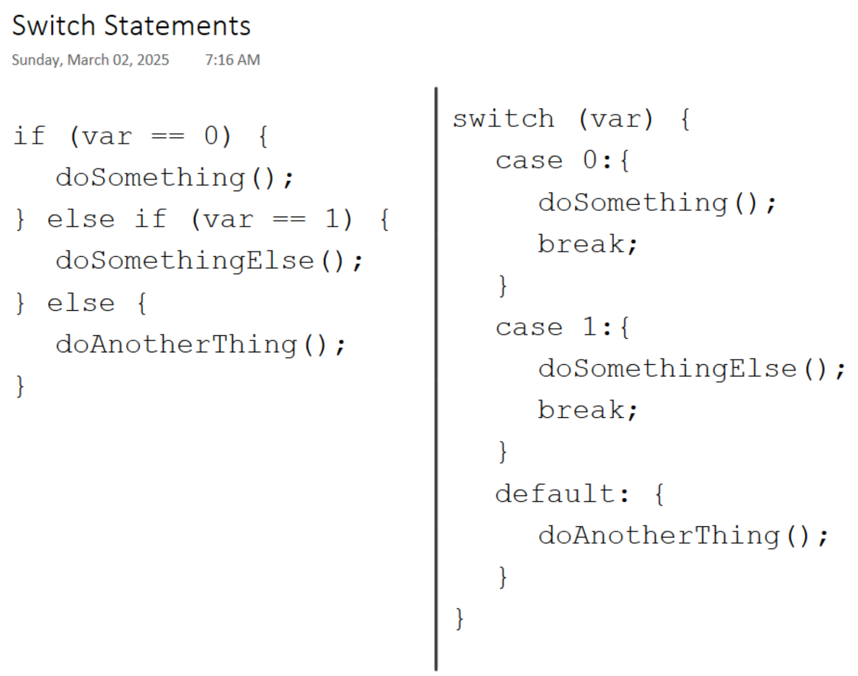
- We want to do the same kind of action (operation) with multiple types of things
- Function overloading
  - We can write multiple versions of the same function that take different types of parameters
  - void printArray(const int a[], int n);
  - void printArray(const float a[], int n);
  - void printArray(const char a[], int n);
  - …
  - Need to do this for every type we want to support!

- Template functions
  - Blueprint for generating functions on demand that will support different types
  - template <typename T>
  - void printArray(const T a[], int n);
  - …
  - template <typename T>
  - void printArray(const T a[], int n){ ...
  - }

Simple vs. Structured Types

- Simple types
  - Atomic value
  - Can't be broken down into pieces
  - char
  - integer: int, short, long
  - float/double
- Enumeration: enum
  - User-defined
  - One atomic value at a time
  - We list/enumerate a set of values
    - Have an ordering
    - Map to integer values
  - Why use these?
    - To represent a concept/map names to meanings
    - Makes state clearer
    - Readability
- Structured types
  - Can be broken into pieces
  - Arrays
    - Collection of elements
    - All elements need to be the same type

- Structs/records
    - Collection of elements (members, fields)
    - Each element can be a different type
    - Bundle together pieces of related information
    - Each piece makes up part of a whole
    - To get/access a member/field: use dot notation
        - record.member
        - date.month
            - Gets the month field of date



Switch Statements

Sunday, March 02, 2025 7:16 AM

```
if (var == 0) {
   doSomething();
} else if (var == 1) {
   doSomethingElse();
} else {
   doAnotherThing();
}
```

```
switch (var) {
   case 0:{
      doSomething();
      break;
   }
   case 1:{
      doSomethingElse();
      break;
   }
   default: {
      doAnotherThing();
   }
}
```

```
struct Date{
  int day;
  Month month;
  int year;
};
```

An int day
AND
A Month month
AND
An int year

```
Date today;
  today.day = 3;
  today.month = MAR;
  today.year = 2025;
```

ADTs

- Abstract Data Types (ADTs)
    - Define types independently of any programming language/specific implementation
    - Tell us:
        - What does the type represent?
        - What can instances of the type do? How do they behave?
- Date ADT:
    - Data members:
        - Day (1-31)
        - Month (Jan-Dec)
        - Year (0-whatever language-specific integer max)
    - Methods (function members):
        - Get (accessors/observers)
        - Change/update (transformers/mutators)

C++ Classes

- A class is used as a blueprint to create **objects**
    - Each object has **state** and behaves based on its class definition
    - State: the values of the data members in an object – unique to that object (changing the year of date1 should not change the year of date2)
- A class defines the **private** and **public** data members and methods (function members) required to implement an ADT
- Private members:
    - No external access (can't be accessed outside of the class definition)
        - Client code can't access
    - Data members are usually private
- Public members:
    - Can be accessed in client code, outside the class definition
    - Often we make public methods that define what objects can do in client code
- We often split a class definition into **specification** (.h) and **implementation** (.cpp) files

Class Methods

- Constructors
    - Initialize new instances (objects) of a class
    - Called when we create an object (declare it)
    - Set initial values for data members
    - Kinds of constructors:
        - Default constructor:
            - No parameters
            - Sets initial values for data members
        - Standard:
            - Takes parameters, sets initial values for data members using those parameters
        - Copy constructors:
            - Takes an object of the class and creates a new object with copies of the data members as values
- Destructors
    - Called when we destroy an object
    - We don't want objects to have unfinished business when they are destroyed
    - Sometimes we need to clean up
- Accessors/Observers/Getters
    - Functions/methods that let clients see the values of an object's data members
    - Tells the client something about the object's state
    - Let us look through windows into the object's state, but we can't go into the house and change anything
- Transformers/Mutators/Setters
    - Functions/methods that let clients change an object's state (change data members)
    - This is where it's important to restrict the types of changes allowed

Operator Overloading

Friday, March 14, 2025 9:40 AM

- * more on this & exceptions can be found in Chapter 16 of the book
- What is really happening when we use an operator?

x = y + z;

Left-hand operand
Calling operand/object

Right-hand operand
Argument/parameter

Operator returns a new value of x's type
So it can be assigned to x

friend special keyword

- Lets us give another class access to our class' private members and define how it works with objects of our class

Assignment operators

Thursday, March 27, 2025 4:26 PM

- 3 things you gotta do as an assignment operator
- r1 = r2
- 1: act like a destructor
  - Clean up whatever was previously referenced by the lefthand operand (e.g., get rid of what used to be in r1)
  - For Complex/Rational -- do nothing
- 2: act like a copy constructor
  - Getting the values of the data members for righthand operand
  - Copying over values for each data member
  - Setting each data member for the lefthand operand
- 3: return *this
  - Return the object this is pointing to (e.g., r1 – the lefthand operand)
  - Support chaining: r1 = r2 = r3

Define an enumeration type called Level with 3 possible values: EASY, NORMAL, HARD.

Fill in the Player struct definition below which contains fields for score (a floating point number), lives (an integer), and level (a Level value).

```
struct Player
{









};
```

Then in the main function, show how you could set initial values for each field for a Player: player1 (you can pick whichever initial values you want).

```
int main(){









};
```

What does the sizeof() function do? Be specific.

What does typedef allow you to do?

What distinguishes a structured datatype from a simple datatype?

```
class Box
{
   public:
      Box( );
      Box(float w, float l, float d);

      float getVolume( ) const;
      float getWidth( ) const;
      float getLength( ) const;
      float getDepth( ) const;

      void setWidth( float w );
      void setLength( float l );
      void setDepth( float d );

   private:
      float width;
      float length;
      float depth;
};
```

Implement the non-default constructor for this class (the second constructor). You just need to write the definition for this constructor, not the entire class implementation.

Now write some client code (assuming the rest of the class has been implemented) that creates a Box object, sets its initial dimensions, and prints the volume of the box using cout. There are multiple ways to do this - any working approach is fine.

```
#include <iostream>
#include "Box.h"
using namespace std;

int main(){







}
```

What do the private and public access modifiers do to class methods and data? When do we generally use public and when do we generally use private in creating a C++ class?

What is the difference between the specifications file and the implementation file in class design? Why have two different files for a single class?

What is the relationship between a C++ class and an abstract datatype (ADT)?