Class Six

Object oriented programming

- the world is full of objects
- think of oop as defining and creating objects and relationships between them
- Until now we have been using objects created by c++ for us

Structs
Structs are sets of varriables you can store together in user defined types or aggregate data types.

- **Structs are quick ways to essentially have lists of variables stored in one easy to use object**
t
- **You can instantiate and use their variables using the dot syntax**

Here's an example of how they work:

```
struct Employee
{
    int nID;
    int nAge;
    float fWage;
};
```

```
Employee myEmployee;
myEmployee.nID = 10;
```

You can also use short hand to set up the variables up when you create them

```
struct Person{
    string name;
    int age;
};

int main(){
   //how to do it using short hand
    Person cleo = {“cleo”, 40};
    cout << cleo.age << “ is “ << cleo.name << “’s age” <<endl;
   return 0;
}
```

Stucts can contain structs as data memebers

```
struct Team {
    vector<Person>players;
};
```

Questions

What is a struct?

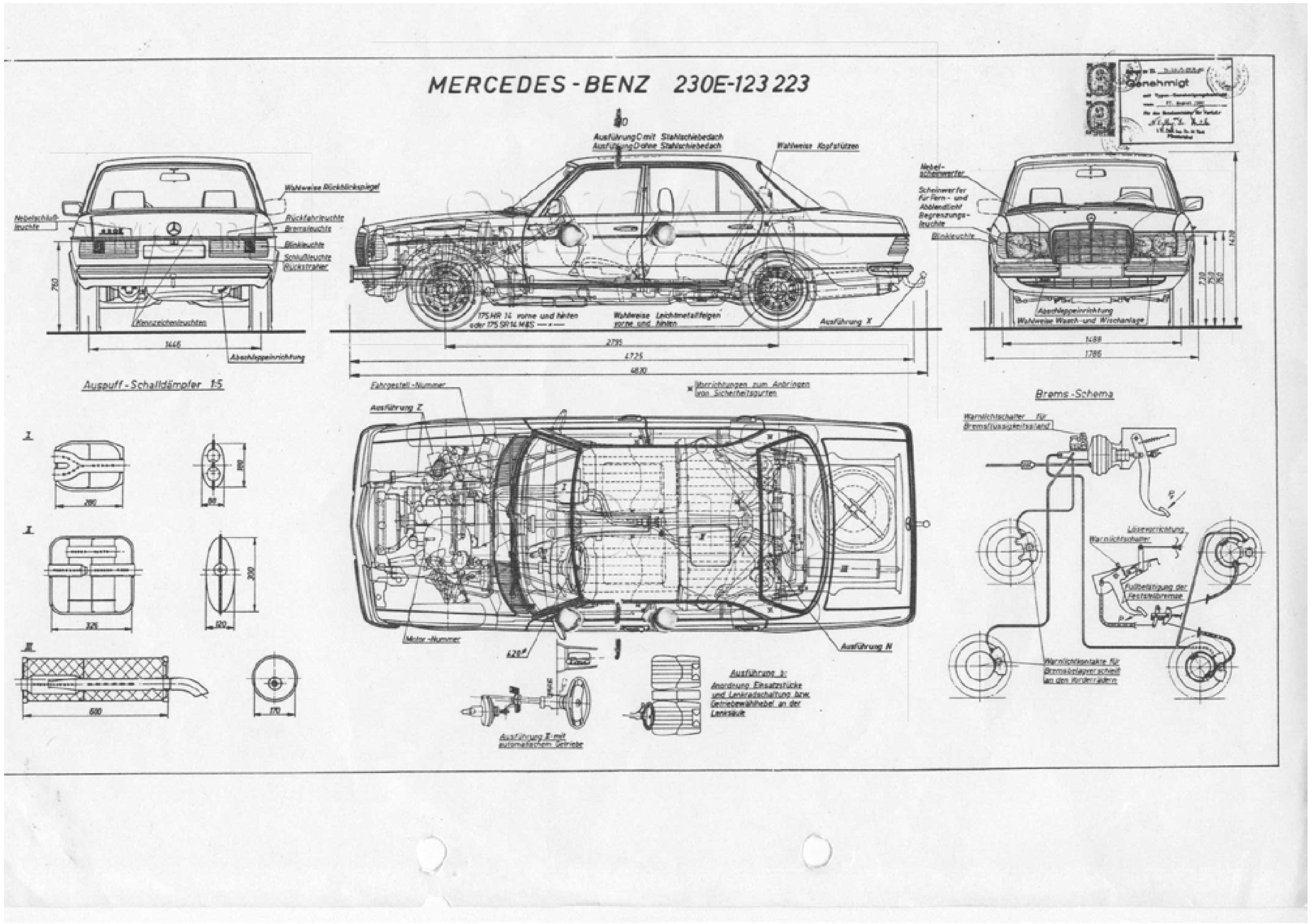
where do you create a struct?

How do you create a struct?

What can structs contain?

Classses

Classes are objects in C++
They are the blueprint from which other objects are created.



Encapsulation

Definition: In Object Oriented Programming, encapsulation is an attribute of object design. It means that all of the object's data is contained and hidden in the object and access to it restricted to members of that class

Example:

You are building a 2 player game with two armies. Each army has a fixed number of soliders. You do not want the general of the army to be able to command the soilders of the other so you encapsulate the functionality of each object by making the command function **private.**

This is thought of as procedural programming
Tasks are broken down into a series of smaller tasks and implemented in manage-able chunks of code, such as functions. In procedural programming, functions and data are separate.

They contain memeber variables and functions

```
// Public members are accessable to in all locations an object exsists
public:
    vector<string>carFeatures;
    string m_carName;
    int m_numberOfDoors;

    //constructors are always public
    Car(string name, int doors);

    void cruiseControl();

//Private members are only available to the member functions and memeber variables of the class

private:
    vector<string> contentsOfGloveBox;

//protected are only available to objects that extend this class. More on this later!
protected:
    int maxSpeed;
```

What are constructors?

Classes can contain constructors. They are chunks of code that set up your object for you. Think of them as functions that run whenever **create your class.**

They are the gateway into your object. They should pass into the class all of the data the class needs from the outside world to be created.
You set them up in two places
1. You must prototype it according to scope scope
2. By using the scope resolution operator or :: you define the function. You can also simply place it inside of the class itself.

(I cut this example down so I could get it on one slide)

```
class Car
{

public:
    Car(string name, int doors);
};
```

Member variables

It’s common to assign member variables to the temp variables passed into the con-stuctor.

A common practice to distingush the two is use m_

```
Car::Car(string name=”noName”, int doors=0){
    m_carName = name;
    m_numberOfDoors = doors;

}
```

This is done so you can set up a bridge between your object and the rest of the world. It’s good practice to not be constantly assigning variables in classes at the object level. This is because you should have functions that do this for you - leverage **encapsulation**

t

1. You prototype them to the appropriate public, private or protected scope

```
public:
   void driveFast();
```

2. You define them in a function scoped to that class.

```
Car::cruiseControl(){
      cout << “cruise control on”;
}
```

Default constructors

Constructors come in a few forms.

• Default constuctors. These constructors have no parameters. Also they will NOT instantiate any of your member variables so be careful. That's up to you!

```
Car::Car()  {

}
```

• Constuctors with parameters.

Car::Car(string name, int model);

• Constuctors with default parameters.

```
Car::Car(string name= ”sedan”, int doors=4);
```

You can spot these guys by the assignement operation happening here.

Multiple constructors

You can have an overloaded constuctor. AKA you can have as many constructors as you want.

Be careful though b/c if they take the same type but one has a default value, c++ will not see them as uniquely overloaded.

```
Car::Car(string name=”sedan”);

Car::Car();

Car::Car(string name =”sedan”, int doors =4, bool sunroof);
```

Setting up the Constructor with the class.

```
class Enemy{
    public:
    int damage =10;
    string name = “goomba”;
    Enemy(){}
    Enemy(int damage_){
        damage = damage_;
    }
    void foo(){
        cout << “foo!” <<endl;
    }
//    Enemy(int damage=1){}
//not a valid method signiture

};

//note that you can do this in this style as well
```

Making our first object!

To create an object out of the class we do it just like making any object.
An object is a instance of the class not the class itself.

```
int main(){
Car myCar(“mustang”,2);
return 0;
}
```

Destructors

A destructor is another special kind of class member function that is executed when an object of that class is destroyed. They are the counterpart to constructors. When a variable goes out of scope, or a dynamically allocated variable is explicitly deleted using the delete keyword, the class destructor is called

Like constructors, destructors have specific naming rules:
1) The destructor must have the same name as the class, preceded by a tilde (~).
2) The destructor can not take arguments.
3) The destructor has no return type.

Destructor Example

```
class Enemy{


    public:
    int damage =10;
    string name = “goomba”;
    string allies = new allies[“king kong”, “the corporation”, “my landlords”];

   //default constructor
    Enemy(){}

  //destructor
  ~Enemy() {
    delete[]allies;
  }

};
```

Static members

tstatic member variables and functions belong to the class not the object.
You can use them to do things like keep track of how many instances of an object have been created so far.

```
public:
  static int s_Total;     //static member variable declaration
 //total number of Critter objects in existence

Critter(int hunger = 0);
static int GetTotal();   //static member function prototype private:
int m_Hunger;
};

int Critter::s_Total = 0;
 //static member variable initialization Critter::Critter(int hunger):

Critter::Critter(int hunger): m_Hunger(hunger) {
cout << “A critter has been born!” << endl;
++s_Total;
 }
```

Also note the constructor shorthand

```
Critter::Critter(int hunger):  m_Hunger(hunger) {      cout << “A crit-
ter has been born!” << endl;

   ++s_Total;

 }
```

This is shorthand for assigning the values passed to the constructor to the member varaibles.

Questions:

What is a class?

What is an object?

What is a constuctor?

What is a member variable?

What is overloading?

What is a destructor?

The stack and the heap

The memory a program uses is typically divided into four different areas:

- The code area, where the compiled program sits in memory.
- The globals area, where global variables are stored.
- The heap, where dynamically allocated variables are allocated from.
- The stack, where parameters and local variables are allocated from.

There isn't really much to say about the first two areas. The heap and the stack are where most of the interesting stuff takes place, and those are the two that will be the focus of this section.

The heap

The heap (also known as the "free store") is a large pool of memory used for dynamic allocation. In C++, when you use the new operator to allocate memory, this memory is assigned from the heap.

```
int *pValue = new int; // pValue is assigned 4 bytes from the heap

int *pArray = new int[10]; // pArray is assigned 40 bytes from the heap
```

Because the precise location of the memory allocated is not known in advance, the memory allocated has to be accessed indirectly — which is why new returns a pointer.

When a dynamically allocated variable is deleted, the memory is "returned" to the heap and can then be reassigned as future allocation requests are received.

The heap has advantages and disadvantages:
1) Allocated memory stays allocated until it is specifically deallocated (beware memory leaks).
2) Dynamically allocated memory must be accessed through a pointer.
3) Because the heap is a big pool of memory, large arrays, structures, or classes should be allocated here.

Stack

The call stack (usually referred to as “the stack”) has a much more interesting role to play. Before we talk about the call stack, which refers to a particular portion of memory, let’s talk about what a stack is.

Consider a stack of plates in a cafeteria. Because each plate is heavy and they are stacked, you can really only do one of three things:
1) Look at the surface of the top plate
2) Take the top plate off the stack
3) Put a new plate on top of the stack

Another metaphor for the call stack

Consider a bunch of mailboxes, all stacked on top of each other. Each mailbox can only hold one item, and all mailboxes start out empty. Furthermore, each mailbox is nailed to the mailbox below it, so the number of mailboxes can not be changed. If we can't change the number of mailboxes, how do we get a stack-like behavior?

The call stack is a fixed-size chunk of sequential memory addresses. The mailboxes are memory addresses, and the "items" are pieces of data (typically either variables or addreses). The "marker" is a register (a small piece of memory) in the CPU known as the stack pointer. The stack pointer keeps track of where the top of the stack currently is.

The only difference between our hypothetical mailbox stack and the call stack is that when we pop an item off the call stack, we don't have to erase the memory (the equivalent of emptying the mailbox). We can just leave it to be overwritten by the next item pushed to that piece of memory. Because the stack pointer will be below that memory location, we know that memory location is not on the stack.

What do we use the stack for?

Parameters, local variables, and... function calls.

Because parameters and local variables essentially belong to a function, we really only need to consider what happens on the stack when we call a function. Here is the sequence of steps that takes place when a function is called:

The address of the instruction beyond the function call is pushed onto the stack. This is how the CPU remembers where to go after the function returns.
Room is made on the stack for the function's return type. This is just a placeholder for now.
The CPU jumps to the function's code.
The current top of the stack is held in a special pointer called the stack frame. Everything added to the stack after this point is considered "local" to the function.
All function arguments are placed on the stack.
The instructions inside of the function begin executing.
Local variables are pushed onto the stack as they are defined.
When the function terminates, the following steps happen:

When the function terminates, the following steps happen:

The function’s return value is copied into the placeholder that was put on the stack for this purpose.

Everything after the stack frame pointer is popped off. This destroys all local variables and arguments.

The return value is popped off the stack and is assigned as the value of the function. If the value of the function isn’t assigned to anything, no assignment takes place, and the value is lost.

The address of the next instruction to execute is popped off the stack, and the CPU resumes execution at that instruction.

Stack overflow

The stack has a limited size, and consequently can only hold a limited amount of information. If the program tries to put too much information on the stack, stack overflow will result. Stack overflow happens when all the memory in the stack has been allocated — in that case, further allocations begin overflowing into other sections of memory.

```
int main()
{
    int nStack[100000000];
    return 0;
}
```

The stack has advantages and disadvantages:

Memory allocated on the stack stays in scope as long as it is on the stack. It is destroyed when it is popped off the stack.
All memory allocated on the stack is known at compile time. Consequently, this memory can be accessed directly through a variable.
Because the stack is relatively small, it is generally not a good idea to do anything that eats up lots of stack space. This includes allocating large arrays, structures, and classes, as well as heavy recursion.

**These last few slides are shared from:
http://www.learncpp.com/cpp-tutorial/79-the-stack-and-the-heap/**

Homework

Read and complete chapter 8 including the homework:

1. Improve the Critter Caretaker program so that you can enter an unlisted menu choice that reveals the exact values of the critter's hunger and boredom levels.

2. Change the Critter Caretaker program so that the critter is more expressive about its needs by hinting at how hungry and bored it is.