Chapter 10

Objects and Classes

OOP

Object Oriented Programming (OOP) is a programming paradigm based upon a concept called “objects” which contain data and methods.

OOP programs are designed by making objects interact with one another and is widely used in industry today

Examples of OOP languages:

C++, Java, Python, C#, Ruby, Swift, Scala, Kotlin, Javascript, etc….

Let’s do a brief review of Object Oriented Design!

OOP: A Vending Machine Example

Where is the logic for:

- Accepting money and dispensing change?
- Dispensing the product selected by the customer?
- Knowing how much money is required to buy a product?
- What if not all products have the same price?
- Knowing whether or not a product is available?

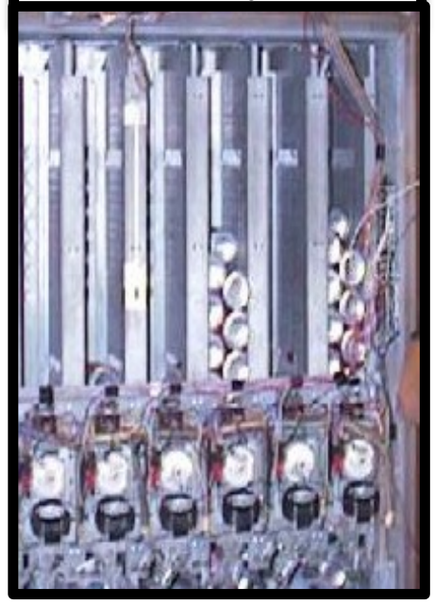
Building a System from Modular Parts

Coke Machine

Payment Handler

Product Selector

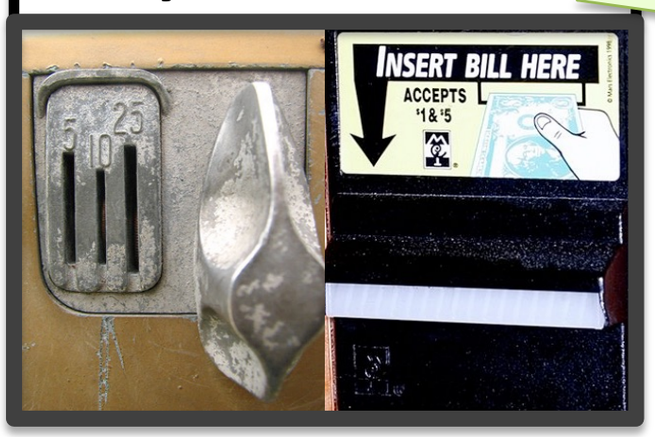
Product Dispenser



Warning Light

Modular parts have hidden details/complexity

Payment Handler



Hidden from other components:

- ***How it reads value of bills***
- ***How it determines value of coins***
- ***How it accumulates total inserted***
- ***What it does with money it collects***
- ***How it rejects counterfeits***

Product Dispenser

Scale (or sensor) that detects if empty

Chutes that open long enough to release one item

Refrigeration device that keeps products cold

Modular parts are..

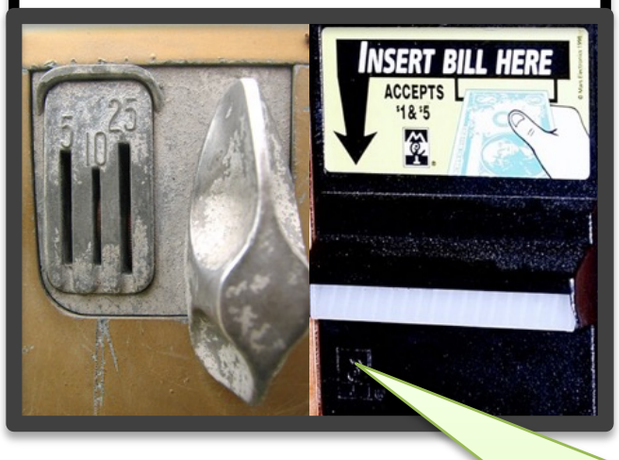
Connected and interact with each other

Coke Machine

Unified interface for customer to interact with

Modular to be reusable in other products

Payment Handler



Warning Light

These components can be used as-is in any device that require money collection or a warning indicator

Interchangeable with different parts that service the same functional needs

You can upgrade the payment handler without requiring changes to any of the other components

Connecting software to hardware...

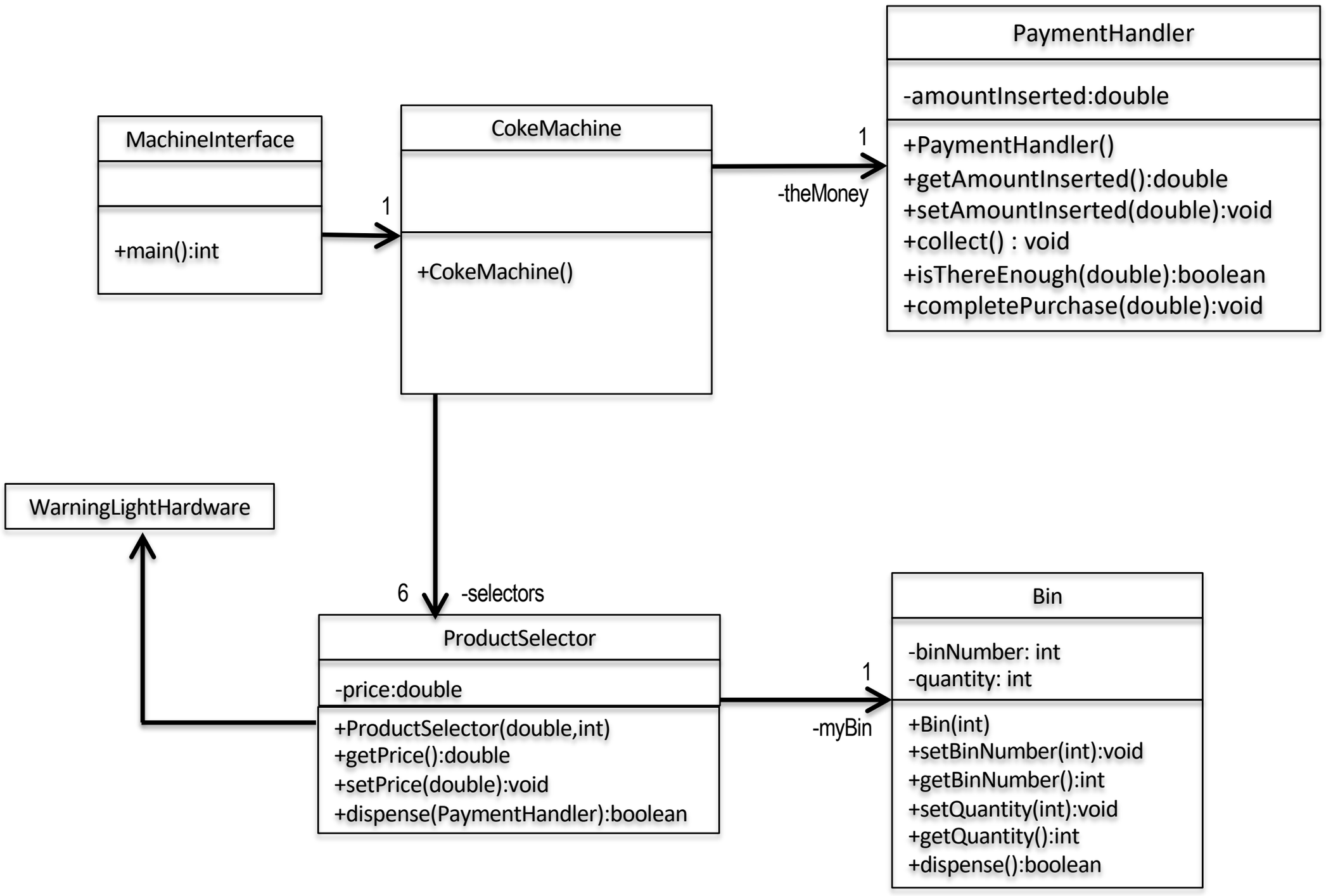
Your program written in C++ (or Java, or C#, or ...)

ProductSelector
-price:double
+ProductSelector(double,int) +getPrice():double +setPrice(double):void +dispense(PaymentHandler):boolean

API provided by the hardware vendor

(Usually) compiled program provided by the manufacturer that controls the hardware, with public interface to be called from your program.

Connect all the parts…



Moving toward objects…

```
int main{
     code…
     code…
     code…
     code…
     code…
     code…
     code…
     code…
     code…
     code…
     code…
     code…
     code…
     }
```

```
int main{
     code…
     code…
     Other();
     code…
     code…
     code…
     code…
     code…
     AndOther()
     ;
     code…
     code…
     code…
     code…
     code…
     code…
     }
```

```
void Other()
   {
   code…
   code…
   code…
   code…
   code…
   }
```

```
int AndAnother() {
   code…
   code…
   code…
   code…
   return...
   }

}
```

Moving toward objects…

```
int main {

MyClass anObject;

MyOtherClass anotherObject;


AndOtherClass oneMoreObject;
oneMoreObject.function1();

//etc...



}
```

```
class MyClass {
public:
  void function1();
  void function2();
private:
  string attribute1;
  string attribute1;
function code...
```

```
class MyOtherClass {
public:
  void function1();
  void function2();
  void function3();
private:
  int attribute1;
  string attribute2;
function code...
```

```
class AndOtherClass {
public:
  void function1();
  void function2();
private:
  int attribute1;
  int attribute2;
  string attribute3;
function code...
```

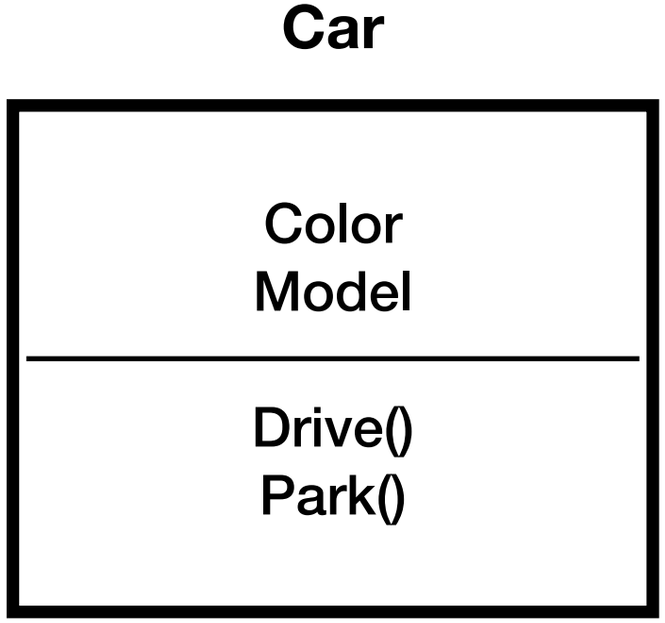
Separating the "parts" of your system allow each to be more easily maintained AND reusable in other systems.

Objects

Object - a grouping of data and operations that can be performed on that data

- Data / Variable Attributes

- Functions / Methods



Abstraction

Objects are another common form of abstraction

Objects strongly support abstraction, hiding entire groups of functions and variables, exposing only certain functions to a user.

An ***abstract data type*** (***ADT***) is a data type whose creation and update are constrained to specific well-defined operations. A class can be used to implement an ADT.

ADT's are used to define what operations are available rather than how (Think of the a vector!)

Classes vs Objects

Class - a class is a definition of data and methods that are encapsulated in one unit

Object - an object is an *instance* of a class.

When a class is defined, no memory is allocated

When an object is instantiated, memory becomes allocated for that object.

C++ OO Implementation Details

Function declarations

```
class ClassName {

public:
//externally accessible functions go here
void functionName();

protected:
//members accessible by subclasses (more on this later)

private:
//For internal use only - all (most) data goes here

};

//constructor and function details go down here

void ClassName::functionName() {

}
```

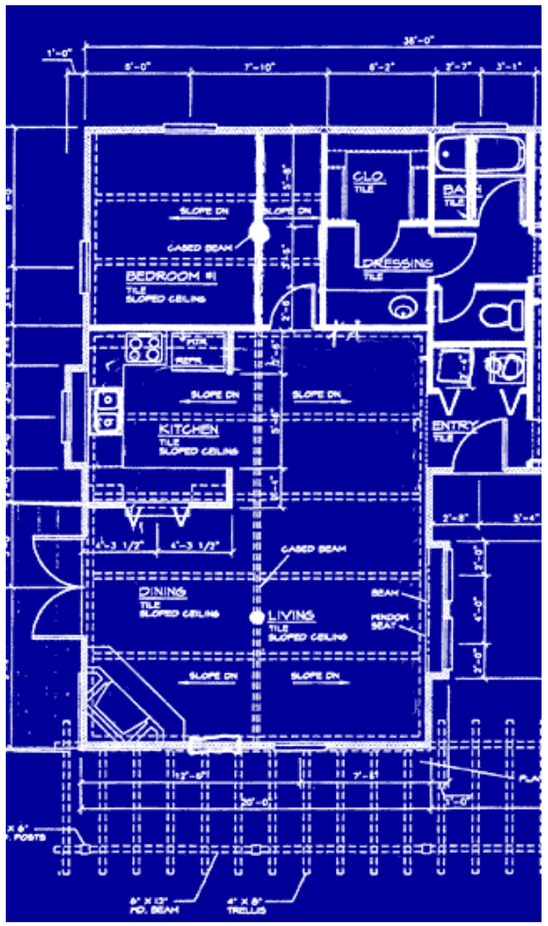
Function definition

Namespace syntax

:: is the scope resolution operator

Object-Oriented Terminology

Class



Object/Instance

```
class House {
public:
 House();
//etc...
```

No memory used

Occupies memory

```
House myHouse();
myHouse.setSquareFootage(1750);
```

Data Members

Classes can define 3 different types of data members:

1. **Public Data Members** - data members within the class that are accessible to the public anywhere outside of the class

2. **Private Data Members** (*default*)- data members that can only be accessed within the class (cannot be accessed or viewed by public). *PREFERRED METHOD!!*

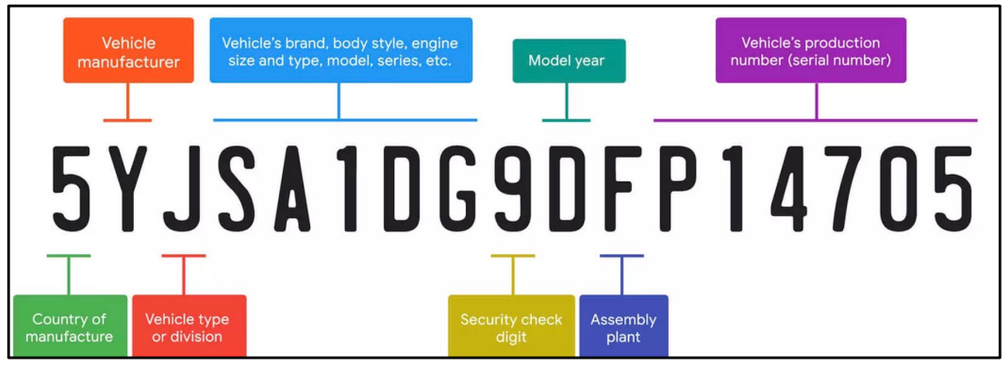
3. **Protected Data Members** - private data that can be accessed by child classes (More on this during inheritance)

example: class.cpp

Why private?

```
class Car
{
public:
    Car();
    void setVin(string s);
    string getVin();
    //etc...

private:
    string make;
    string model;
    string color;
    int year;
    string vin;
};

Car::setVin(string s)
{
//set the value of vin
//verify format and contents


}
```

```
class SomeOtherClass
{
public:
    //SomeOtherClass public members
private:
    void doSomething(Car c);
    //SomeOtherClass private members
};


SomeOtherClass::doSomething(Car c)
{
    c.vin = "ABC123XYX1341";   //WON'T compile

    c.setVin("ABC123XYX1341"); //WILL compile
}
```



Member Functions

Function Declarations - provides function name, return type, and arguments

Function Definition - provides class name, return type, arguments, and the function's statements

- We can define a function definition within the class using **inline member functions.**

- Best practice is to define it outside of class definition using the **scope resolution operator (::)**

examples: class.cpp
restaurant.cpp
dog_license.cpp

Member Functions

Class using scope resolution

```
class MyClass {
   public:
      void Fct1();
   private:
      int numA;
};

void MyClass::Fct1() {
   numA = 0;
}
```

Class with inline member function

```
class MyClass {
   public:
      void Fct1() {
         numA = 0;
      }
   private:
      int numA;
};
```

Member Functions

Class using scope resolution

```
class MyClass {
   public:
      void Fct1();
   private:
      int numA;
};

void MyClass::Fct1() {
   numA = 0;
}
```

Class with inline member function

```
class MyClass {
   public:
      void Fct1() {
         numA = 0;
      }
   private:
      int numA;
};
```

C++ OO Implementation Details

Accessors and Mutators

Accessors (Getters)

- Functions used to access a data member but not modify it.
- *Usually* defined as const to prevent changing the member.
- Caution: A function that calls a const function must also be const
- Simplest form: `string getName() const { return name; }`

Mutators (Setters)

- Functions used to modify ("mutate") a data member.
- Simplest form: `void setName(string s) {name = s; }`

Accessors and Mutators provide a way for instances of other classes to view and change private data members.

These functions provide a place to include logic to safeguard data (e.g., data validation, security rules, etc.)

Initialization

It is considered good practice to initialize all variables when they are declared.

```
int val;
int val = 0;
```

When initializing objects we want to initialize all data members as well.

One way is to initialize variables in the class definition

Note: in-class data member initialization is available in version **c++11**

examples: init_data_members.cpp

Constructors

A **constructor** is a member function of a class which initializes objects of a class.

- Constructor has same name as the class itself
- Constructors don’t have a return type
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

Default Constructor

A **default constructor** is a constructor without any arguments. This is the constructor that is called automatically when we create an object like:

```
Employee accountant;
```

We create a default constructor like a member function:

```
Employee::Employee()  {
   /*  ...  */
}
```

example: constructors.cpp

Constructor Overloading

We can define multiple constructors for a class in order to initialize member variables to values other than default.

```
Employee accountant("name", id);
```

The constructor matching these parameters will be called

```
Employee::Employee(string name, int id) {
   /* ... */
}
```

example: constructors.cpp

Constructor Overloading

We can define multiple constructors for a class in order to initialize member variables to values other than default.

```
Employee accountant("name", id);
```

The constructor matching these parameters will be called

```
Employee::Employee(string name, int id) {
   /* ... */
}
```

example: constructors.cpp

Constructor Overloading

```
class Employee {
    Employee();                          // A
    Employee(string name);               // B
    Employee(string name, int num); // C
}
```

Given the above, which constructor is called given the following object declarations.

1. Employee manager(“George”);

2. Employee seniorAssociate(“Elroy”, 123);

3. Employee temp;

Constructor Overloading

```
class Employee {
    Employee();                          // A
    Employee(string name);               // B
    Employee(string name, int num); // C
}
```

Given the above, which constructor is called given the following object declarations.

1. Employee manager(“George”); **B**

2. Employee seniorAssociate(“Elroy”, 123); **C**

3. Employee temp; **A**

Constructors with Default Params

Constructors, just like functions, can have default parameters.

```
int setName(string name = "name");
```

If a Constructor defined with default params can be called without a parameter, the constructor will act as the **default constructor.**

```
class Employee {
  public:
    Employee(string name = "No Name");
}
```

example: constructor_default_params.cpp

Constructor initializer lists

Programmers can utilize a **constructor initializer list** as an alternative for initializing data members in a constructor.

```
SampleClass::SampleClass() {
    field1 = 100;
    field2 = 200;
}

SampleClass::SampleClass() : field1(100),
field2(200) {
}
```

example: initializer_list.cpp

this

When an object member function is called *object.Function( ).* The compiler converts this to a function with the object passed by pointer (similar to pass by reference).

```
Function(object, ...);
```

We can access this object pointer with ***this.***

```
this -> member;
```

We mostly use the object pointer to explicitly refer to the objects data member. Used commonly with parameters that are the same name as a data member.

example: this.cpp

Operator Overloading

Operation Overloading allows a programmer to redefine functionality of built in operators to work with the class.

=, +, -, *, /, ^, >, <, >=, <=, ==

Using an operator is actually calling a function.

```
1.operator+(1);
```

example: operator_overload.cpp

File Structure with Classes

In C++, we often times break out classes into implementation files and header files to make our programs more organized and legible.

Table 10.8.1: Typical two files per class.

ClassName.h	Contains the class definition, including data members and member function declarations.
ClassName.cpp	Contains member function definitions.

The header file must be included in the implementation file so that the compiler knows the namespace of ClassName.

example: class_files/

Static Data and Functions

The keyword **static** is used to indicate that a variable is allocated in memory only once during execution.

Static variables reside in Static / Global Memory and have global scope.

In a class, a **static data member** is a data member of the class instead of a data member of each class object. Thus, static data members are independent of any class object, and can be accessed without creating a class object.

We can access a static data member by using the scope resolution operator for that class. (i.e. ClassName::variableName)

example: static.cpp

Pointers with Objects/Classes

It is very common that we use pointers and the *new* keyword to create objects.

```
Student *a = new Student;
```

When the program executes this line, we first allocate the space for *Student* on the **heap,** then we call the *Student* default constructor.

We can also use other overloaded constructers for a class during initialization:

```
Student *a = new Student;
Student *a = new Student("Erik");
```

example: class.cpp

Member Access Operator

Accessing data members of a class can be done like the following:

```
Student *a = new Student;
(*a).data_member;
(*a).function();
```

However, with C++ and classes, we can use the **member access operator (->)** as an alternative (and easier) way of accessing data members or functions of a class.

```
a->data_member;
a->function();
```

example: class.cpp

Rule of Three

- **Destructor**

- **Copy Constructor**

- **Copy Assignment Operator**

Destructors

A **destructor** is a member function that is called which destructs or deletes an object.

```
Employee::~Employee()
{
    /* Clean up data */
}
```

Destructors are called when:

1. A function ends
2. The program ends
3. A block containing local variables end
4. A delete operator is called

A class only has one destructor

example: destructor.cpp

Copy Constructor

A **copy constructor** is used for a class to make a copy of an existing instance.

We use constants when we want to customize or tell the program how to copy an object when we *pass by value.*

*Remember: *Pass by value* creates a copy of the data and performs any operations on that copy in the function.

Copy Constructors are implicitly defined by the compiler if you do not define one and performs a member-wise copy of the source object.

Copy Constructor

It is important that we declare a copy constructor by passing in the class we want to copy as *const.*

```
MyClass::MyClass(const MyClass &copyClass) {
    . . .
}
```

The most common use for the copy constructor is when the class has raw pointers as member variables and we need to make a **deep** copy of the data.

example: copy_constructor.cpp

Deep vs Shallow copy

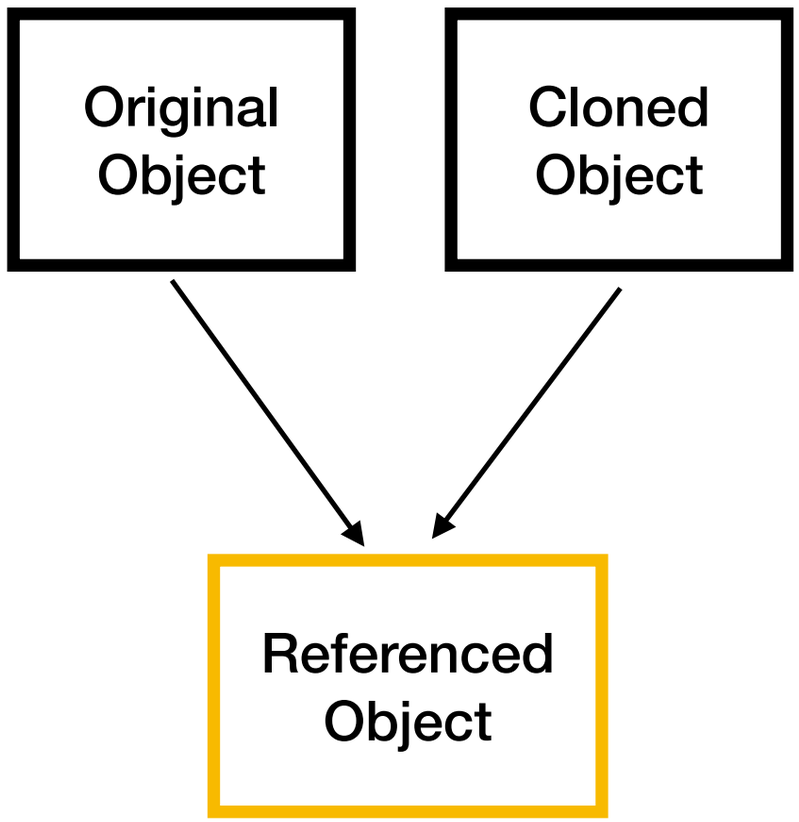
Deep Copy - a complete full copy of all data members of a class, including dynamically allocated memory.

Shallow Copy - a member-wise copy where we only copy the data members and NOT any dynamically allocated memory.

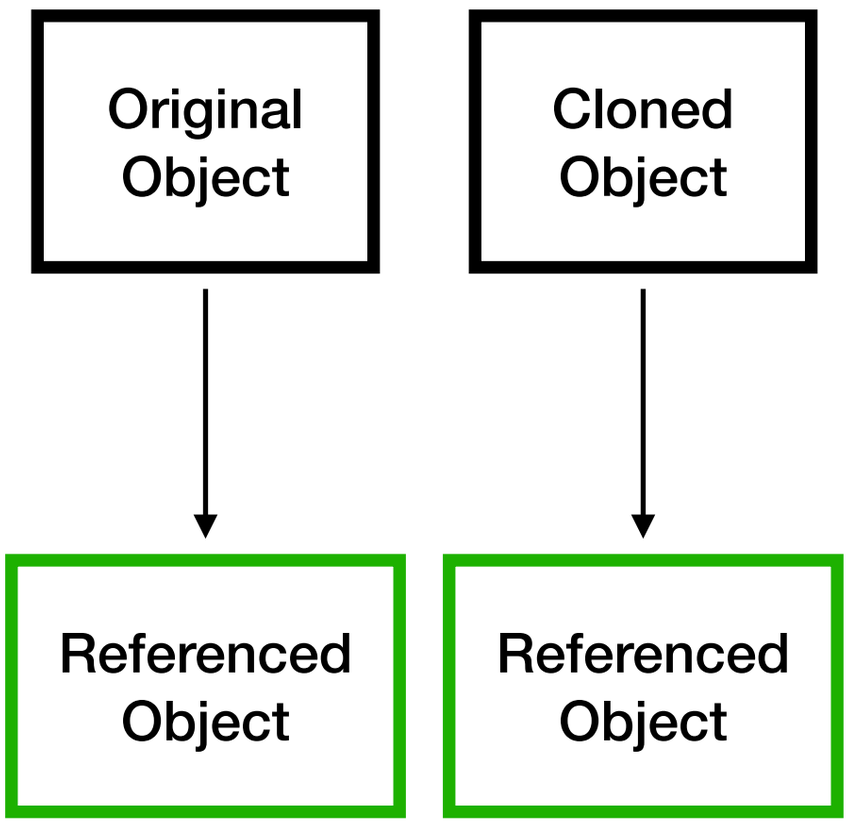
Your computer performs a shallow copy by default if no copy constructor is provided.

example: deep_copy.cpp

Deep vs Shallow copy



Shallow
An object is copied but the underlying data (pointers) still point to the same thing



Deep Copy
A copy of each referenced object is made so that there is no shared entity between the objects

Copy Assignment Operator

Often times we want to copy one object to another using the assignment operator.

The program will implicitly create a copy assignment operator for you and do a member-wise copy.

But again we want to do a **deep copy** with an objects dynamically allocated data.

```
MyClass& Myclass::operator=(const MyClass& copy) {

}
```

example: copy_assignment.cpp

Rule of Three

If you have to define one of the following as a programmer, then it is a good practice to define all of them.

- **Destructor**

- **Copy Constructor**

- **Copy Assignment Operator**

These are often known as the “big three”

Command Line Arguments

Command-line arguments are values entered by the user when running the programs executable from the command line or terminal.

We read in command line arguments by providing additional parameters to our *main* function.

```
int main(int argc, char* argv[]) {

}
```

In C++ the only way to read in command line arguments is by using c-strings!

example: command_line_args.cpp