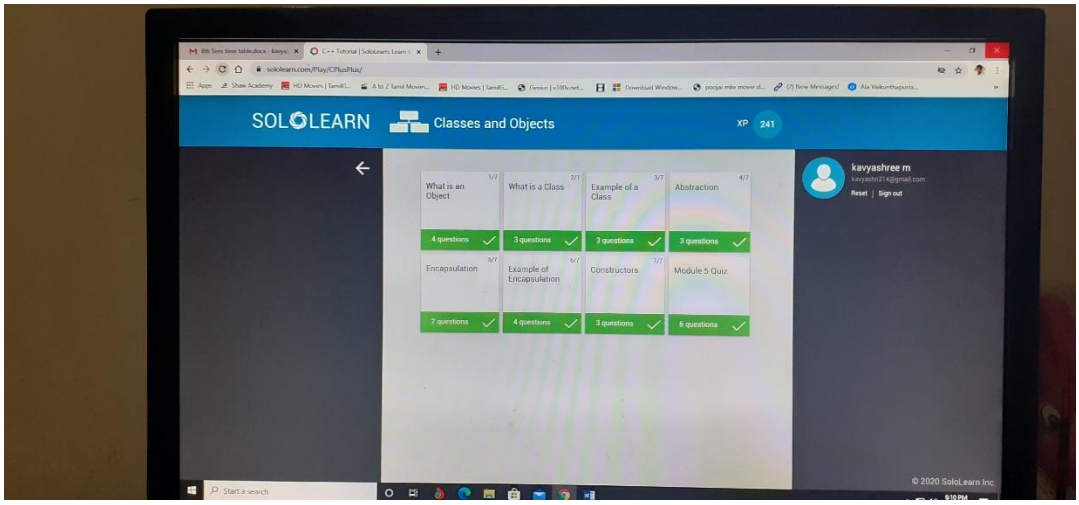
DAILY ASSESSMENT

Date:	**24-6-2020**	**Name:**	**Kavyashree m**
Course:	**C++**	**USN:**	**4al15ec036**
Topic:	**What is an object, what is class, abstraction, encapsulation, constructors**	**Semester & Section:**	**8th A**
GitHub Repository:	**kavya**		

FORENOON SESSION DETAILS



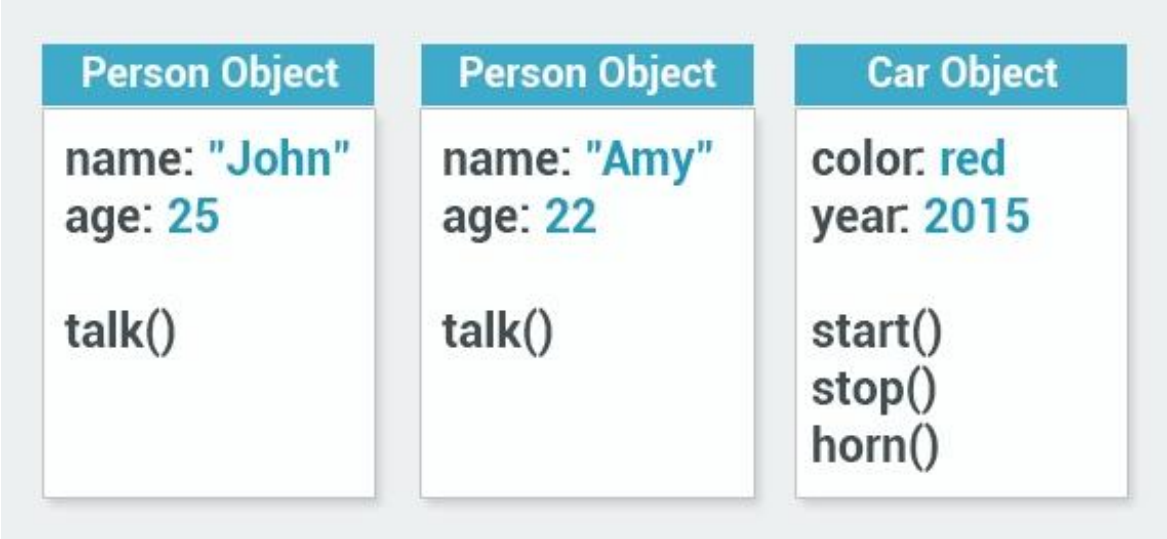
What is an Object

Object Oriented Programming is a programming style that is intended to make thinking about programming closer to thinking about the real world. In programming, objects are independent units, and each has its own identity, just as objects in the real world do. An apple is an object; so is a mug. Each has its unique identity. It's possible to have two mugs that look identical, but they are still separate, unique objects.

An object might contain other objects but they're still different objects. Objects also have characteristics that are used to describe them. For example, a car can be red or blue,

a mug can be full or empty, and so on. These characteristics are also called attributes. An attribute describes the current state of an object. Objects can have multiple attributes (the mug can be empty, red and large). An object's state is independent of its type; a cup might be full of water, another might be empty.

In the real world, each object behaves in its own way. The car moves, the phone rings, and so on.The same applies to objects - behavior is specific to the object's type. So, the following three dimensions describe any object in object oriented programming: identity, attributes, behavior. Each object has its own attributes, which describe its current state. Each exhibits its own behavior, which demonstrates what they can do.



In computing, objects aren't always representative of physical items. For example, a programming object can represent a date, a time, a bank account. A bank account is not tangible; you can't see it or touch it, but it's still a well-defined object - it has its own identity, attributes, and behavior.

What is a Class

Objects are created using classes, which are actually the focal point of OOP. The class describes what the object will be, but is separate from the object itself. In other words, a class can be described as an object's blueprint, description, or definition. You can use the same class as a blueprint for creating multiple different objects. For example, in preparation to creating a new building, the architect creates a blueprint, which is used as a basis for actually building the structure. That same blueprint can be used to create multiple buildings.

Programming works in the same fashion. We first define a class, which becomes the blueprint for creating objects. Each class has a name, and describes attributes and behavior. In programming, the term type is used to refer to a class name: We're creating an object of a particular type. Attributes are also referred to as properties or data.

Methods

Method is another term for a class' behavior. A method is basically a function that belongs to a class. Methods are similar to functions - they are blocks of code that are called, and they can also perform actions and return values.

A Class Example

For example, if we are creating a banking program, we can give our class the following characteristics:

name: BankAccount

attributes: accountNumber, balance, dateOpened

behavior: open(), close(), deposit()

The class specifies that each object should have the defined attributes and behavior. However, it doesn't specify what the actual data is; it only provides a definition. Once we've written the class, we can move on to create objects that are based on that class. Each object is called an instance of a class. The process of creating objects is called instantiation.

Declaring a Class

Begin your class definition with the keyword class. Follow the keyword with the class name and the class body, enclosed in a set of curly braces.
The following code declares a class called BankAccount:class BankAccount { };

A class definition must be followed by a semicolon. Define all attributes and behavior (or members) in the body of the class, within curly braces. You can also define an access specifier for members of the class.A member that has been defined using the public keyword can be accessed from outside the class, as long as it's anywhere within the scope of the class object.

Let's create a class with one public method, and have it print out "Hi".class BankAccount
{
public:
void sayHi() {
cout << "Hi" << endl;
}
};

The next step is to instantiate an object of our BankAccount class, in the same way we define variables of a type, the difference being that our object's type will be BankAccount.

```
int main()
{
BankAccount test;
test.sayHi();
}
```

Our object named test has all the members of the class defined. Notice the dot separator (.) that is used to access and call the method of the object.

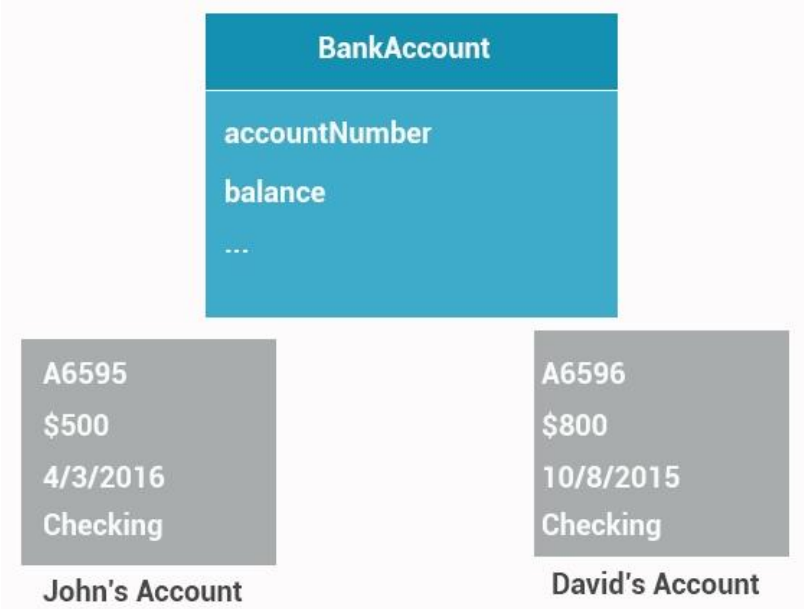
Abstraction

Data abstraction is the concept of providing only essential information to the outside world. It's a process of representing essential features without including implementation details. A good real-world example is a book: When you hear the term book, you don't know the exact specifics, i.e.: the page count, the color, the size, but you understand the idea of a book - the abstraction of the book.

The concept of abstraction is that we focus on essential qualities, rather than the specific characteristics of one particular example. Abstraction means, that we can have an idea or a concept that is completely separate from any specific instance. It is one of the fundamental building blocks of object oriented programming. For example, when you use cout, you're actually using the cout object of the class ostream. This streams data to result in standard output.cout << "Hello!" << endl;

Abstraction allows us to write a single bank account class, and then create different objects based on the class, for individual bank accounts, rather than creating a separate class for each bank account.

Abstraction allows us to write a single bank account class, and then create different objects based on the class, for individual bank accounts, rather than creating a separate class for each bank account.



Encapsulation

Part of the meaning of the word encapsulation is the idea of "surrounding" an entity, not just to keep what's inside together, but also to protect it. In object orientation, encapsulation means more than simply combining attributes and behavior together within a class; it also means restricting access to the inner workings of that class.

The key principle here is that an object only reveals what the other application components require to effectively run the application. All else is kept out of view. For example, if we take our BankAccount class, we do not want some other part of our program to reach in and change the balance of any object, without going through the deposit() or withdraw() behaviors.

We should hide that attribute, control access to it, so it is accessible only by the object itself. This way, the balance cannot be directly changed from outside of the object and is accessible only using its methods. This is also known as "black boxing", which refers to closing the inner working zones of the object, except of the pieces that we want to make public.

This allows us to change attributes and implementation of methods without altering the overall program. For example, we can come back later and change the data type of the balance attribute.

Access Specifiers

Access specifiers are used to set access levels to particular members of the class. The three levels of access specifiers are public, protected, and private.
A public member is accessible from outside the class, and anywhere within the scope of the class object.

For example:

```
#include <iostream>
#include <string>
using namespace std;
class myClass {
public:
string name;
};

int main() {
myClass myObj;
myObj.name = "SoloLearn";
```

```
cout << myObj.name;
return 0;
}
//Outputs "SoloLearn"
```

Private

A private member cannot be accessed, or even viewed, from outside the class; it can be accessed only from within the class.

A public member function may be used to access the private members. For example:

```
#include <iostream>
#include <string>
using namespace std;
class myClass {
public:
void setName(string x) {
name = x;
}
private:
string name;
};

int main() {
myClass myObj;
myObj.setName("John");
return 0;
}
```

Access specifier

We can add another public method in order to get the value of the attribute.class myClass

{

public:

void setName(string x) {

name = x;

}

string getName() {

return name;

}

private:

string name;

};

Putting it all together:

#include <iostream>

#include <string>

using namespace std;

class myClass {

public:

void setName(string x) {

name = x;

}

string getName() {

return name;

```
}
private:
string name;
};

int main() {
myClass myObj;
myObj.setName("John");
cout << myObj.getName();
return 0;
}
//Outputs "John"
```

Private

A private member cannot be accessed, or even viewed, from outside the class; it can be accessed only from within the class.

A public member function may be used to access the private members. For example:

```
#include <iostream>
#include <string>
using namespace std;
class myClass {
public:
void setName(string x) {
name = x;
```

```
}
private:
string name;
};

int main() {
myClass myObj;
myObj.setName("John");
return 0;
}
```

We can add another public method in order to get the value of the attribute.class myClass

```
{
public:
void setName(string x) {
name = x;
}
string getName() {
return name;
}
private:
string name;
};
```

Putting it all together:

```
#include <iostream>
#include <string>
using namespace std;
class myClass {
public:
void setName(string x) {
name = x;
}
string getName() {
return name;
}
private:
string name;
};

int main() {
myClass myObj;
myObj.setName("John");
cout << myObj.getName();
return 0;
}
//Outputs "John"
```

Constructors

Class constructors are special member functions of a class. They are executed whenever new objects are created within that class. The constructor's name is identical to that of the class. It has no return type, not even void.

For example:

```
class myClass {
public:
myClass() {
cout <<"Hey";
}
void setName(string x) {
name = x;
}
string getName() {
return name;
}
private:
string name;
};

int main() {
myClass myObj;
return 0;
}
//Outputs "Hey"
```

Constructors can be very useful for setting initial values for certain member variables. A default constructor has no parameters. However, when needed, parameters can be added to a constructor.

This makes it possible to assign an initial value to an object when it's created, as shown in the following example:

```
class myClass {
public:
myClass(string nm) {
setName(nm);
}
void setName(string x) {
name = x;
}
string getName() {
return name;
}
private:
string name;
};
```

When creating an object, you now need to pass the constructor's parameter, as you would when calling a function:

```
class myClass {
public:
myClass(string nm) {
```

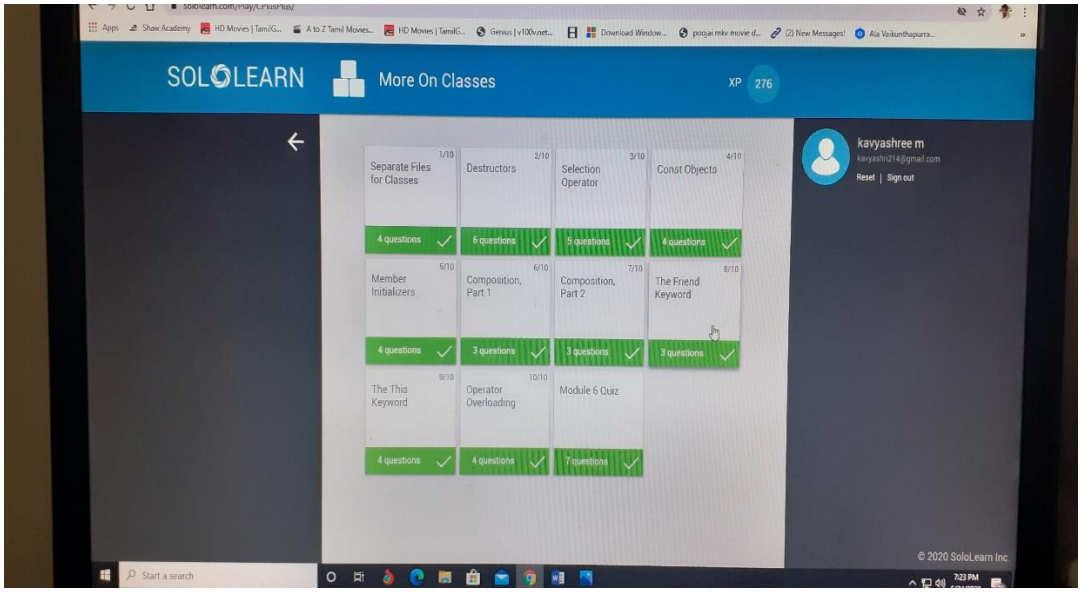
```
setName(nm);
}
void setName(string x) {
name = x;
}
string getName() {
return name;
}
private:
string name;
};

int main() {
myClass ob1("David");
myClass ob2("Amy");
cout << ob1.getName();
}
//Outputs "David"
```

AFTERNOON SESSION DETAILS

Date:	**24-6-2020**	**Name:**	**Kavyashree m**
Course:	**C++**	**USN:**	**4al15ec036**
Topic:	**Classes, destructors, selection operator , member initialize, operator overloading**	**Semester & Section:**	**8th A**
GitHub Repository:	**kavya**		

Image of session

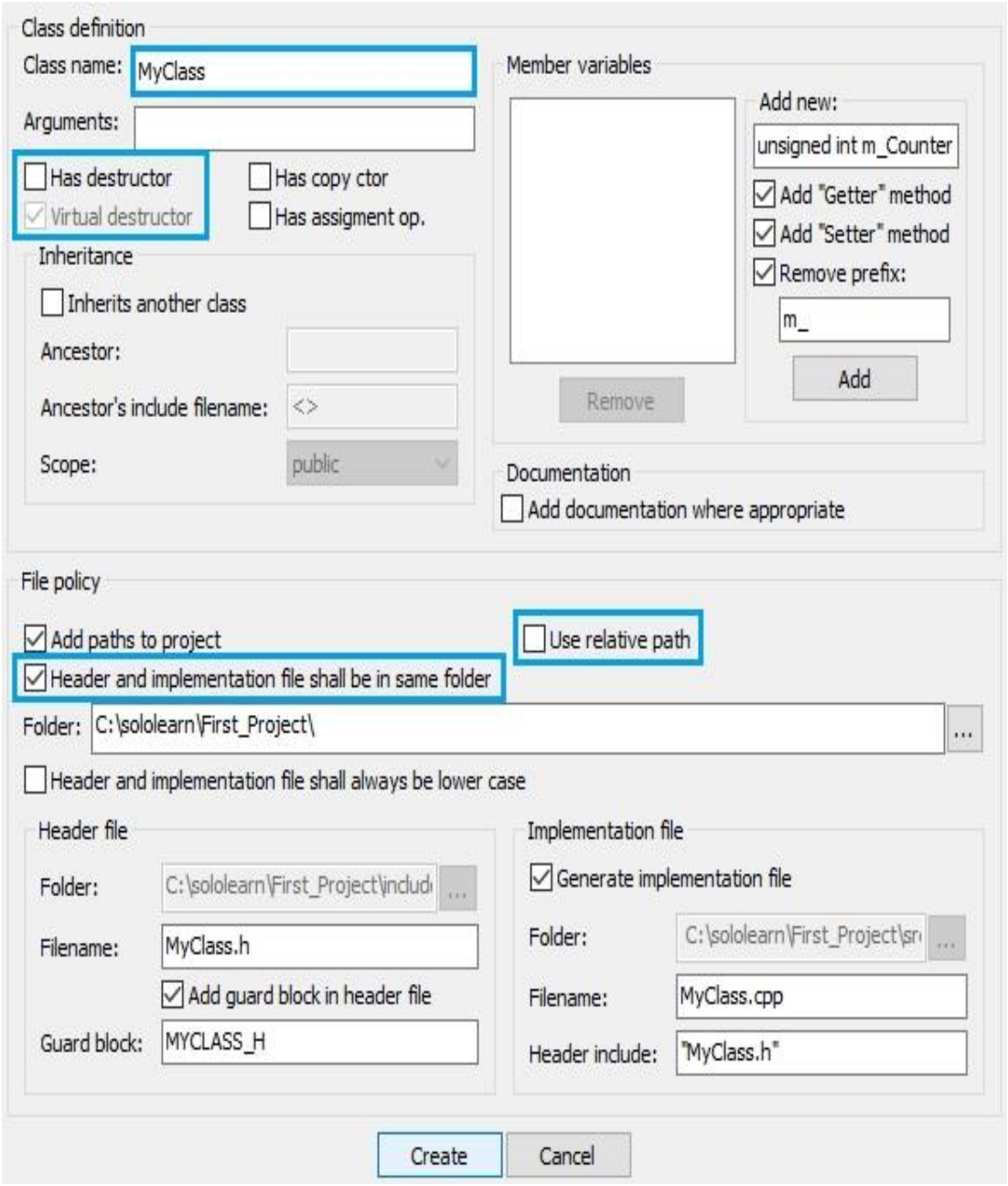


Creating a New Class

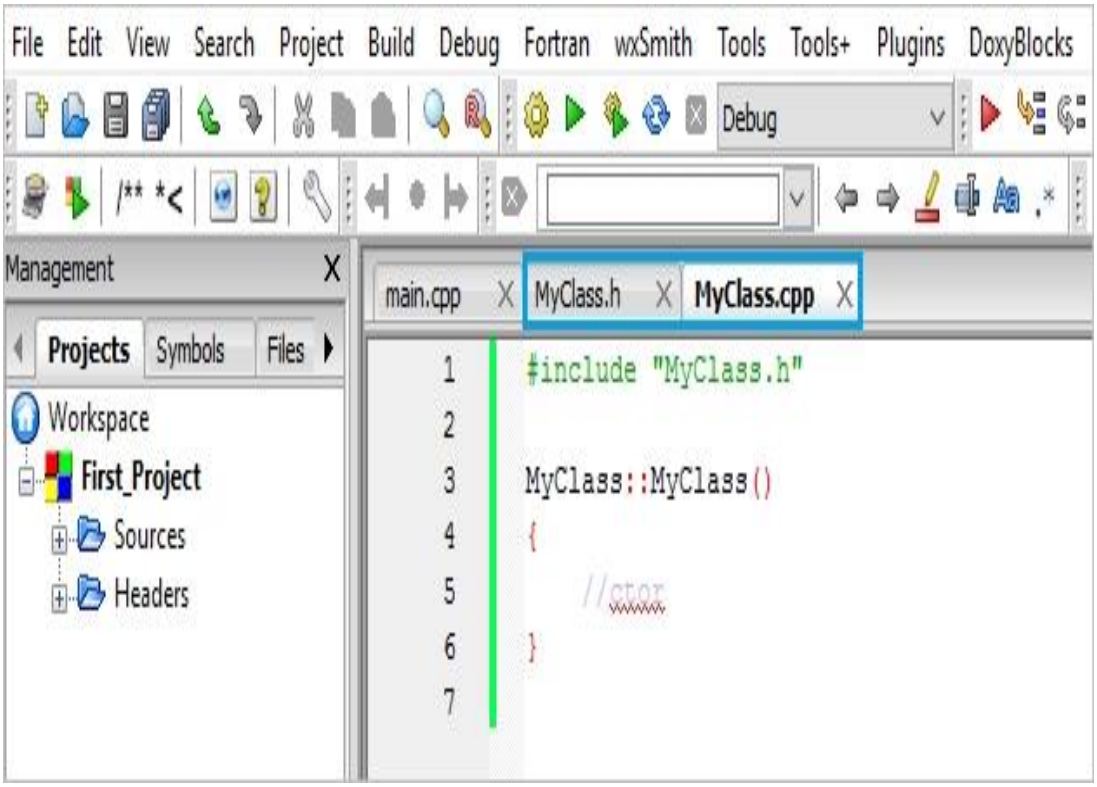
It is generally a good practice to define your new classes in separate files. This makes maintaining and reading the code easier. To do this, use the following steps in CodeBlocks:

Click File->New->Class...

Give your new class a name, uncheck "Has destructor" and check "Header and implementation file shall be in same folder", then click the "Create" button.



Note that two new files have been added to your project:



The new files act as templates for our new class.

- MyClass.h is the header file.

- MyClass.cpp is the source file.

Source & Header

The header file (.h) holds the function declarations (prototypes) and variable declarations.

It currently includes a template for our new MyClass class, with one default constructor.

MyClass.h#ifndef MYCLASS_H

#define MYCLASS_H

class MyClass

{

public:

MyClass();

protected:

private:

};

#endif // MYCLASS_H

The implementation of the class and its methods go into the source file (.cpp). Currently it includes just an empty constructor.

MyClass.cpp#include "MyClass.h"

MyClass::MyClass()

{

//ctor

}

Scope Resolution Operator

The double colon in the source file (.cpp) is called the scope resolution operator, and it's used for the constructor definition:#include "MyClass.h"

MyClass::MyClass()

{

//ctor

}

The scope resolution operator is used to define a particular class' member functions, which have already been declared. Remember that we defined the constructor prototype in the header file. So, basically, MyClass::MyClass() refers to the MyClass() member function - or, in this case, constructor - of the MyClass class.

To use our classes in our main, we need to include the header file.

For example, to use our newly created MyClass in main:#include <iostream>

#include "MyClass.h"

using namespace std;

int main() {

MyClass obj;

}

Destructors

Remember constructors? They're special member functions that are automatically called when an object is created. Destructors are special functions, as well. They're called when an object is destroyed or deleted. Objects are destroyed when they go out of scope, or whenever the delete expression is applied to a pointer directed at an object of a class. The name of a destructor will be exactly the same as the class, only prefixed with a tilde (~). A destructor can't return a value or take any parameters.class MyClass {

public:

~MyClass() {

// some code

}

};

For example, let's declare a destructor for our MyClass class, in its header file MyClass.h:class MyClass

{

public:

MyClass();

```
~MyClass();
};
```

After declaring the destructor in the header file, we can write the implementation in the source file MyClass.cpp:#include "MyClass.h"

```
#include <iostream>
using namespace std;
MyClass::MyClass()
{
cout<<"Constructor"<<endl;
}
MyClass::~MyClass()
{
cout<<"Destructor"<<endl;
}
```

Since destructors can't take parameters, they also can't be overloaded. Each class will have just one destructor.

Let's return to our main.

```
#include <iostream>
#include "MyClass.h"
using namespace std;
int main() {
MyClass obj;
return 0;
}
```

We included the class' header file and then created an object of that type.
This returns the following output:Constructor

Destructor

When the program runs, it first creates the object and calls the constructor. The object is deleted and the destructor is called when the program's execution is completed. Remember that we printed "Constructor" from the constructor and "Destructor" from the destructor.

#ifndef & #define

We created separate header and source files for our class, which resulted in this header file.#ifndef MYCLASS_H

#define MYCLASS_H

class MyClass

{

public:

MyClass();

protected:

private:

};

#endif // MYCLASS_H

ifndef stands for "if not defined". The first pair of statements tells the program to define the MyClass header file if it has not been defined already. endif ends the condition.

Member Functions

Let's create a sample function called myPrint() in our class.

MyClass.hclass MyClass

{

public:

MyClass();

void myPrint();

};

MyClass.cpp#include "MyClass.h"

#include <iostream>

using namespace std;

MyClass::MyClass() {

}

void MyClass::myPrint() {

cout <<"Hello"<<endl;

}

Dot Operator

Next, we'll create an object of the type MyClass, and call its myPrint() function using the dot (.) operator:

#include "MyClass.h"

int main() {

MyClass obj;

obj.myPrint();

}

// Outputs "Hello"

Pointers

We can also use a pointer to access the object's members. The following pointer points to the obj object:MyClass obj;

MyClass *ptr = &obj;

Selection Operator

The arrow member selection operator (->) is used to access an object's members with a pointer.

MyClass obj;

MyClass *ptr = &obj;

ptr->myPrint();

Constants

A constant is an expression with a fixed value. It cannot be changed while the program is running.

Use the const keyword to define a constant variable.const int x = 42;

Constant Objects

As with the built-in data types, we can make class objects constant by using the const keyword.const MyClass obj;

All const variables must be initialized when they're created. In the case of classes, this initialization is done via constructors. If a class is not initialized using a parameterized constructor, a public default constructor must be provided - if no public default

constructor is provided, a compiler error will occur. Once a const class object has been initialized via the constructor, you cannot modify the object's member variables. This includes both directly making changes to public member variables and calling member functions that set the value of member variables.

As with the built-in data types, we can make class objects constant by using the const keyword.const MyClass obj;
All const variables must be initialized when they're created. In the case of classes, this initialization is done via constructors. If a class is not initialized using a parameterized constructor, a public default constructor must be provided - if no public default constructor is provided, a compiler error will occur. Once a const class object has been initialized via the constructor, you cannot modify the object's member variables. This includes both directly making changes to public member variables and calling member functions that set the value of member variables.

Only non-const objects can call non-const functions. A constant object can't call regular functions. Hence, for a constant object to work you need a constant function.

To specify a function as a const member, the const keyword must follow the function prototype, outside of its parameters' closing parenthesis. For const member functions that are defined outside of the class definition, the const keyword must be used on both the function prototype and definition. For example:
MyClass.hclass MyClass
{
public:
void myPrint() const;

```
};
MyClass.cpp#include "MyClass.h"
#include <iostream>
using namespace std;
void MyClass::myPrint() const {
cout <<"Hello"<<endl;
}
```

Now the myPrint() function is a constant member function. As such, it can be called by our constant object:

```
int main() {
const MyClass obj;
obj.myPrint();
}
// Outputs "Hello"
```

Attempting to call a regular function from a constant object results in an error. In addition, a compiler error is generated when any const member function attempts to change a member variable or to call a non-const member function.

Member Initializers

Recall that constants are variables that cannot be changed, and that all const variables must be initialized at time of creation. C++ provides a handy syntax for initializing members of the class called the member initializer list (also called a constructor initializer).

Consider the following class:class MyClass {

public:

MyClass(int a, int b) {

regVar = a;

constVar = b;

}

private:

int regVar;

const int constVar;

};

This class has two member variables, regVar and constVar. It also has a constructor that takes two parameters, which are used to initialize the member variables.

Running this code returns an error, because one of its member variables is a constant, which cannot be assigned a value after declaration. In cases like this one, a member initialization list can be used to assign values to the member variables.class MyClass {

public:

MyClass(int a, int b)

: regVar(a), constVar(b)

{

}

private:

int regVar;

const int constVar;

};

Let's write the previous example using separate header and source files. MyClass.hclass MyClass {

public:

MyClass(int a, int b);

private:

int regVar;

const int constVar;

};

MyClass.cppMyClass::MyClass(int a, int b)

: regVar(a), constVar(b)

{

cout << regVar << endl;

cout << constVar << endl;

}

We have added cout statements in the constructor to print the values of the member variables.

Our next step is to create an object of our class in main, and use the constructor to assign values.

#include "MyClass.h"

int main() {

MyClass obj(42, 33);

}

/*Outputs

42

33
*/

The member initialization list may be used for regular variables, and must be used for constant variables. Even in cases in which member variables are not constant, it makes good sense to use the member initializer syntax.

Composition

In the real world, complex objects are typically built using smaller, simpler objects. For example, a car is assembled using a metal frame, an engine, tires, and a large number of other parts. This process is called composition.

In C++, object composition involves using classes as member variables in other classes. This sample program demonstrates composition in action.

It contains Person and Birthday classes, and each Person will have a Birthday object as its member.

```
Birthday:class Birthday {
public:
Birthday(int m, int d, int y)
: month(m), day(d), year(y)
{
}
private:
int month;
int day;
int year;
```

```
};
```

Let's also add a printDate() function to our Birthday class:

```
class Birthday {
public:
Birthday(int m, int d, int y)
: month(m), day(d), year(y)
{
}
void printDate()
{
cout<<month<<"/"<<day
<<"/"<<year<<endl;
}
private:
int month;
int day;
int year;
};
```

Next, we can create the Person class, which includes the Birthday class.

```
#include <string>
#include "Birthday.h"
class Person {
public:
Person(string n, Birthday b)
: name(n),
bd(b)
```

{
}
private:
string name;
Birthday bd;
};

Now, our Person class has a member of type Birthday:class Person { public:
Person(string n, Birthday b)
: name(n),
bd(b)
{
}
private:
string name;
Birthday bd;
};

Friend Functions

Normally, private members of a class cannot be accessed from outside of that class. However, declaring a non-member function as a friend of a class allows it to access the class' private members. This is accomplished by including a declaration of this external function within the class, and preceding it with the keyword friend.
In the example below, someFunc(), which is not a member function of the class, is a friend of MyClass and can access its private members.class MyClass {

```
public:
MyClass() {
regVar = 0;
}
private:
int regVar;
friend void someFunc(MyClass &obj);
};
```

The function someFunc() is defined as a regular function outside the class. It takes an object of type MyClass as its parameter, and is able to access the private data members of that object.class MyClass {

```
public:
MyClass() {
regVar = 0;
}
private:
int regVar;
friend void someFunc(MyClass &obj);
};
void someFunc(MyClass &obj) {
obj.regVar = 42;
cout << obj.regVar;
}
```

This

Every object in C++ has access to its own address through an important pointer called the this pointer. Inside a member function this may be used to refer to the invoking object.

Let's create a sample class:class MyClass {

public:

MyClass(int a) : var(a)

{

}

private:

int var;

};

The printInfo() method offers three alternatives for printing the member variable of the class.class MyClass {

public:

MyClass(int a) : var(a)

{

}

void printInfo() {

cout << var<<endl;

cout << this->var<<endl;

cout << (*this).var<<endl;

}

private:

int var;

};

To see the result, we can create an object of our class and call the member function.

```
#include <iostream>
using namespace std;
class MyClass {
public:
MyClass(int a) : var(a)
{
}
void printInfo() {
cout << var <<endl;
cout << this->var <<endl;
cout << (*this).var <<endl;
}
private:
int var;
};

int main() {
MyClass obj(42);
obj.printInfo();
}

/* Outputs
```

42

42

42

*/

Operator Overloading

Most of the C++ built-in operators can be redefined or overloaded. Thus, operators can be used with user-defined types as well (for example, allowing you to add two objects together).

This chart shows the operators that can be overloaded.

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new[]	delete	delete[]

Overloaded operators are functions, defined by the keyword operator followed by the symbol for the operator being defined. An overloaded operator is similar to other functions in that it has a return type and a parameter list. In our example we will be

overloading the + operator. It will return an object of our class and take an object of our class as its parameter.class MyClass {

```
public:
int var;
MyClass()
{
}
MyClass(int a)
: var(a) { }
MyClass operator+(MyClass &obj) {
}
};
```