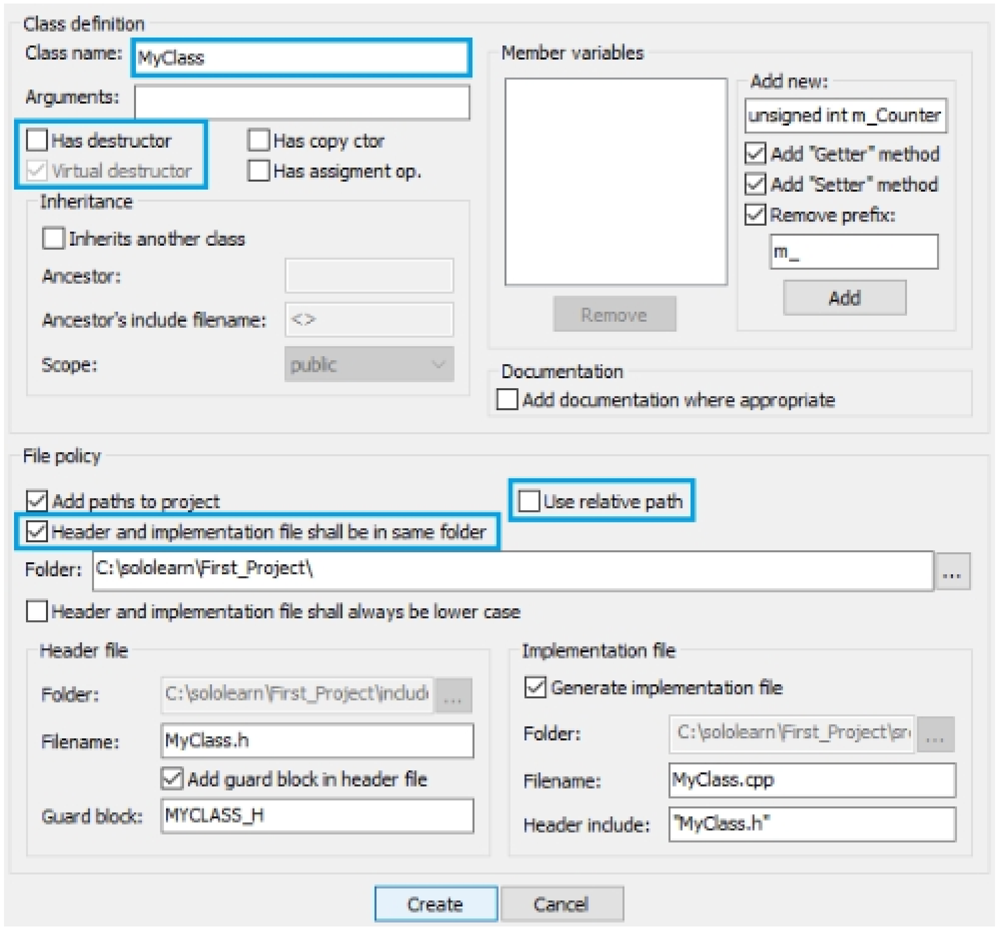
More On Classes

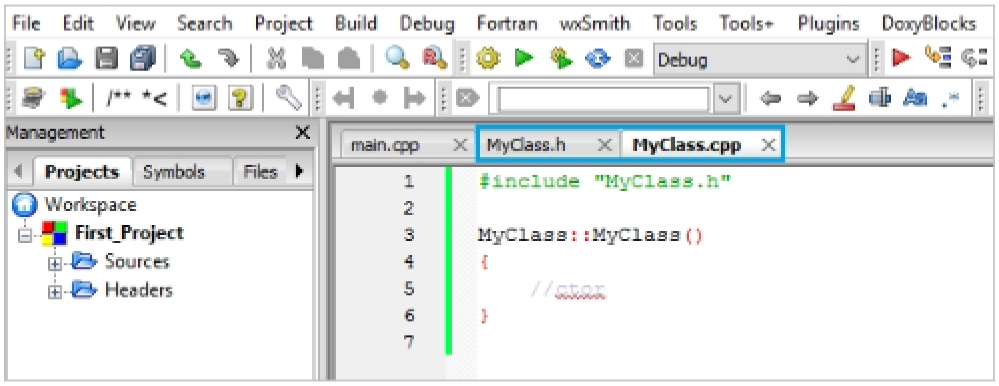
Creating a New Class

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.



Class definition
Class name: MyClass
Arguments:
[ ] Has destructor
[✓] Virtual destructor
[ ] Has copy ctor
[ ] Has assigment op.
Inheritance
[ ] Inherits another class
Ancestor:
Ancestor's include filename: <>
Scope: public
Member variables
Add new: unsigned int m_Counter
[✓] Add "Getter" method
[✓] Add "Setter" method
[✓] Remove prefix: m_
Add
Remove
Documentation
[ ] Add documentation where appropriate
File policy
[✓] Add paths to project
[ ] Use relative path
[✓] Header and implementation file shall be in same folder
Folder: C:\sololearn\First_Project\
[ ] Header and implementation file shall always be lower case
Header file
Folder: C:\sololearn\First_Project\includ
Filename: MyClass.h
[✓] Add guard block in header file
Guard block: MYCLASS_H
Implementation file
[✓] Generate implementation file
Folder: C:\sololearn\First_Project\sr
Filename: MyClass.cpp
Header include: "MyClass.h"
Create Cancel

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



```
#include "MyClass.h"

MyClass::MyClass()
{
    //ctor
}
```

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
}
```

The #ifndef and #define statements in the header file will be discussed in the upcoming lessons.

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.

Source & Header

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;
}
```

The **header** declares "what" a class (or whatever is being implemented) will do, while the **cpp source** file defines "how" it will perform those features.

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.

Destructors

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
    }
};
```

Destructors can be very useful for releasing resources before coming out of the program. This can include closing files, releasing memory, and so on.

Destructors

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

```
class MyClass
{
  public:
   MyClass();
   ~MyClass();
};
```

Declare a destructor for our MyClass class.

Destructors

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;
}
```

Note that we included the <iostream> header, so that we can use **cout**.

Destructors

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

Defining a destructor is not mandatory; if you don't need one, you don't have to define one.

Destructors

Let's return to our main.

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

int main() {
  MyClass obj;

  return 0;
}
```

Try It Yourself

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.

> This prevents a header file from being included more than once within one file.

Member Functions

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

```
class MyClass
{
  public:
   MyClass();
   void myPrint();
};
```

MyClass.cpp

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

MyClass::MyClass() {
}

void MyClass::myPrint() {
  cout <<"Hello"<<endl;
}
```

> Since **myPrint()** is a regular member function, it's necessary to specify its **return type** in both the declaration and the definition.

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"
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

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;
```

The type of the pointer is **MyClass**, as it points to an object of that type.

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();
```

Try It Yourself

When working with an object, use the **dot** member selection operator (.).
When working with a pointer to the object, use the **arrow** member selection operator (->).

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;
```

All constant variables **must** be initialized at the time of their creation.

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.

When you've used **const** to declare an object, you can't change its data members during the object's lifetime.

Constant Objects

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.h

```
class 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"
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

Constant Objects

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.

Defining constant objects and functions ensures that corresponding data members cannot be unexpectedly modified.

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**).

Tap **Continue** to learn more!

Member Initializers

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;
};
```

Note that in the syntax, the initialization list follows the constructor parameters. The list begins with a **colon** (:), and then lists each variable to be initialized, along with the value for that variable, with a comma to separate them.
Use the syntax **variable(value)** to assign values.

> The initialization list eliminates the need to place explicit assignments in the constructor body. Also, the initialization list does not end with a semicolon.

Member Initializers

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

```
class MyClass {
  public:
   MyClass(int a, int b);
  private:
   int regVar;
   const int constVar;
};
```

MyClass.cpp

```
MyClass::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 constructor is used to create the object, assigning two parameters to the member variables via the member initialization list.

Member Initializers

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;
};
```

Our **Birthday** class has three member variables. It also has a constructor that initializes the members using a member initialization list.

> The class was declared in a single file for the sake of simplicity. Alternatively, you could use header and source files.

Composition

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;
};
```

Add a printDate() function to our Birthday class.

Composition

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;
};
```

The Person class has a **name** and a **Birthday** member, and a constructor to initialize them.
Ensure that the corresponding header files are included.

More on **Composition** coming up in the next lesson!

Composition

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;
};
```

Composition is used for objects that share a **has-a** relationship, as in "A **Person** has a **Birthday**".

Composition

Let's add a **printInfo()** function to our Person class, that prints the data of the object:

```
class Person {
 public:
  Person(string n, Birthday b)
  : name(n),
  bd(b)
  {
  }
  void printInfo()
  {
   cout << name << endl;
   bd.printDate();
  }
 private:
  string name;
  Birthday bd;
};
```

Notice that we can call the **bd** member's **printDate()** function, since it's of type **Birthday**, which has that function defined.

Composition

Now that we've defined our **Birthday** and **Person** classes, we can go to our main, create a **Birthday** object, and then pass it to a **Person** object.

```
int main() {
  Birthday bd(2, 21, 1985);
  Person p("David", bd);
  p.printInfo();
}

/*Outputs
David
2/21/1985
*/
```

Try It Yourself

We've created a **Birthday** object for the date of 2/21/1985. Next, we created a **Person** object and passed the **Birthday** object to its constructor. Finally, we used the **Person** object's **printInfo()** function to print its data.

In general, **composition** serves to keep each individual class relatively simple, straightforward, and focused on performing one task. It also enables each sub-object to be self-contained, allowing for reusability (we can use the **Birthday** class within various other classes).

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);
};
```

Note that when passing an object to the function, we need to pass it **by reference**, using the & operator.

Friend Functions

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;
}
```

The **someFunc()** function changes the private member of the object and prints its value.

To make its members accessible, the class has to declare the function as a **friend** in its definition. You cannot "make" a function a friend to a class without the class "giving away" its friendship to that function.

Friend Functions

Now we can create an object in main and call the **someFunc()** function:

```
int main() {
  MyClass obj;
  someFunc(obj);
}

//Outputs 42
```

Try It Yourself

someFunc() had the ability to modify the private member of the object and print its value.

Typical use cases of **friend** functions are operations that are conducted between two different classes accessing private members of both.

You can declare a function **friend** across any number of classes.
Similar to friend functions, you can define a **friend class**, which has access to the private members of another class.

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;
};
```

Friend functions do not have a **this** pointer, because friends are not members of a class.

This

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;
};
```

All three alternatives will produce the same result.

this is a **pointer** to the object, so the arrow selection operator is used to select the member variable.

This

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
*/
```

Try It Yourself

All three of the ways to access the member variable work.

Note that only member functions have a **this** pointer.

This

You may be wondering why it's necessary to use the **this** keyword, when you have the option of directly specifying the variable.

The **this** keyword has an important role in **operator overloading**, which will be covered in the following lesson.

Tap **Continue** to learn more!

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[]

Operators that can't be overloaded include :: | .* | . | ?:

Operator Overloading

Let's declare a sample class to demonstrate operator overloading:

```
class MyClass {
 public:
  int var;
  MyClass() {}
  MyClass(int a)
  : var(a) { }
};
```

Our class has two constructors and one member variable.

We will be overloading the + operator, to enable adding two objects of our class together.

Operator Overloading

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) {
  }
};
```

Now, we need to define what the function does.

Operator Overloading

We need our + operator to return a new **MyClass** object with a member variable equal to the sum of the two objects' member variables.

```
class MyClass {
 public:
  int var;
  MyClass() {}
  MyClass(int a)
  : var(a) { }

  MyClass operator+(MyClass &obj) {
   MyClass res;
   res.var= this->var+obj.var;
   return res;
  }
};
```

Here, we declared a new **res** object. We then assigned the sum of the member variables of the current object (**this**) and the parameter object (**obj**) to the **res** object's var member variable. The **res** object is returned as the result.

This gives us the ability to create objects in main and use the overloaded + operator to add them together.

```
int main() {
  MyClass obj1(12), obj2(55);
  MyClass res = obj1+obj2;

  cout << res.var;
}

//Outputs 67
```

Try It Yourself

With overloaded operators, you can use any custom logic needed. However, it's not possible to alter the operators' precedence, grouping, or number of operands.

End.