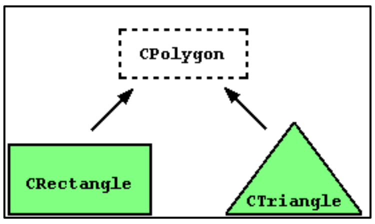
Note: All the code that you write for the following labs must be uploaded to Github.

C++ Inheritance

Classes in C++ can be extended, creating new classes which retain characteristics of the base class. This process, known as inheritance, involves a base class and a derived class: The derived class inherits the members of the base class, on top of which it can add its own members.

For example, let's imagine a series of classes to describe two kinds of polygons: rectangles and triangles. These two polygons have certain common properties, such as the values needed to calculate their areas: they both can be described simply with a height and a width (or base).

This could be represented in the world of classes with a class Polygon from which we would derive the two other ones: Rectangle and Triangle:



The Polygon class would contain members that are common for both types of polygon. In our case: width and height. And Rectangle and Triangle would be its derived classes, with specific features that are different from one type of polygon to the other.

Classes that are derived from others inherit all the accessible members of the base class. That means that if a base class includes a member A and we derive a class from it with another member called B, the derived class will contain both member A and member B.

The inheritance relationship of two classes is declared in the derived class. Derived classes definitions use the following syntax:

class derived_class_name: public base_class_name

{ /*...*/ };

Where **derived_class_name** is the name of the derived class and base_class_name is the name of the class on which it is based. The public access specifier may be replaced by any one of the other access specifiers (protected or private). This access specifier limits the most accessible level for the members inherited from the base class: The members with a more accessible level are inherited with this level instead, while the members with an equal or more restrictive access level keep their restrictive level in the derived class.

Create a new program using Visual studio Code and do the following:

Within a file titled **Polygon.h**, write the following code.

```
#pragma once

class Polygon {
  protected:
    int mWidth
    int mHeight;
  public:
    void SetValues(int width, int height);
 };
```

Within a file titled **Polygon.cpp**, write the following code.

```
#pragma once

#include “Polygon.h”

void Polygon::SetValues(int width, int height)
{
   mWidth = width;
   mHeight = height;
}
```

Within a file titled **Rectangle.h**, write the following code

```
#pragma once

#include “Polygon.h”

class Rectangle : public Polygon {
  public:
    int Area();
 };
```

Within a file titled **Rectangle.cpp**, write the following code

```
#include “Rectangle.h”

int Rectangle:: Area()
{
   return mWidth * mHeight;
}
```

Within a file titled **Triangle.h**, write the following code

```
#include “Polygon.h”

class Triangle : public Polygon {
  public:
    int Area();
  };
```

Within a file titled **Triangle.cpp**, write the following code

```
#include “Triangle.h”

int Triangle::Area()
{
   return mWidth * mHeight / 2;
}
```

Within a file titled **main.cpp**, write the following code and run the program

```
#include <iostream>
#include “Rectangle.h”
#include “Triangle.h”

using namespace std;

int main() {
  Rectangle rect;
  Triangle trgl;
  rect.SetValues(4,5);
  trgl.SetValues (4,5);
  cout << rect.Area() << '\n';
  cout << trgl.Area() << '\n';
  return 0;
}
```

The objects of the classes **Rectangle** and **Triangle** each contain members inherited from Polygon. These are**: mWidth**, **mHeight** and **SetValues**.

The protected access specifier used in class Polygon is similar to private. Its only difference occurs in fact with inheritance: When a class inherits another one, the members of the derived class can access the protected members inherited from the base class, but not its private members.

By declaring width and height as protected instead of private, these members are also accessible from the derived classes Rectangle and Triangle, instead of just from members of Polygon. If they were public, they could be accessed just from anywhere.

We can summarize the different access types according to which functions can access them in the following way:

Access	**public**	**protected**	**private**
members of the same class	yes	yes	yes
members of derived class	yes	yes	no
not members	yes	no	no

Where "not members" represents any access from outside the class, such as from main, from another class or from a function.

In the example above, the members inherited by Rectangle and Triangle have the same access permissions as they had in their base class Polygon:

```
Polygon::width            // protected access
Rectangle::width          // protected access

Polygon::SetValues()    // public access
Rectangle::SetValues()  // public access
```

This is because the inheritance relation has been declared using the **public** keyword on each of the derived classes:

```
class Rectangle: public Polygon { /* ... */ }
```

This public keyword after the colon (:) denotes the most accessible level the members inherited from the class that follows it (in this case **Polygon**) will have from the derived class (in this case **Rectangle**). Since public is the most accessible level, by specifying this keyword the derived class will inherit all the members with the same levels they had in the base class.

With **protected**, all public members of the base class are inherited as protected in the derived class. Conversely, if the most restricting access level is specified (**private**), all the base class members are inherited as private.

For example, if daughter were a class derived from mother that we defined as:

```
class Daughter : protected Mother;
```

This would set protected as the less restrictive access level for the members of Daughter that it inherited from mother. That is, all members that were public in Mother would become protected in Daughter. Of course, this would not restrict Daughter from declaring its own public members. That less restrictive access level is only set for the members inherited from Mother.

If no access level is specified for the inheritance, the compiler assumes private for classes declared with keyword class and public for those declared with **struct**.

Most use cases of inheritance in C++ should use **public** inheritance. When other access levels are needed for base classes, they can usually be better represented as member variables instead.

In principle, a publicly derived class inherits access to every member of a base class except:

- its constructors and its destructor
- its assignment operator members (operator=)

Even though access to the constructors and destructor of the base class is not inherited as such, they are automatically called by the constructors and destructor of the derived class.

Unless otherwise specified, the constructors of a derived class calls the default constructor of its base classes (i.e., the constructor taking no arguments). Calling a different constructor of a base class is possible, using the same syntax used to initialize member variables in the initialization list:

derived_constructor_name (parameters) : base_constructor_name (parameters) {...}

Create a new program using Visual studio Code and do the following:

Within a file titled **Mother.h**, write the following code:

```
#pragma once

class Mother {
  public:
    Mother();
    Mother(int a);
};
```

Within a file titled **Mother.cpp**, write the following code:

```
#include “Mother.h”
#include <iostream>
using namespace std;

Mother::Mother()
{
  cout << "Mother: no parameters\n";
}
Mother::Mother(int a)
{
  cout << "Mother: int parameter\n";
}
```

Within a file titled **Daughter.h**, write the following code

```
#pragma once

#include “Mother.h”

class Daughter : public Mother {
  public:
    Daughter (int a);
};
```

Within a file titled **Daughter.cpp**, write the following code

```
#include “Daughter.h”
#include <iostream>
using namespace std;

Daughter::Daughter(int a)
{
  cout << "Daughter: int parameter\n\n";
}
```

Within a file titled **Son.h**, write the following code

```
#pragma once

#include “Mother.h”

class Son : public Mother {
  public:
    Son(int a);
};
```

Within a file titled **Son.cpp**, write the following code

```
#include “Son.h”
#include <iostream>
using namespace std;

Son::Son(int a) : Mother(a)
{
   cout << "Son: int parameter\n\n";
}
```

Within a file titled **main.cpp**, write the following code and run the program

```
#include <iostream>
#include “Daughter.h”
#include “Son.h”

using namespace std;

int main ()
{

  Daughter theDaughter(1);
  Son theSon(2);

  return 0;
}
```

Notice the difference between which Mother's constructor is called when a new Daughter object is created and which when it is a Son object. The difference is due to the different constructor declarations of Daughter and Son:

```
Daughter (int a) // nothing specified: call default constructor
Son (int a) : Mother (a) // constructor specified: call this specific constructor
```

C++ Polymorphism

One of the key features of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class. Polymorphism is the art of taking advantage of this simple but powerful and versatile feature.

The example about the rectangle and triangle classes can be rewritten using pointers taking this feature into account:

Rewrite the main function in the program with rectangles and triangles as follows:

```
#include <iostream>
#include “Rectangle.h”
#include “Triangle.h”

using namespace std;

int main() {

Rectangle rect;
  Triangle trgl;
  Polygon* pPoly1 = &rect;
  Polygon* pPoly2 = &trgl;
  pPoly1->SetValues(4,5);
  pPoly2->SetValues(4,5);

  cout << rect.Area() << '\n';
  cout << trgl.Area() << '\n';

  return 0;
}
```

Function **main** declares two pointers to Polygon (named **pPoly1** and **pPoly2**). These are assigned the addresses of **rect** and **trgl**, respectively, which are objects of type Rectangle and Triangle. Such assignments are valid, since both Rectangle and Triangle are classes derived from Polygon.

Dereferencing **pPoly1** and **pPoly2** (with **pPoly1->** and **pPoly2->**) is valid and allows us to access the members of their pointed objects. For example, the following two statements would be equivalent in the previous example:

```
pPoly1->SetValues(4,5);
rect.SetValues(4,5);
```

But because the type of both **pPoly1** and **pPoly2** is pointer to Polygon (and not pointer to Rectangle nor pointer to Triangle), only the members inherited from Polygon can be accessed, and not those of the derived classes Rectangle and Triangle. That is why the program above accesses the area members of both objects using rect and trgl directly, instead of the pointers; the pointers to the base class cannot access the area members.

Member **area** could have been accessed with the pointers to Polygon if area were a member of Polygon instead of a member of its derived classes, but the problem is that Rectangle and Triangle implement different versions of area, therefore there is not a single common version that could be implemented in the base class.

Virtual Members

A virtual member is a member function that can be redefined in a derived class, while preserving its calling properties through references. The syntax for a function to become virtual is to precede its declaration with the virtual keyword.

Rewrite the Polygon class in the program with rectangles and triangles as follows:

Within the file titled **Polygon.h**, write the following code.

```
#pragma once

class Polygon {
  protected:
    int mWidth
    int mHeight;
  public:
    void SetValues(int width, int height);
    virtual int Area();
 };
```

Within the file titled **Polygon.cpp**, write the following code.

```
#include “Polygon.h”

void Polygon::SetValues(int width, int height)
{
   mWidth = width;
   mHeight = height;
}

int Polygon::Area()
{
   return 0;
}
```

Within the file titled **main.cpp**, write the following code and run the program

```
#include <iostream>
#include “Rectangle.h”
#include “Triangle.h”

using namespace std;

int main()
{
  Rectangle rect;
  Triangle trgl;
  Polygon poly;
  Polygon* pPoly1 = &rect;
  Polygon* pPoly2 = &trgl;
  Polygon* pPoly3 = &poly;
  pPoly1->SetValues(4,5);
  pPoly2->SetValues(4,5);
  pPoly3->SetValues(4,5);
  cout << pPoly1->Area() << '\n';
  cout << pPoly2->Area() << '\n';
  cout << pPoly3->Area() << '\n';

  return 0;
}
```

In this example, all three classes (Polygon, Rectangle and Triangle) have the same members: **mWidth**, **mHeight**, and functions **SetValues** and **Area**.

The member function area has been declared as virtual in the base class because it is later redefined in each of the derived classes. Non-virtual members can also be redefined in derived classes, but non-virtual members of derived classes cannot be accessed through a reference of the base class: i.e., if virtual is removed from the declaration of area in the example above, all three calls to area would return zero, because in all cases, the version of the base class would have been called instead.

Therefore, essentially, what the virtual keyword does is to allow a member of a derived class with the same name as one in the base class to be appropriately called from a pointer, and more precisely when the type of the pointer is a pointer to the base class that is pointing to an object of the derived class, as in the above example.

A class that declares or inherits a virtual function is called a polymorphic class.

Note that despite of the virtuality of one of its members, Polygon was a regular class, of which even an object was instantiated (poly), with its own definition of member area that always returns 0.

Abstract Base Classes

Abstract base classes are something very similar to the Polygon class in the previous example. They are classes that can only be used as base classes, and thus are allowed to have virtual member functions without definition (known as pure virtual functions). The syntax is to replace their definition by =0 (an equal sign and a zero).

An abstract base Polygon class declaration could look like this:

```
#pragma once

class Polygon {
  protected:
    int mWidth
    int mHeight;
  public:
    void SetValues(int width, int height);
    virtual int Area() = 0;
 };
```

And the implementation could look like this:

```
#include "Polygon.h"

void Polygon::SetValues(int width, int height)
{
   mWidth = width;
   mHeight = height;
}
```

Notice that **Area** has no definition; this has been replaced by =0, which makes it a pure virtual function. Classes that contain at least one pure virtual function are known as abstract base classes.

Abstract base classes cannot be used to instantiate objects. Therefore, this last abstract base class version of Polygon could not be used to declare objects like:

```
Polygon mypolygon; // not working if Polygon is abstract base class
```

However, an abstract base class is not completely useless. It can be used to create pointers to it, and take advantage of all its polymorphic abilities. For example, the following pointer declarations would be valid:

```
Polygon* pPoly1;
Polygon* pPoly2;
```

And can actually be dereferenced when pointing to objects of derived (non-abstract) classes.

Rewrite the Polygon class in the program with rectangles and triangles as follows:

Within the file titled **Polygon.h**, write the following code.

```
#pragma once

class Polygon {
  protected:
    int mWidth
    int mHeight;
  public:
    void SetValues(int width, int height);
    virtual int Area() = 0;
 };
```

Within the file titled **Polygon.cpp**, write the following code.

```
#include “Polygon.h”

void Polygon::SetValues(int width, int height)
{
   mWidth = width;
   mHeight = height;
}
```

Within the file titled **main.cpp**, write the following code and run the program.

```
#include <iostream>
#include “Rectangle.h”
#include “Triangle.h”

using namespace std;

int main() {

  Rectangle rect;
  Triangle trgl;
  Polygon* pPoly1 = &rect;
  Polygon* pPoly2 = &trgl;

  pPoly1->SetValues(4,5);
  pPoly2->SetValues(4,5);

  cout << rect.Area() << '\n';
  cout << trgl.Area() << '\n';

  return 0;
}
```

In this example, objects of different but related types are referred to using a unique type of pointer (Polygon*) and the proper member function is called every time, just because they are virtual. This can be really useful in some circumstances.

In addition, it is even possible for a member of the abstract base class Polygon to use the special pointer **this** to access the proper virtual members, even though Polygon itself has no implementation for this function. In this regard, rewrite the Polygon class in the program with rectangles and triangles as follows:

Within the file titled **Polygon.h**, write the following code.

```
#pragma once

class Polygon {
  protected:
    int mWidth
    int mHeight;
  public:
    void SetValues(int width, int height);
    virtual int Area() = 0;
    void PrintArea();
 };
```

Within the file titled **Polygon.cpp**, write the following code.

```
#include “Polygon.h”
#include <iostream>

using namespace std;


void Polygon::SetValues(int width, int height)
{
   mWidth = width;
   mHeight = height;
}

void Polygon::PrintArea()
{
   cout << this->area() << '\n';
}
```

Within the file titled **main.cpp**, write the following code and run the program.

```
#include <iostream>
#include “Rectangle.h”
#include “Triangle.h”

using namespace std;

int main() {

  Rectangle rect;
  Triangle trgl;
  Polygon * pPoly1 = &rect;
  Polygon * pPoly2 = &trgl;

  pPoly1->SetValues(4,5);
  pPoly2->SetValues(4,5);

  pPoly1->PrintArea();
  pPoly2->PrintArea();

  return 0;
}
```

Virtual members and abstract classes grant C++ polymorphic characteristics, most useful for object-oriented projects. Of course, the examples above are very simple use cases, but these features can be applied to arrays of objects or dynamically allocated objects. In this regard, rewrite the main function in the previous example, as follows:

```
#include <iostream>
#include “Rectangle.h”
#include “Triangle.h”

using namespace std;

int main()
{
  Polygon* pPoly1 = new Rectangle(4,5);
  Polygon* pPoly2 = new Triangle(4,5);

  pPoly1->Printarea();
  pPoly2->Printarea();

  delete pPoly1;
  delete pPoly2;

  return 0;
}
```

Notice that the **pPoly** pointers:

```
Polygon * pPoly1 = new Rectangle(4,5);
Polygon * pPoly2 = new Triangle(4,5);
```

are declared being of type "pointer to Polygon", but the objects allocated have been declared having the derived class type directly (Rectangle and Triangle).

Type Conversions

Implicit conversions are automatically performed when a value is copied to a compatible type. For example:

```
short a = 2000;
int b;
b=a;
```

Here, the value of a is promoted from short to int without the need of any explicit operator. This is known as a standard conversion. Standard conversions affect fundamental data types, and allow the conversions between numerical types (short to int, int to float, double to int...), to or from bool, and some pointer conversions.

Converting to int from some smaller integer type, or to double from float is known as promotion, and is guaranteed to produce the exact same value in the destination type. Other conversions between arithmetic types may not always be able to represent the same value exactly:

If a negative integer value is converted to an unsigned type, the resulting value corresponds to its 2's complement bitwise representation (i.e., -1 becomes the largest value representable by the type, -2 the second largest, ...).

The conversions from/to bool consider false equivalent to zero (for numeric types) and to null pointer (for pointer types); true is equivalent to all other values and is converted to the equivalent of 1.

If the conversion is from a floating-point type to an integer type, the value is truncated (the decimal part is removed). If the result lies outside the range of representable values by the type, the conversion causes undefined behavior.

Otherwise, if the conversion is between numeric types of the same kind (integer-to-integer or floating-to-floating), the conversion is valid, but the value is implementation-specific (and may not be portable).

Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This warning can be avoided with an explicit conversion.

For non-fundamental types, arrays and functions implicitly convert to pointers, and pointers in general allow the following conversions:

Null pointers can be converted to pointers of any type

Pointers to any type can be converted to void pointers.

Pointer upcast: pointers to a derived class can be converted to a pointer of an accessible and unambiguous base class, without modifying its const or volatile qualification.

Type Casting

C++ is a strong-typed language. Many conversions, especially those that imply a different interpretation of the value, require an explicit conversion, known in C++ as type-casting. There exist two main syntaxes for generic type-casting: functional and c-like:

```
double x = 10.3;
int y;
y = int (x);  // functional notation
y = (int) x;  // c-like cast notation
```

The functionality of these generic forms of type-casting is enough for most needs with fundamental data types. However, these operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that -while being syntactically correct- can cause runtime errors. For example, the following code compiles without errors:

```
// class type-casting
#include <iostream>
using namespace std;

class Dummy {
    double i,j;
};

class Addition {
    int x,y;
  public:
    Addition (int a, int b) { x=a; y=b; }
    int result() { return x+y;}
};

int main () {
  Dummy d;
  Addition * padd;
  padd = (Addition*) &d;
  cout << padd->result();
  return 0;
}
```

The program declares a pointer to Addition, but then it assigns to it a reference to an object of another unrelated type using explicit type-casting:

```
padd = (Addition*) &d;
```

Unrestricted explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to. The subsequent call to member result will produce either a run-time error or some other unexpected results.

To control these types of conversions between classes, we have four specific casting operators: dynamic_cast, reinterpret_cast, static_cast and const_cast. Their format is to follow the new type enclosed between angle-brackets (<>) and immediately after, the expression to be converted between parentheses.

dynamic_cast <new_type> (expression)

reinterpret_cast <new_type> (expression)

static_cast <new_type> (expression)

const_cast <new_type> (expression)

The traditional type-casting equivalents to these expressions would be:

(new_type) expression

new_type (expression)

but each one with its own special characteristics.

dynamic_cast

dynamic_cast can only be used with pointers and references to classes. Its purpose is to ensure that the result of the type conversion points to a valid complete object of the destination pointer type.

This naturally includes pointer upcast (converting from pointer-to-derived to pointer-to-base), in the same way as allowed as an implicit conversion.

However, **dynamic_cast** can also downcast (convert from pointer-to-base to pointer-to-derived) polymorphic classes (those with virtual members) if -and only if- the pointed object is a valid complete object of the target type.

In this regard, create a new program using Visual studio Code and do the following:

In a file titled **Base.h,** write the following code**.**

```
#pragma once

class Base {
     virtual void DummyFunction();
 };
```

In a file titled **Base.cpp,** write the following code**.**

```
#include “Base.h”

void Base::DummyFunction
{

}
```

In a file titled **Derived.h**, write the following code.

```
#pragma once

#include “Base.h”

class Derived : public Base {
public:
     Derived();

private:
     int a;
 };
```

In a file titled **Derived.cpp**, write the following code.

```
#include “Derived.h”

Derived::Derived()
{
   a = 0;
}
```

Within the file titled **main.cpp**, write the following code and run the program.

```
#include <iostream>
#include “Derived.h”

using namespace std;

int main ()
{
   Base* pba = new Derived();
   Base* pbb = new Base();
   Derived * pd;

   pd = dynamic_cast<Derived*>(pba);
   if (pd == nullptr)
   {
     cout << "Null pointer on first type-cast.\n";
   }

   pd = dynamic_cast<Derived*>(pbb);
   if (pd == nullptr)
   {
     cout << "Null pointer on second type-cast.\n";
   }

   delete pba;
   pba = nullptr;

   delete pbb;
   pbb = nullptr;

  return 0;
}
```

The code above tries to perform two dynamic casts from pointer objects of type Base* (pba and pbb) to a pointer object of type Derived*, but only the first one is successful. Notice their respective initializations:

```
Base* pba = new Derived();
Base* pbb = new Base();
```

Even though both are pointers of type Base*, pba actually points to an object of type Derived, while pbb points to an object of type Base. Therefore, when their respective type-casts are performed using dynamic_cast, pba is pointing to a full object of class Derived, whereas pbb is pointing to an object of class Base, which is an incomplete object of class Derived.

When dynamic_cast cannot cast a pointer because it is not a complete object of the required class -as in the second conversion in the previous example- it returns a null pointer to indicate the failure. If dynamic_cast is used to convert to a reference type and the conversion is not possible, an exception of type bad_cast is thrown instead.

dynamic_cast can also perform the other implicit casts allowed on pointers: casting null pointers between pointers types (even between unrelated classes).

static_cast

static_cast can perform conversions between pointers to related classes, not only upcasts (from pointer-to-derived to pointer-to-base), but also downcasts (from pointer-to-base to pointer-to-derived). No checks are performed during runtime to guarantee that the object being converted is in fact a full object of the destination type. Therefore, it is up to the programmer to ensure that the conversion is safe. On the other side, it does not incur the overhead of the type-safety checks of dynamic_cast.

```
#include <iostream>
#include “Derived.h”

using namespace std;

int main ()
{
   Base* pBase = new Base();

   Derived* pDerived = static_cast<Derived*>(pBase);

   delete pBase;
   pBase = nullptr;

  return 0;
}
```

static_cast is also able to perform all conversions allowed implicitly (not only those with pointers to classes), and is also able to perform the opposite of these. It can convert integers, floating-point values and enum types to enum types.

Exercise:

Read about:

- reinterpret_cast
- const_cast

References

- https://cplusplus.com/doc/tutorial/inheritance/
- https://cplusplus.com/doc/tutorial/polymorphism/
- https://cplusplus.com/doc/tutorial/typecasting/