

8.1. Inheritance

8.2. The **protected** keyword

8.3. (Delegate) constructors and destructors

8.4. **mutable**, **using**, **friend**, and **delete**

8.5. Polymorphism

8.1. Inheritance <https://isocpp.org/wiki/faq/basics-of-inheritance>

- Generalization: Relationship between a more general class (base class, or superclass) and a more specialized class (subclass)
 - the specialized class is consistent with the base class, but contains additional attributes, operations and/or associations
 - an object of the subclass can be used wherever an object of the super class is permitted
- Generalization is not about just summarizing common properties and behaviors, but about generalizing in the literal sense:
Every object of the subclass is an object of the superclass

8.1. Inheritance

```
class Person {  
    private:  
        std::string name;  
        std::string address;  
        int birthYear;  
    public:  
        void printBadge(); // prints name  
        int getAge(); // returns age in years  
};
```

```
int main() {  
    Staff newStaff;  
    Trainee newTrainee;  
    // newStaff and newTrainee objects  
    // can access Person public method  
    newStaff.printBadge();  
    newTrainee.getAge();  
    return 0;  
}
```

```
class Staff: public Person {  
    private:  
        double salary;  
    public:  
        void setSalary(double s);  
};
```

```
class Trainee: public Person {  
    private:  
        int startDay;  
    public:  
        int daysInTraining();  
};
```

8.2. The **protected** keyword

- Private members (attributes or methods) are only accessible within the class that has defined them
- Public members are accessible from anywhere
- Protected members are accessible in the class that defines them, and in classes that inherit from that class

8.2. The protected keyword: Inaccessible from outside the class..

```
class Person {  
    private:  
        std::string address;  
        int birthYear;  
    protected:  
        std::string name;  
    public:  
        void printBadge(); // prints name  
        int getAge(); // returns age in years  
};
```

```
int main() {  
    Staff newStaff;  
    // newStaff object cannot access  
    // Person protected attributes  
    // or methods from outside class:  
    // std::cout << newStaff.name;  
    // -> ERROR  
    return 0;  
}
```

```
class Staff: public Person {  
    private:  
        double salary;  
    public:  
        void setSalary(double s);  
};
```

```
class Trainee: public Person {  
    private:  
        int startDay;  
    public:  
        int daysInTraining();  
};
```

8.2. The protected keyword: .. but accessible from child classes

```
class Person {  
    private:  
        std::string address;  
        int birthYear;  
    protected:  
        std::string name;  
    public:  
        void printBadge(); // prints name  
        int getAge(); // returns age in years  
};
```

```
int Trainee::daysInTraining() {  
    int returnVal;  
    // Trainee cannot access Person's  
    // private attributes:  
    // returnVal = birthYear -> ERROR  
    // but Trainee can access Person's  
    // protected attributes in class:  
    std::cout << name << std::endl;  
    return (today()-startDay);  
}
```

```
class Staff: public Person {  
    private:  
        double salary;  
    public:  
        void setSalary(double s);  
};
```

```
class Trainee: public Person {  
    private:  
        int startDay;  
    public:  
        int daysInTraining();  
};
```

8.3. (Delegate) constructors and destructors

Remember the special syntax in constructors, to initialize attributes:

```
class Example {
public:
    Example(int & aVar);
private:
    const int aConst;    // a constant
    int & aRef;           // a reference
};
// This would lead to errors:
// Example::Example(int &aVar) {
//     aConst = 47; aRef = aVar;
// }
// But this works:
Example::Example(int & aVar)
    : aConst(47), aRef(aVar) {
    // here can follow more code
}
```

Passing arguments to the base class constructor is possible with:

```
class BaseClass {
public:
    BaseClass(int var) : var(var) {}
private:
    int var;
};

class SubClass : public BaseClass {
public:
    SubClass(bool myBool)
        : BaseClass(7), myBool(myBool) {}
private:
    bool myBool;
};
```

8.3. (Delegate) constructors and destructors

Passing arguments to a base class constructor is possible with:

```
class BaseClass {
public:
    BaseClass(int var) : var(var) {}
private:
    int var;
};

class SubClass : public BaseClass {
public:
    SubClass(bool myBool)
        : BaseClass(7), myBool(myBool) {}
private:
    bool myBool;
};
```

Similarly, ***delegate constructors*** call others from the same class to reduce repetitive code (from C++11):

```
class MyClass {
public:
    MyClass(int a1, bool b1) : a(a1), b(b1) {
        // lots of initialization work
    }
    MyClass(int a1) : MyClass(a1, true) {}
    MyClass(bool b1) : MyClass(10, b1) {}

private:
    int a;
    bool b;
    // ...
};
```


8.3. (Delegate) constructors and destructors

Example 00 (difficulty level: 🌶️)

```
#include <iostream>
class Book { // a Book object always has a title and a price.
    // change this class so that the title cannot be changed, and the price can be changed by Magazine:
    Book(std::string name, double val) : title(name), price(val) {}
    void show() { std::cout << title << " - " << price << "\n"; }
    std::string title;
    double price;
};

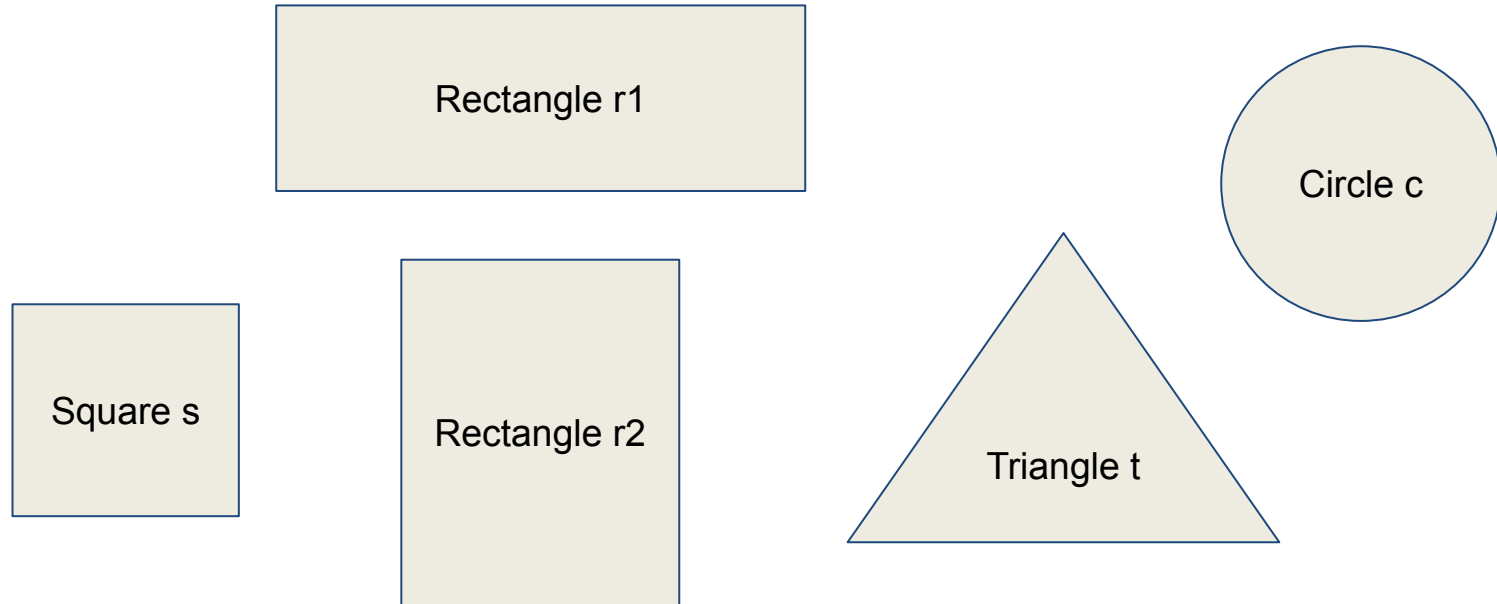
class Magazine { // a Magazine object uses the Book's constructor, and can apply a discount
    // change this class so that an object is created through Book's constructor
    void discount(double percent);
};
// implement Magazine's discount method here

int main() {
    Magazine mag(std::string("C++ Monthly"), 10.0);
    mag.show(); mag.discount(25.0); // this should show 10 in the console, then we apply a 25 % discount
    mag.show(); // this should now show 7.5
    return 0;
}
```

8.3. (Delegate) constructors and destructors

Example (difficulty level: 🌶️🌶️)

Creating 2D graphics elements such as the ones below as classes' objects



8.3. (Delegate) constructors and destructors Example

```
class Element { // class representing graphic element
public:
    Element(double x, double y) : x(x), y(y) {}
private:
    double x, y; // position of graphic element
};
```

```
class Rectangle : public Element { // class representing a rectangle
public:
    Rectangle(double x, double y, double a, double b) : Element(x, y), a(a), b(b) {}
private:
    double a, b; // width and height of rectangle
};
```

```
class Square : public Rectangle { // class representing a square
public:
    Square(double x, double y, double a) : Rectangle(x, y, a, a) {}
};
```



Homework: Add a method to Square that prints out its location: What needs changing?

8.4. mutable, using, friend, and delete

Any **mutable** data members of **const** class instances can be modified. This is useful if most of the class' members should be constant, but a few need to be modified.

```
#include <iostream>

class A { // class A
public:
    int x = 4; // public attributes, default initialized (since C++11)
    mutable int y = 3;
};

int main() {
    const A a; // constant object of class A
    a.y = 7; // this works, a.x would result in compile error
    std::cout << a.y << "\n";
    return 0;
}
```

8.4. mutable, using, friend, and delete

The **using** keyword can be used to change the inheritance properties of class attributes or methods:

```
class A { // class A
    protected:
        int x = 4; // protected attribute
};

class B : public A { // class A
    public:
        using A::x; // inherits x and exposes it as a public attribute
};

int main() {
    B b;
    b.x = 7; // this works, b.x is public (even though A.x is protected)
    return 0;
}
```

8.4. mutable, using, friend, and delete

The **friend** keyword allows a class to access the private and protected attributes and methods of the class that is declared as a friend.

A **friend** relation is **not**:

symmetric: Class A as a friend of B does not imply class B being a friend of A

transitive: A is a friend of B and B is a friend of C does not imply A is a friend of C

inherited: Class Base as a friend of class X does not imply subclass Derived is a friend of class X; Class X as a friend of class Base does not imply class X is a friend of subclass Derived

8.4. mutable, using, friend, and delete

```
#include <iostream>

class A { // class A declares that B is a friend
private:
    friend class B;
    int x = 4; // private attribute
};

class B { // class B is a friend of A
public:
    B() { A a; y = a.x;} // and thus can access
    int f(A a) { return a.x; } // A's private attribute x
private:
    int y;
};

int main() {
    A a; B b;
    std::cout << b.f(a) << "\n";
    return 0;
}
```

8.4. mutable, using, friend, and delete

A **friend *method*** can access the private and protected members of a class if it is declared a friend of that class.

```
#include <iostream>

class A { // class A declares funct as a friend method
private:
    int x = 4; // private attribute
    friend int funct(A a);
};

int funct(A a) { return a.x; } // funct is not a class method

int main() {
    A a;
    std::cout << funct(a) << "\n";
    return 0;
}
```


8.4. mutable, using, friend, and delete

Example 01 (difficulty level: 🌶️)

```
#include <iostream>

class Rectangle { // class Rectangle has width and height as attributes and area() as a method
public:
    Rectangle() {} // default constructor, allows to define width and height later
    Rectangle(int x, int y): width(x), height(y) {} // constructor that sets attributes
    int area() { return width*height; };
    // declare the friend method "enlarge()" here, with a rectangle as parameter, returning a rectangle
private:
    int width, height; // width and height are private, so not accessible from outside the class
};

/* define the friend method here, so that it creates and returns a copy of the rectangle that
   has twice the width and height. The friend method has access to the private attributes. */

int main() {
    Rectangle rectangle1, rectangle2(3,4); // rectangle1 will obtain twice the width and height
    rectangle1 = enlarge(rectangle2); // of rectangle2 through the enlarge method
    std::cout << rectangle1.area() << "\n"; // this should return "48" (6 times 8)
    return 0;
}
```

8.4. mutable, using, friend, and delete

The **delete** keyword marks a class method as deleted. Calling that method (explicitly or implicitly) will result in a compiler error. The **delete** keyword prevents implicitly generating default copy constructors or assignments.

```
class Element { // class representing graphic element
public:
    Element(double x, double y) : x(x), y(y) {} // default constructor is deleted
    Element(Element const & e) = delete; // copying an object gives a compiler error
    void setX(double x) { this->x = x;}
    void setX(float x) = delete; // calling with a float results in a compiler error
private:
    double x, y; // position of graphic element
};

int main() {
    Element e(2.0, 3.0);
    e.setX(5.0); // works, but: e.setX(5.0f) would fail (due to delete above)
}
```

8.4. mutable, using, friend, and delete: Maze Game v.5.00

Class Player and Maze were changed, so we can load a map and add player 2:

```
/* Fifth draft of Maze Game: we now use class "Player" and use a file */
#include "Maze.h" // class that holds everything related to the maze
#include "Player.h" // class that holds everything related to the player
int main() {
    auto c = ' '; // used for user key input
    std::string mapFile("maze.csv");
    Maze maze(mapFile); // initialize a maze object from this file
    Player player1(10, 5);
    Player player2(20, 7);
    while ( c != 'q' ) { // as long as the user doesn't press q ..
        maze.draw(); // draw maze
        player1.draw('@', 3); // draw player 1 as a '@' with color pair 3
        player2.draw('X', 3); // draw player 2 as a 'X' with color pair 3
        c = getch(); // capture the user's pressed key
        switch (c) {
            // ...
        }
    }
}
```

mazeGame.cpp

8.4. mutable, using, friend, and delete: Maze Game v.5.00

Modify the class Player and add a child class Enemy, so that moving the players is now part of the draw class, and the enemy can move toward the player:

```
/* Fifth draft of Maze Game: we now use class "Player" and use a file */
#include "Maze.h" // class that holds everything related to the maze
#include "Player.h" // class that holds everything related to the player
int main() {
    auto c = ' '; // used for user key input
    std::string mapFile("maze.csv");
    Maze maze(mapFile); // initialize a maze object from this file
    Player player(10, 5, 'w', 's', 'a', 'd'); // player at (10,5) and moves with wasd
    EnemyPlayer enemy(20, 7); // enemy player starts at (20,7)
    while ( c != 'q' ) { // as long as the user doesn't press q ..
        maze.draw(); // draw maze
        player.draw('@', 3, c); // draw player 1 as a '@' with color pair 3
        enemy.draw('X', 3, player); // draw enemy player as a 'X' with color pair 3
        c = getch(); // capture the user's pressed key
    }
}
```

mazeGame.cpp

8.5. Polymorphism

C++ provides a way to create a base object with methods that through overriding change their behavior. At run-time, objects of the base class behave as objects of a derived class

If **Sub** is a subclass of **Super**, then the following assignments are allowed:

```
Sub sub;  
Super * superPtr = &sub;  
Super & superRef = sub;
```

A base class pointer or reference can also point or refer to a subclass' object. The other way round is not allowed.

8.5. Polymorphism

For example: **Dog** and **Fish** are classes that inherit from **Animal**. We want any objects from these classes to have a **print()** method, which displays the values of the classes' attributes. Then an **Animal** object pointer could be used to flexibly point to a **Dog** or **Fish** object and access the right **print()** method:

```
Animal * animal;  
animal = new Dog("Scooby");  
animal->print(); // prints out: I am Scooby. Bark!  
animal = new Fish("Salmon");  
animal->print(); // prints out: I'm Salmon (fish)
```

For polymorphism to work in C++, the base class must declare the methods in question as being virtual

8.5. Polymorphism: Example (1/2)

polyDemo.cpp

```
#include <iostream>
#include <cstdlib>

class Animal { // Animal class stores the species and prints this in print()
protected:
    std::string _species;
public:
    Animal(std::string species) { _species = species; }
    virtual void print() { std::cout << "I'm " + _species << std::endl; }
};

class Dog : public Animal { // Dogs inherit species from Animal and have a name
protected:
    std::string _name;
public:
    Dog(std::string name) : Animal("dog"), _name(name) {}
    void print() { std::cout << "I am " << _name << ". Bark!" << std::endl; }
};
```

8.5. Polymorphism: Example (2/2)

```
class Fish : public Animal { // Fishes have species and subspecies
protected:
    std::string _subspecies;
public:
    Fish(std::string subspecies) : Animal("fish"), _subspecies(subspecies) {}
    void print() { std::cout << "I'm " << _subspecies << " (fish)" << std::endl; }
};

int main() {
    Animal * animals[4];
    animals[0] = new Dog("Snowy"); // animals[] has pointers
    animals[1] = new Fish("Salmon"); // to objects of different
    animals[2] = new Dog("Scooby"); // subclasses (Dog, Fish, etc.)
    animals[3] = new Animal("some animal"); // or the animal base class
    for (int i=0; i<15; i++) {
        Animal * a = animals[rand() % 4]; // a is a polymorph variable: its
        a->print(); // print's behavior depends on the
    } // object that a points to
    return 0;
}
```

polyDemo.cpp

8.5. Polymorphism: Example

- Note that animals is an array of pointers to the base class (Animal)
- Yet, we can let the pointers point to a subclass of Animal (Dog, Fish, ...)
- And when we call print() from Animal pointer a, the right method executes

```
Animal * animals[4];  
animals[0] = new Dog("Snowy");  
animals[1] = new Fish("Salmon");  
...  
Animal * a = animals[rand()%4];  
a->print();
```

```
~\> g++ polyDemo.cpp -o polymorph_demo  
~\> ./polymorph_demo  
I'm Salmon (fish)  
I am Scooby. Bark!  
I'm Salmon (fish)  
I am Scooby. Bark!  
I am Snowy. Bark!  
I'm some animal  
I am Scooby. Bark!  
I'm Salmon (fish)  
I am Snowy. Bark!  
I am Scooby. Bark!  
I am Snowy. Bark!  
I am Scooby. Bark!  
I'm some animal  
I am Scooby. Bark!  
I'm Salmon (fish)  
~\>
```

8.5. Polymorphism: **virtual** and late binding

Connecting the function call to the function body is called ***Binding***

Early / static / compile-time binding is done by the compiler through object type, letting the program jump to the function's address

Late / dynamic / run-time binding uses the object type while the executable is running and then matches the function call with the correct function definition: The program has to read the address held in the pointer and then jump to that address. This extra jump makes it less efficient.

C++ achieves late binding by declaring a virtual function

8.5. Polymorphism: `virtual` and late binding

C++ achieves late binding by declaring a virtual function in the base class:

```
#include <iostream>
class A { // class A has a virtual function funct:
public:
    virtual void funct() { std::cout << "A \n"; };
};
class B : public A { // class B inherits from A and overrides funct:
public:
    virtual void funct() { std::cout << "B \n"; }; // virtual here is optional
};

void g( A & a ) { a.funct(); } // g accepts A and B as parameter

int main() {
    A a; B b;
    g(a); g(b); // outputs first "A", and then "B"
    return 0;
}
```

8.5. Polymorphism: **virtual** and **override**

The **override** keyword (from C++11) ensures that the method is virtual and is overriding a virtual function from a base class:

```
#include <iostream>
class A { // class A has a virtual function funct:
public:
    virtual void funct() { std::cout << "A \n"; };
};
class B : public A { // class B inherits from A and overrides funct:
public:
    void funct() override { std::cout << "B \n"; };
};

void g( A & a ) { a.funct(); } // g accepts A and B as parameter

int main() {
    B b; g(b); // outputs "B"
    return 0;
}
```

8.5. Polymorphism: **final**

The **final** keyword (from C++11) prevents inheriting from classes or overriding methods in derived classes

```
class A final { // class A cannot be extended
    // ...
};
class B : public A { // leads to compile error: A is "final"
    // ...
};

class C { // class C contains a virtual final method:
    public:
        virtual void funct() final;
};
class D : public C {
    public:
        void funct(); // leads to compile error: method funct is "final"
};
```