

6 Inheritance

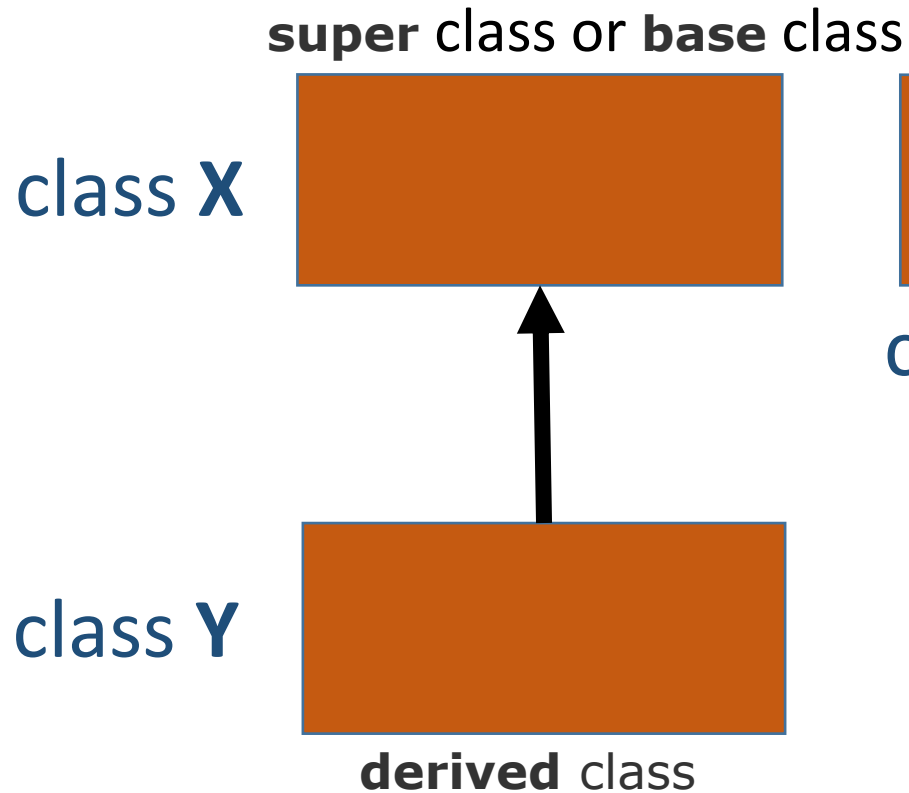
Objectifs

- [6.1 Defining class hierarchy](#)
- [6.2 Classes, inheritance and type compatibility](#)
- [6.3 Polymorphism and virtual methods](#)
- [6.4 Objects as parameters and dynamic casting](#)
- [6.5 Various supplements](#) (**copying constructor**)
- [6.6 The const keyword](#)
- [6.7 Friendship in the “C++” world](#)

6.1 Defining class hierarchy

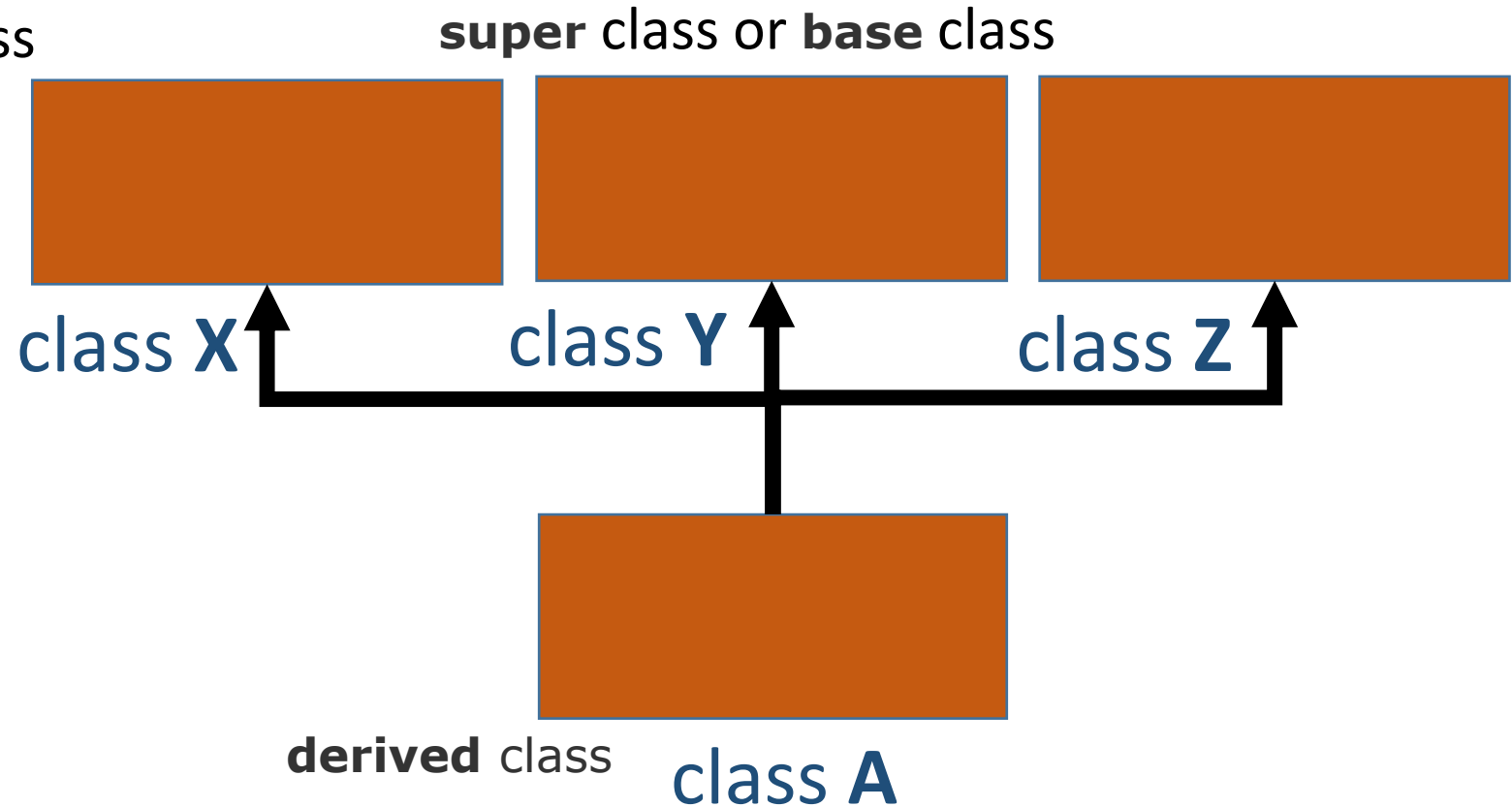
6.1.1 Defining a simple subclass

single inheritance



```
class Y: {visibility specifier} X {...} ;
```

multiple inheritance or multi-inheritance



```
class A : X, Y, Z { ... } ;
```

When we **omit the visibility specifier**, the compiler assumes that we're going to apply a "**private inheritance**".

6.1.4 Defining a simple subclass (4)

compilation errors saying that the *put* and *get* methods are inaccessible. Why?

```
#include <iostream>
using namespace std;
class Super {
private:
    int storage;
public:
    void put(int val) { storage = val; }
    int get(void) { return storage; }
};
class Sub : Super {
    private inheritance
};
```


```
int main(void) {
    Sub object;

    object.put(100);
    object.put(object.get() + 1);
    cout << object.get() << endl;
    return 0;
}
```

When we **omit the visibility specifier**, the compiler assumes that we're going to apply a **"private inheritance"**. This means that **all public superclass components turn into private access, and private superclass components won't be accessible** at all

6.1.5 Defining a simple subclass (5)

```
#include <iostream>
using namespace std;
class Super {
private:    int storage;
public:    void put(int val) { storage = val; }
           int get(void) { return storage; }
};
class Sub : public Super {
};
int main(void) {
    Sub object;
    object.put(100);
    object.put(object.get() + 1);
    cout << object.get() << endl;
    return 0;
}
```



We have to tell the compiler that **we want to preserve** the previously used access policy. We do this by using a “public” visibility specifier:

```
class Sub : public Super { };
```

Private components will remain **private**,
public components will remain **public**

the class has lost access to the
private components of the superclass.

6.1.6 Defining a simple subclass (6)

```
#include <iostream>
using namespace std;
class Super {
protected:    int storage;
public:         void put(int val) { storage = val; }
               int get(void) { return storage; }
};
class Sub : public Super {
public:
    void print(void) { cout << "storage = " << storage << endl; }
};
int main(void) {
    Sub object;
    object.put(100);
    object.put(object.get() + 1);
    object.print();
    return 0;
}
object.storage = 0; //the variable remains hidden anyway
```

the class has lost access to the **private components of the superclass**. . We cannot write a member function of the *Sub* class which would be able to directly manipulate the *storage* variable (private). This is a very serious restriction.

This wouldn't be possible if the variable was declared as **private**.

The keyword ***protected*** means that any component marked with it **behaves like a public component when used by any of the subclasses and looks like a private component to the rest of the world.**

storage = 101

01

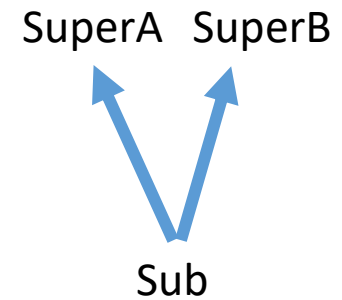
6.1.8 Defining a simple subclass (8)

simple example demonstrating **multi-inheritance**

```
#include <iostream>
using namespace std;
class SuperA {
protected:    int storage;
public:        void put(int val) { storage = val; }
               int get(void) { return storage; }
};
class SuperB {
protected:    int safe;
public:        void insert(int val) { safe = val; }
               int takeout(void) { return safe; }
};
class Sub : public SuperA, public SuperB {
public:        void print(void) {
               cout << "storage = " << storage << endl;
               cout << "safe   = " << safe << endl;
               }
};
```

```
int main(void) {
    Sub object;

    object.put(1);
    object.insert(2);
    object.put(object.get() + object.takeout());
    object.insert(object.get() + object.takeout());
    object.print();
    return 0;
}
```



```
storage = 3
safe    = 5
```


6.2.1 Type compatibility – the simplest case

```
#include <iostream>
using namespace std;
```

```
class Cat {
public:
```

Each new class constitutes a new type of data.

Each object constructed on the basis of such a class is like **a value of the new type.**

```
    void MakeSound(void) { cout << "Meow! Meow!" << endl; }
```

```
};
```

```
class Dog {
public:
```

```
    void MakeSound(void) { cout << "Woof! Woof!" << endl; }
```

```
};
```

```
int main(void) {
```

```
    Cat *a_cat = new Cat();
```

```
    Dog *a_dog = new Dog();
```

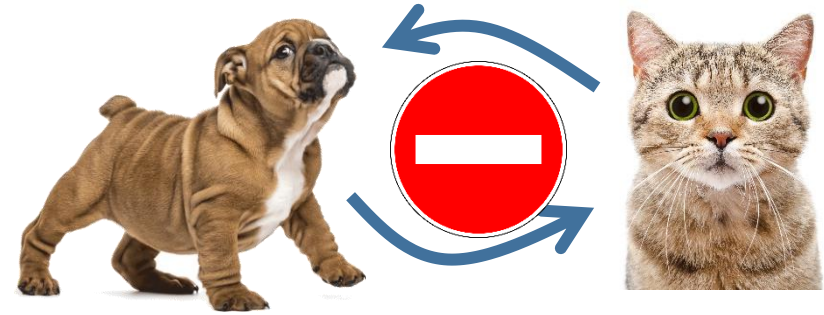
```
    a_cat -> MakeSound();
```

```
    a_dog -> MakeSound();
```

```
    return 0;
```

```
}
```

```
Meow! Meow!
Woof! Woof!
```



This means that **any two objects may (or may not) be compatible** in the sense of their types.



```
a_dog = a_cat;
a_cat = a_dog;
```

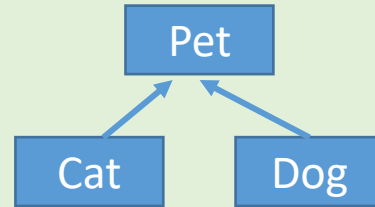
a compiler error

So we can say that **objects derived from classes which lie in different branches of the inheritance tree are always incompatible**

6.2.2 Type compatibility – more complex case (1/2)

```
#include <iostream>
#include <string>
using namespace std;
```

```
class Pet {
protected:
    string Name;
public:
    Pet(string n) { Name = n; }
    void Run(void) { cout << Name << ": I'm running" << endl; }
};
```



```
class Dog : public Pet {
public:
    Dog(string n) : Pet(n) {};
    void MakeSound(void) { cout << Name << ": Woof! Woof!" << endl; }
};
```

```
class Cat : public Pet {
public:
    Cat(string n) : Pet(n) {};
    void MakeSound(void) { cout << Name << ": Meow! Meow!" << endl; }
};
```

```
int main(void) {
    Pet a_pet("pet");
    Cat a_cat("Tom");
    Dog a_dog("Spike");
    a_pet.Run();
    a_dog.Run();
    a_dog.MakeSound();
    a_cat.Run();
    a_cat.MakeSound();
    return 0;
}
```

```
pet: I'm running
Spike: I'm running
Spike: woof! woof!
Tom: I'm running
Tom: Meow! Meow!
```

6.2.2 Type compatibility – more complex case (2/2)

objects of the superclass are compatible with objects of the subclass

objects of the subclass are not compatible with objects of the superclass

This means that:

you **can** do the following:

```
a_pet = new Dog("Huckleberry");
```

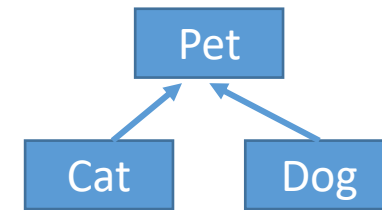
```
a_pet -> Run();
```

but you **cannot** do anything like this:

```
a_pet -> MakeSound();
```

because *Pets* don't know how to make sounds, you are **not allowed** to do the following:

```
a_dog = new Pet("Strange pet");
```



```
Tom: I'm running
Spike: I'm running
```

6.2.3 Type compatibility – more complex case (2)

```
#include <iostream>
#include <string>
using namespace std;

class Pet {
protected:    string Name;
public:
    Pet(string n) { Name = n; }
    void Run(void) {
        cout << Name << ": I'm running" << endl; }
};

class Dog : public Pet {
public:
    Dog(string n) : Pet(n) {};
    void MakeSound(void) {
        cout << Name << ": Woof! Woof!" << endl; }
};

class Cat : public Pet {
public:
    Cat(string n) : Pet(n) {};
    void MakeSound(void) {
        cout << Name << ": Meow! Meow!" << endl; }
};
```

```
int main(void) {
    Pet *a_pet1 = new Cat("Tom");
    Pet *a_pet2 = new Dog("Spike");

    a_pet1 -> Run();
    // 'a_pet1 -> MakeSound();' is not allowed here!
    a_pet2 -> Run();
    // 'a_pet2 -> MakeSound();' is not allowed here!

    return 0;
}
```

The problem comes from **static checks made by the compiler during the compilation process.**

Peut on convaincre le compilateur a changer d'avis?
Voir slide suivant

Cat and *Dog* objects can do all the things *Pets* are able to do
Pets **cannot** do all the thing that *Cat* and *Dog* can do

6.2.4 Type compatibility – how to recover the lost

Peut on convaincre le compilateur a changer d'avis?

```
static_cast<target_type>(an_expression)
```

We use the **cast operator** when we want to affect the compiler and express a message like this:

I want to use the pointer to the superclass in relation to the object of the subclass; I guarantee that the proper object exists.

The **target_type** is a type name (or a type description) which we want the compiler to use when evaluating the value of *an_expression*.

For example: the following form → `static_cast<Dog *>(a_pet)`

forces the compiler to assume that *a_pet* is (temporarily) converted into a pointer of type *Dog **.

`z = double(n) ; // conversion de int en double` ou : `z = static_cast<double> (n)`

C++ Cast Operators (New Style)

(complement masque)

CAST SYNTAX

static_cast< type>(expression)

reinterpret_cast<type>(expression)

dynamic_cast<type>(expression)

const_cast< type>(expression)

DESCRIPTION

Recasts expression into the data format of type, such as casting **double** to **int** (and removing warning messages), or to cast to or from an enum type. Essentially, static_cast says, “**Yes, I really do want to do this.**” For the cast to work, some sort of conversion must be possible between the types involved.

Recasts one pointer type to another or casts between a pointer type and int, or vice versa. This cast is potentially dangerous (so make sure you need it before using it), because it changes how data at a particular address is to be interpreted.

Casts a **base-class pointer** to a **subclass pointer** **after verifying** that the object pointed to has the specified subclass type. Produces **NULL if cast is not valid**. **Requires** that the classes involved **have one or more virtual functions**. This is casting **downward** through an inheritance hierarchy; going the other way (assigning subclass pointer to a base-class pointer) is freely permitted and requires no cast.

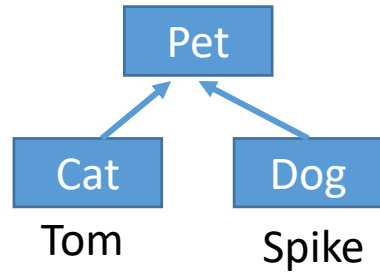
Casts a non-const expression to a const type. It is your responsibility to make sure the expression is not one that will be changed.

6.2.5 Type compatibility – back to our pets

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected: string Name;
public:
    Pet(string n) { Name = n; }
    void Run(void) {
        cout << Name << ": I'm running" << endl; }
};
```

```
class Dog : public Pet {
public:
    Dog(string n) : Pet(n) {};
    void MakeSound(void) {
        cout << Name << ": Woof! Woof!" << endl; }
};
```

```
class Cat : public Pet {
public:
    Cat(string n) : Pet(n) {};
    void MakeSound(void) {
        cout << Name << ": Meow! Meow!" << endl; }
};
```



```
int main(void) {
    Pet *a_pet1 = new Cat("Tom");
    Pet *a_pet2 = new Dog("Spike");
    a_pet1 -> Run();
    static_cast<Cat *>(a_pet1) -> MakeSound();
    a_pet2 -> Run();
    static_cast<Dog *>(a_pet2) -> MakeSound();
    return 0;
}
```

```
Tom: I'm running
Tom: Meow! Meow!
Spike: I'm running
Spike: woof! woof!
```

6.2.6 Type compatibility – abusing owner's power

Unfortunately, it's easy to misuse and abuse the power of the *static_cast* operator. If you use it thoughtlessly and without adequate care, you can get yourself into trouble.

```
int main(void) {  
    Pet *a_pet1 = new Cat("Tom");  
    Pet *a_pet2 = new Dog("Spike");  
    a_pet2 -> Run();  
    static_cast<Cat *>(a_pet2) -> MakeSound();  
    a_pet1 -> Run();  
    static_cast<Dog *>(a_pet1) -> MakeSound();  
    return 0;  
}
```

The compiler is forced to recognize the pointer as valid – it has no other choice, actually.

This effect is caused by the fact that the compiler **isn't able to check if the pointer being converted is compatible with the object it points to.**

```
Spike: I'm running  
Spike: Meow! Meow!  
Tom: I'm running  
Tom: Woof! Woof!
```

We've tried to treat a cat like a dog and *vice versa*. The results are disastrous


6.2.6 Type compatibility – abusing owner's power

Ce n'est qu'**au moment de l'exécution** qu'on saura si la conversion est réalisable ou non.

Full pointer validity verification is possible when and only when the program is being executed (in other words, during runtime).

The “C++” language has a second conversion operator.

Its name is : ***dynamic_cast***.

the conversion is carried out dynamically regarding the current state of all created objects. This means that the conversion may (or  may not) be successful, causing our program to stop if it wants any dog to meow.

D'une manière générale, l'opérateur ***dynamic_cast*** aboutit si l'objet réellement pointé est, par rapport au type d'arrivée demandé, d'un type identique ou d'un type descendant

6.2.7 Type compatibility – final case

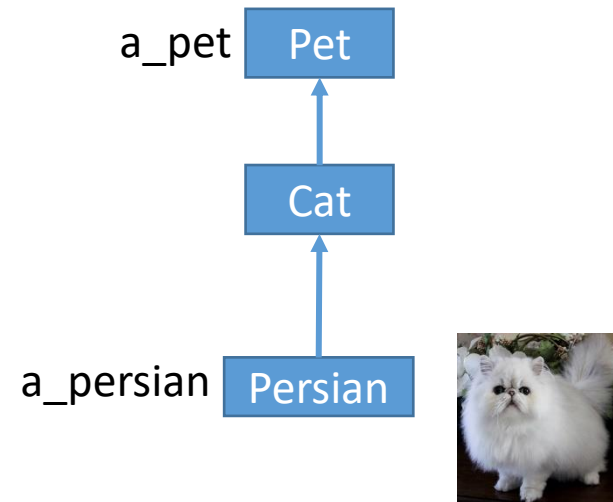
```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected:    string Name;
public:    Pet(string n) { Name = n; }
    void Run(void) {
        cout << Name << ": I'm running" << endl; }
};

class Cat : public Pet {
public:    Cat(string n) : Pet(n) {};
    void MakeSound(void) {
        cout << Name << ": Meow! Meow!" << endl; }
};

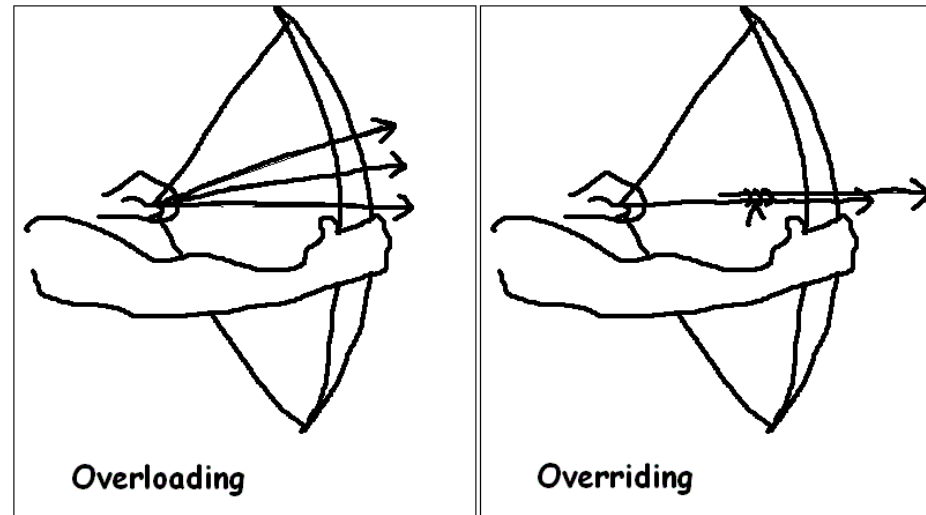
class Persian : public Cat {
public:    Persian(string n) : Cat(n) {};
};
```

```
Mr. Bigglesworth: Meow! Meow!
Mr. Bigglesworth: Meow! Meow!
```

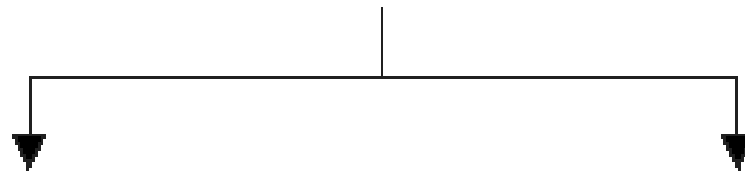
```
int main(void) {
    Pet  *a_pet;
    Persian *a_persian;
    a_pet = a_persian = new Persian("Mr. Bigglesworth");
    a_persian -> MakeSound();
    static_cast<Persian *>(a_pet) -> MakeSound();
    return 0;
}
```



6.3 Polymorphism and virtual methods



Types of Polymorphism



Compile-time / Static poly. / Early binding

Ex: Function Overloading
Operator Overloading

Run-time / Dynamic poly. / Late binding

Ex: Virtual function

```
#include <iostream>
using namespace std;
class Pet {
```

```
    protected:        string Name;
public: Pet(string n) { Name = n; }
        void MakeSound(void) { cout << Name
<< " the Pet says: Shh! Shh!" << endl; }
};
```

```
class Cat : public Pet {
public: Cat(string n) : Pet(n) { }
        void MakeSound(void) { cout << Name
<< " the Cat says: Meow! Meow!" << endl; }
};
```

```
class Dog : public Pet {
public:
    Dog(string n) : Pet(n) { }
    void MakeSound(void) { cout << Name
<< " the Dog says: Woof! Woof!" << endl; }
};
```

6.3.1 Overriding a method in the subclass (1)

Overriding a method in the subclass – the remainder

When a subclass declares a method of the name previously known in its superclass, the original method **is overridden**. the subclass **hides** the previous meaning of the method identifier

```
int main(void) {
    Cat *a_cat;
    Dog *a_dog;
```

```
Kitty the Cat says: Meow! Meow!
Kitty the Pet says: Shh! Shh!
Doggie the Dog says: Woof! Woof!
Doggie the Pet says: Shh! Shh!
```

```
    a_cat = new Cat("Kitty");
    a_dog = new Dog("Doggie");
    a_cat -> MakeSound();
    static_cast<Pet *>(a_cat) -> MakeSound();
    a_dog -> MakeSound();
    static_cast<Pet *>(a_dog) -> MakeSound();
    return 0;
```

the effects of the overriding may be reversed (or voided) if you use the **static_cast** operator in reverse.

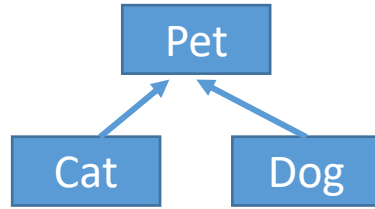
```
}
```

6.3.2 Overriding a method in the subclass (2)

```
#include <iostream>
using namespace std;
class Pet {
protected:    string Name;
public:    Pet(string n) { Name = n; }
    void MakeSound(void) {
cout << Name << " the Pet says: Shh! Shh!" << endl; }
};

class Cat : public Pet {
public:    Cat(string n) : Pet(n) { }
    void MakeSound(void) {
cout << Name << " the Cat says: Meow! Meow!" << endl; }
};

class Dog : public Pet {
public:
    Dog(string n) : Pet(n) { }
    void MakeSound(void) {
cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
```



```
int main(void) {
```

```
    Pet *a_pet1, *a_pet2;
```

```
    Cat *a_cat;
```

```
    Dog *a_dog;
```

```
Kitty the Pet says: Shh! Shh!
Kitty the Cat says: Meow! Meow!
Doggie the Pet says: Shh! Shh!
Doggie the Dog says: woof! woof!
```

```
    a_pet1 = a_cat = new Cat("Kitty");
```

```
    a_pet2 = a_dog = new Dog("Doggie");
```

```
    a_pet1 -> MakeSound();
```

```
    a_cat -> MakeSound();
```

```
    a_pet2 -> MakeSound();
```

```
    a_dog -> MakeSound();
```

```
    return 0;
```

```
}
```

we don't use the *static_cast*, but the some of the pointers are explicitly declared as pointing to the common superclass (*Pet*). The classes themselves remain untouched. We've merely changed the way in which we treat the pointers.

```
#include <iostream>
using namespace std;
class Pet {
```

polymorphism is the ability to realize class behaviour in multiple ways.

6.3.3 Overriding a method in the subclass (3)

The word "**polymorphism**" means that the one and same class may show many ("**poly**" – like in "*polygamy*") forms ("*morphs*") **not** defined **by the class** itself, but **by its subclasses**.

```
protected:    string Name;
public:   Pet(string n) { Name = n; }
    virtual void MakeSound(void) {
cout << Name << " the Pet says: Shh! Shh!" << endl; }
};
class Cat : public Pet {
public:   Cat(string n) : Pet(n) { }
    void MakeSound(void) {
cout << Name << " the Cat says: Meow! Meow!" << endl; }
};
class Dog : public Pet {
public:   Dog(string n) : Pet(n) { }
    void MakeSound(void) {
cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
```

```
Kitty the Cat says: Meow! Meow!
Kitty the Cat says: Meow! Meow!
Kitty the Cat says: Meow! Meow!
Doggie the Dog says: Woof! Woof!
Doggie the Dog says: Woof! Woof!
Doggie the Dog says: Woof! Woof!
```

```
int main(void) {
    Pet *a_pet1, *a_pet2;
    Cat *a_cat;
    Dog *a_dog;
    a_pet1 = a_cat = new Cat("Kitty");
    a_pet2 = a_dog = new Dog("Doggie");
    a_pet1 -> MakeSound();
    a_cat -> MakeSound();
    static_cast<Pet *>(a_cat) -> MakeSound();
    a_pet2 -> MakeSound();
    a_dog -> MakeSound();
    static_cast<Pet *>(a_dog) -> MakeSound();
    return 0;
}
```

6.3.4 Overriding a method in the subclass (4)

the binding between the origin of the virtual function (inside the superclass) and its replacement (defined within the subclass) is created dynamically, during the execution of the program.

```
#include <iostream>
using namespace std;
class Pet {
protected:    string Name;
public:    Pet(string n) { Name = n; MakeSound(); }
    virtual void MakeSound(void) {
        cout << Name << " the Pet says: Shh! Shh!" << endl; }
};
```

```
class Cat : public Pet {
public:    Cat(string n) : Pet(n) { }
        void MakeSound(void) {
            cout << Name << " the Cat says: Meow! Meow!" << endl; }
};

class Dog : public Pet {
public:    Dog(string n) : Pet(n) { }
        void MakeSound(void) {
            cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
```

Kitty the Pet says: Shh! Shh!
Doggie the Pet says: Shh! Shh!


```
int main(void) {
    Cat *a_cat;
    Dog *a_dog;

    a_cat = new Cat("Kitty");
    a_dog = new Dog("Doggie");

    return 0;
}
```

6.3.5 Overriding a method in the subclass (5)

```
#include <iostream>
using namespace std;
class Pet {
protected:
    string Name;
public: Pet(string n) { Name = n; }
    virtual void MakeSound(void) {
        cout << Name << " the Pet says: Shh! Shh!" << endl; }
    void WakeUp(void) { MakeSound(); }
};
class Cat : public Pet {
public: Cat(string n) : Pet(n) { }
    void MakeSound(void) {
        cout << Name << " the Cat says: Meow! Meow!" << endl; }
};
class Dog : public Pet {
public: Dog(string n) : Pet(n) { }
    void MakeSound(void) {
        cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
```



the *MakeSound* method is invoked indirectly, via the *WakeUp* method which is defined once in the superclass and isn't overridden anywhere else.

```
int main(void) {
    Cat *a_cat;
    Dog *a_dog;

    a_cat = new Cat("Kitty");
    a_cat -> WakeUp();
    a_dog = new Dog("Doggie");
    a_dog -> WakeUp();

    return 0;
}
```

```
Kitty the Cat says: Meow! Meow!
Doggie the Dog says: woof! woof!
```


6.4 Objects as parameters and dynamic casting

6.4.1 Passing an object as a function parameter

```
#include <iostream>
#include <string>
using namespace std;
```

```
class Pet {
protected:    string name;
public: void NameMe(string name) {
    this -> name = name; }
    void MakeSound(void) {
    cout << name << " says: no comments"
    << endl; }
};
```

```
void PlayWithPetByPointer(string name, Pet *pet) {
    pet -> NameMe(name);
    pet -> MakeSound();
}
```

by pointer

```
void PlayWithPetByReference(string name, Pet &pet) {
    pet.NameMe(name);
    pet.MakeSound();
}
```

by reference

```
int main(void) {
    Pet *p1 = new Pet;
    Pet p2;
    PlayWithPetByPointer("anonymous", p1);
    PlayWithPetByReference("no_name", p2);
    PlayWithPetByPointer("no_name", &p2);
    PlayWithPetByReference("anonymous", *p1);
    return 0;
}
```

p2 is a subject to the & operator

p1 is dereferenced by the * operator

```
anonymous says: no comments
no_name says: no comments
no_name says: no comments
anonymous says: no comments
```

Any object may be used as a function parameter and, vice versa, **any function may have a parameter as an object of any class.**

6.4.2 Passing an object **by value**

```
#include<iostream>
```

```
#include <string>
```

```
using namespace std;
```

```
class Pet {
```

```
protected: string name;
```

```
public: void NameMe(string name) { this -> name = name; }
```

```
void MakeSound(void) {
```

```
cout << name << " says: no comments" << endl; }
```

```
};
```

```
void NamePetByValue(string name, Pet pet) {
```

```
pet.NameMe(name);
```

```
}
```

```
void NamePetByPointer(string name, Pet *pet) {
```

```
pet -> NameMe(name);
```

```
}
```

```
void NamePetByReference(string name, Pet &pet)
```

```
pet.NameMe(name);
```

```
}
```

```
no_name says: no comments  
Beta says: no comments  
Gamma says: no comments
```

```
int main(void) {
```

```
Pet pet;
```

```
pet.NameMe("no_name");
```

```
NamePetByValue("Alpha", pet);
```

```
pet.MakeSound();
```

```
NamePetByPointer("Beta", &pet);
```

```
pet.MakeSound();
```

```
NamePetByReference("Gamma", pet);
```

```
pet.MakeSound();
```

```
return 0;
```

```
}
```

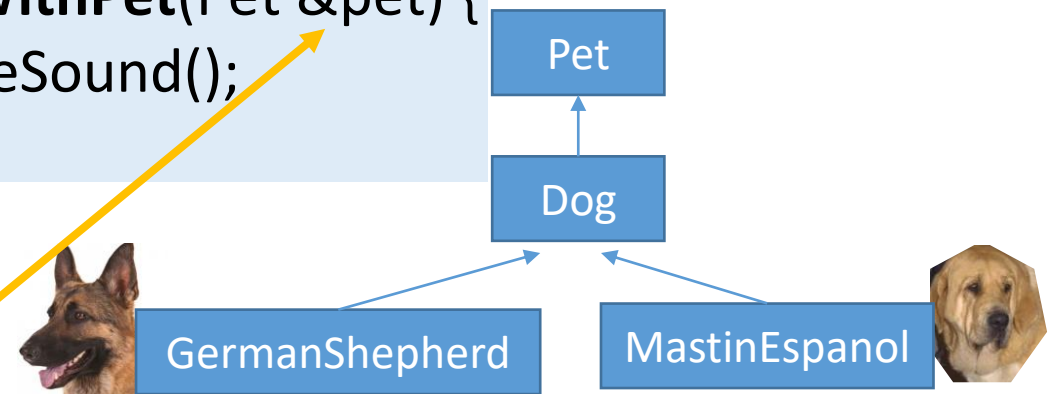
6.4.3 Passing an object of a subclass **by reference** (1)

```
#include<iostream>
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected: string name;
public:   Pet(string name) {
    this -> name = name; }
    void MakeSound(void) {
        cout << name << " is silent :(" << endl; }
};
class Dog : public Pet {
public: Dog(string name) : Pet(name) {}
    void MakeSound(void) {
        cout << name << " says: Woof!" << endl; }
};
```

the *PlayWithPet* function
accepts a reference

```
class GermanShepherd : public Dog {
public: GermanShepherd(string name) :
    Dog(name) {}
    void MakeSound(void) {
        cout << name << " says: Wuff!" << endl; }
};
class MastinEspanol : public Dog {
public: MastinEspanol(string name) : Dog(name) {}
    void MakeSound(void) {
        cout << name << " says: Guau!" << endl; }
};

void PlayWithPet(Pet &pet) {
    pet.MakeSound();
}
```



6.4.3 Passing an object of a subclass (1)

```
int main(void) {  
    Pet pet("creature");  
    Dog dog("Dog");  
    GermanShepherd gs("Hund");  
    MastinEspanol mes("Perro");  
    PlayWithPet(pet);  
    PlayWithPet(dog);  
    PlayWithPet(gs);  
    PlayWithPet(mes);  
    return 0;  
}
```

The function is invoked with four different objects taken from different levels of the class hierarchy. This is possible because an object of the superclass is type compatible with the objects of any of the subclasses.

We can declare a formal parameter of a type as a superclass and pass an actual parameter of any of the formal parameter's subclasses.

```
creature is silent :(  
Dog is silent :(  
Hund is silent :(  
Perro is silent :(  

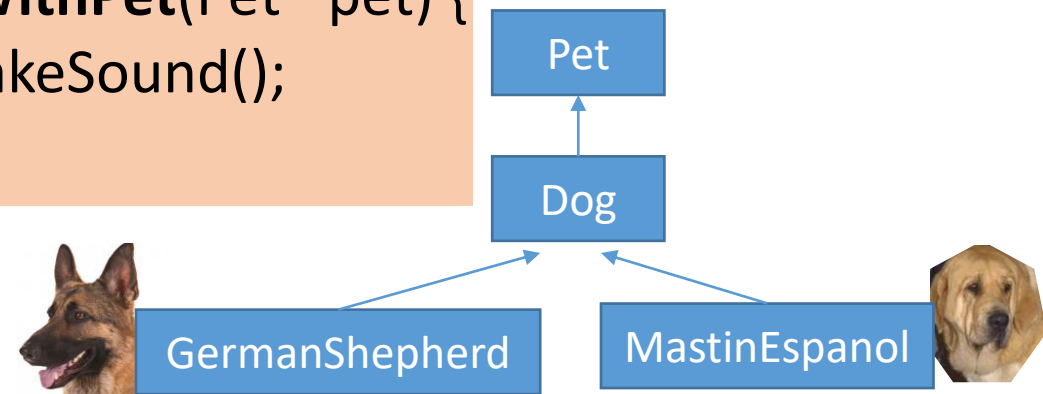
```

6.4.3 Passing an object of a subclass **by pointer** (1)

```
#include<iostream>
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected: string name;
public:   Pet(string name) {
    this -> name = name; }
    void MakeSound(void) {
        cout << name << " is silent :(" << endl; }
};
class Dog : public Pet {
public: Dog(string name) : Pet(name) {}
    void MakeSound(void) {
        cout << name << " says: Woof!" << endl; }
};
```

```
class GermanShepherd : public Dog {
public: GermanShepherd(string name) :
    Dog(name) {}
    void MakeSound(void) {
        cout << name << " says: Wuff!" << endl; }
};
class MastinEspanol : public Dog {
public: MastinEspanol(string name) : Dog(name) {}
    void MakeSound(void) {
        cout << name << " says: Guau!" << endl; }
};

void PlayWithPet(Pet *pet) {
    pet->MakeSound();
}
```



6.4.3 Passing an object of a subclass (1)

```
int main(void) {  
    Pet *pet = new Pet("creature");  
    Dog *dog = new Dog("Dog");  
    GermanShepherd *gs = new GermanShepherd("Hund");  
    MastinEspanol *mes = new MastinEspanol("Perro");  
    PlayWithPet(pet);  
    PlayWithPet(dog);  
    PlayWithPet(gs);  
    PlayWithPet(mes);  
    return 0;  
}
```

We've changed the interface of the *PlayWithPet* function and now the function accepts a pointer, not a reference.

We can declare a formal parameter of a type as a superclass and pass an actual parameter of any of the formal parameter's subclasses.

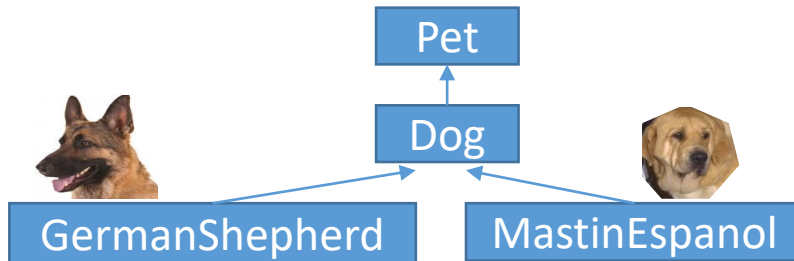
```
creature is silent :(  
Dog is silent :(  
Hund is silent :(  
Perro is silent :(  

```

6.4.5 The dynamic_cast operator (1/2)

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected: string name;
public:   Pet(string name) : name(name) { }
        virtual void MakeSound(void) {
            cout << name << " is silent :(" << endl; }
};

class Dog : public Pet {
public: Dog(string name) : Pet(name) { }
        void MakeSound(void) {
            cout << name << " says: Woof!" << endl; }
};
```



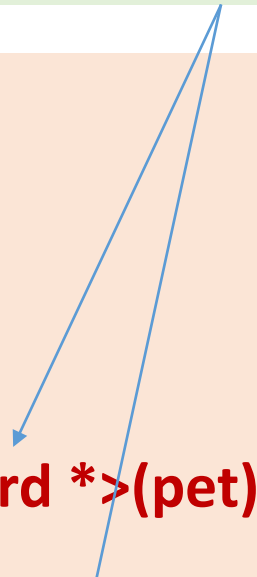
```
class GermanShepherd : public Dog {
public:
    GermanShepherd(string name) : Dog(name) { }
    void MakeSound(void) {
        cout << name << " says: Wuff!" << endl; }
    void Laufen(void) {
        cout << name << " runs (gs)!" << endl; }
};

class MastinEspanol : public Dog {
public: MastinEspanol(string name) :
    Dog(name) { }
        void MakeSound(void) {
            cout << name << " says: Guau!" << endl; }
        void Ejecutar(void) {
            cout << name << " runs (mes)!" << endl; }
};
```


6.4.5 The dynamic_cast operator (2/2)

The **dynamic_cast** operator applied to a pointer

```
void PlayWithPet(Pet *pet) {  
    GermanShepherd *gs;  
    MastinEspanol *mes;  
    pet -> MakeSound();  
    if(gs =  
        dynamic_cast<GermanShepherd *>(pet))  
        gs -> Laufen();  
    if(mes=dynamic_cast<MastinEspanol *>(pet))  
        mes -> Ejecutar();  
}
```



The function does two important things:

- it invokes the *MakeSound* method; the method is marked as virtual so we expect the “national” objects will be able to make their native sounds;
- it tries to recognize the nature of the received pointer and to force the pointed object to behave according to its origin – this is the moment when the **dynamic_cast** operator becomes indispensable

```
int main(void) {  
    Pet *pet = new Pet("creature");  
    Dog *dog = new Dog("Dog");  
    GermanShepherd *gs =  
        new GermanShepherd("Hund");  
    MastinEspanol *mes =  
        new MastinEspanol("Perro");  
    PlayWithPet(pet);  
    PlayWithPet(dog);  
    PlayWithPet(gs);  
    PlayWithPet(mes);  
    return 0;  
}
```

```
creature is silent :(  
Dog says: woof!  
Hund says: wuff!  
Hund runs (gs)!  
Perro says: Guau!  
Perro runs (mes)!
```

6.4.6 The dynamic_cast operator (2)

*The **dynamic_cast** operator applied to a reference*

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected: string name;
public:   Pet(string name) : name(name) {}
        virtual void MakeSound(void) { cout << name << " is silent :(" << endl; }
};
class Dog : public Pet {
public: Dog(string name) : Pet(name) {}
        void MakeSound(void) { cout << name << " says: Woof!" << endl; }
};
class GermanShepherd : public Dog {
public: GermanShepherd(string name) : Dog(name) {}
        void MakeSound(void) { cout << name << " says: Wuff!" << endl; }
        void Laufen(void) { cout << name << " runs (gs)!" << endl; }
};
```

6.4.6 The dynamic_cast operator (2)

```
class MastinEspanol : public Dog {  
public:   MastinEspanol(string name) : Dog(name) {}  
        void MakeSound(void) { cout << name << " says: Guau!" << endl; }  
        void Ejecutar(void) { cout << name << " runs (mes)!" << endl; }  
};
```

```
void PlayWithPet(Pet &pet) {  
    pet.MakeSound();  
    dynamic_cast<GermanShepherd &>(pet).Laufen();  
    dynamic_cast<MastinEspanol &>(pet).Ejecutar();  
}
```

```
int main(void) {  
    Pet pet("creature");  
    Dog dog("Dog");  
    GermanShepherd gs("Hund");  
    MastinEspanol mes("Perro");  
    PlayWithPet(pet);  
    PlayWithPet(dog);  
    PlayWithPet(gs);  
    PlayWithPet(mes);  
    return 0;  
}
```

The *PlayWithPet* function doesn't have a pointer but a **reference**. In consequence, the following two parts of the programs have been changed too

We mustn't use a casted reference (or pointer) without being sure that the result is already defined.

the form of **dynamic_cast** utilization is quite different here; the operator takes the following form:

dynamic_cast<reference_type>(reference_to_object)
and returns a newly transformed (converted) reference which, as a result, may be used like an ordinary **l-value**; we don't need to assign it to a variable if we want to make use of it; this is exactly what we did inside the modified function.

There's a way to protect ourselves from the effects of unsuccessful castings.
the try-catch statement.

```
creature is silent :(  
terminate called after throwing an instance of 'std::bad_cast'  
    what():  std::bad_cast
```


```
This application has requested the Runtime to terminate it in an unusual way.  
Please contact the application's support team for more information.
```

6.4.7 The dynamic_cast operator (3)

the **try-catch statement**. It looks like this:

```
try {  
    thing_we_want_to_try_although_we_are_not_quite_sure_if_it_is_reasonable;  
} catch(...) {}
```

```
void PlayWithPet(Pet &pet) {  
    pet.MakeSound();  
    try {  
        dynamic_cast<GermanShepherd &>(pet).Laufen();  
    } catch(...) {}  
    try {  
        dynamic_cast<MastinEspanol &>(pet).Ejecutar();  
    } catch(...) {}  
}
```



we catch the error and do nothing

creature is silent :(
Dog says: Woof!
Hund says: Wuff!
Hund runs (gs)!
Perro says: Guau!
Perro runs (mes)!

6.5 Various supplements

The **copying constructor**

```
#include <iostream>
```

```
using namespace std;
```

```
class Class {
```

```
    int data;
```

```
public:
```

```
    Class(int value) : data(value) {}
```

```
    void increment(void) { data++; }
```

```
    int value(void) { return data; }
```

```
};
```

```
int main(void) {
```

```
    Class o1(123);
```

```
    Class o2 = o1;
```

```
    Class o3(o2);
```

```
    o1.increment();
```

```
    cout << o1.value() << endl;
```

```
    cout << o2.value() << endl;
```

```
    cout << o3.value() << endl;
```

```
    return 0;
```

```
}
```

6.5.1 More about copying constructors (1)

The **copying constructor** is a specific form of constructor designed to make a more or less literal copy of an object. You can recognize this constructor by its **distinguishable header**.

Assuming that a class is called A, its copying constructor will be declared as:

A(A &)

This means that the constructor expects **one parameter to be a reference to an object** whose content is intended to be copied to the newly created object.

The **implicit constructor** simply clones (bit by bit) the source object, producing a twin copy of it

If there's **no explicit copying** constructor in some class, an implicit constructor will be used instead.

124
123
123

6.5.2 More about copying constructors (2)

```
#include <iostream>
using namespace std;
class Class {
    int *data;
public: Class(int value) {
    data = new int;
    *data = value; }
    void increment(void) { (*data)++; }
    int value(void) { return *data; } };
int main(void) {
    Class o1(123);
    Class o2 = o1;
    Class o3(o2);
    o1.increment();
    cout << o1.value() << endl;
    cout << o2.value() << endl;
    cout << o3.value() << endl;
    return 0;
}
```

now the data isn't stored in a regular variable but in **a piece of memory allocated by the new operator.**

implicit copying constructor may be dangerous and may cause adverse effects.

the implicit copying constructor makes a twin copy of an object. Note that this is of an object, **not** the entities existing **outside** the object. This means that the *data* field will obviously be copied (cloned) into the newly created object, but in effect it will point to the same piece of memory.

This also means that different objects may have something in common – they may share some data among them.

124
124
124

6.5.3 More about copying constructors (3)

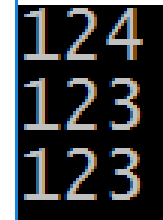
```
include <iostream>
using namespace std;
class Class {    int *data;
public: Class(int value) { data = new int;
                        *data = value;  }

    void increment(void) { (*data)++; }
    int value(void) { return *data; }
};

int main(void) {
    Class o1(123);
    Class o2(o1.value());
    Class o3(o2.value());
    o1.increment();
    cout << o1.value() << endl;
    cout << o2.value() << endl;
    cout << o3.value() << endl;
    return 0;
}
```

we can protect ourselves from this kind of behaviour by just not using the copying constructor. It only uses the explicit constructor, which gets one parameter of type *int*.

it isn't elegant. it isn't safe. We've placed it here for didactical purposes only.



124
123
123

6.5.4 More about copying constructors (4)

Let's assume that we don't want our objects to share any data. We're determined to isolate the objects and keep them all separate. Our new copying constructor has allocated a new piece of memory and copied original data content to it.

```
#include <iostream>
using namespace std;
class Class {
    int *data;
public:
    Class(int value) {
        data = new int;
        *data = value;
    }
    Class(Class &source) {
        data = new int;
        *data = source.value();
    }
    void increment(void) { (*data)++; }
    int value(void) { return *data; }
};
```

```
int main(void) {
    Class o1(123);
    Class o2 = o1;
    Class o3(o2);
    o1.increment();
    cout << o1.value() << endl;
    cout << o2.value() << endl;
    cout << o3.value() << endl;
    return 0;
}
```

124
123
123

6.5.5 More about copying constructors (5)

Using an object as function parameters passed **by value** isn't a good idea. the mechanism of passing parameters by value assumes that a function operates on the copy of an actual parameter. This is clear when we consider parameters of simple types (like *int* or *float*), but it becomes more complex when the parameter is an object.

```
#include <iostream>
using namespace std;
class Dummy {
public:
    Dummy(int value) {}
    Dummy(Dummy &source) {
        cout << "Hi from the copy constructor!" << endl;
    }
};

void DoSomething(Dummy ob) {
    cout << "I'm here!" << endl;
}

int main(void) {
    Dummy o1(123);
    DoSomething(o1);
    return 0;
}
```

the copying constructor will be invoked when an object is passed to a function by value. We've had to make the constructor a little verbose – this is the simplest way to trace its paths.

```
Hi from the copy constructor!
I'm here!
```

6.5.6 More about copying constructors (6)

```
#include <iostream>
using namespace std;
class Dummy {
private:
    Dummy(Dummy & source) { }
public:
    Dummy(int value) {}
};
void DoSomething(Dummy ob) {
    cout << "I'm here!" << endl;
}
int main(void) {
    Dummy o1(123);
    Dummy o2 = o1;
    DoSomething(o1);
    return 0;
}
```

In some cases, you may want to prevent any (any!) use of the copying constructor forcing the user of your class to construct its objects in a more detailed way (e.g. using any of the explicitly described constructors).

All you have to do is specify the explicit copying constructor and put it inside the private part of your class.

Any attempt to make use of the copying constructor (whether implicit or explicit) will cause a **compilation error**.

```
error: 'Dummy::Dummy(Dummy&)' is private
```

6.5.7 More about default constructors (1)

```
#include <iostream>
using namespace std;
class NoConstructorsAtAll {
public:
    int i;
    float f;
    void Display(void) {
        cout << "i=" << i << ",f=" << f << endl;
    }
};
```

```
int main(void) {
    NoConstructorsAtAll o1;
    NoConstructorsAtAll *o2;
    o2 = new NoConstructorsAtAll;
    o1.Display();
    o2 -> Display();
    return 0;
}
```

A class that doesn't have a constructor at all. the class will be implicitly equipped with the so-called **implicit default (parameter-less) constructor** but the constructor will do nothing at all.

The class has no constructor. In effect their fields will not be initialized in any way. The values outputted by the *display* method are completely random.

```
i=72,f=1.12104e-044
i=0,f=0
```

6.5.8 More about default constructors (2)

```
#include <iostream>
using namespace std;
class WithConstructor {
public:
    int i;
    float f;
    WithConstructor(int a, float b) : i(a), f(b) { }
    void Display(void) {
        cout << "i=" << i << ",f=" << f << endl; }
};

int main(void) {
    WithConstructor o1;
    WithConstructor *o2;
    o2 = new WithConstructor;
    o1.Display();
    o2 -> Display();
    return 0;
}
```

There's a constructor within the class. The existence of any constructor is like a statement given by a developer: *"I'm ready to use constructors in my class"*.

How can we solve the problem? we can write our own default (parameter-less) constructor.

```
WithConstructor(void) : i(0), f(0.0) { }
```

There's also at least one simpler (and slightly surprising) solution.

```
error: no matching function for call to 'WithConstructor::WithConstructor()'
note: candidates are:
note: WithConstructor::WithConstructor(int, float)
```

6.5.9 More about default constructors (3)

```
#include <iostream>
using namespace std;
class WithConstructor {
public:
    int i;
    float f;
    WithConstructor(int a = 0, float b = 0) : i(a), f(b) { }
    void Display(void) {
        cout << "i=" << i << ",f=" << f << endl; }
};
int main(void) {
    WithConstructor o1;
    WithConstructor *o2;
    o2 = new WithConstructor;
    o1.Display();
    o2 -> Display();
    return 0;
}
```

```
i=0,f=0
i=0,f=0
```

we've changed the header of the existing constructor by adding default values to both parameters. This means that from a compiler's perspective, an invocation like this one

WithConstructor()
that it corresponds to the following invocation:

WithConstructor(0, 0.0)

6.5.10 Compositions vs. constructors (1)

```
#include <iostream>
using namespace std;
class A {
public:
    void Do(void) {
        cout << "A is doing something" << endl; }
};
class B {
public:
    void Do(void) {
        cout << "B is doing something" << endl; }
};
class Compo {
public:
    A f1;
    B f2;
};
```

```
A is doing something
B is doing something
```

This example shows a simple dummy class consisting (composed) of two objects of two independent classes.

```
int main(void) {
    Compo co;
    co.f1.Do();
    co.f2.Do();
    return 0;
}
```

We can say that any complex structure is **composed** using simpler elements – for example, a car is composed of an engine, transmission, suspension, etc. If we imagine all these parts as classes we'll see the car class as a composition that has nothing to do with inheritance.

6.5.11 Compositions vs. constructors (2)

its components have been modified. We've equipped them with copying constructors. They only emit a message allowing us to notice that the constructor has been invoked.

```
#include <iostream>
using namespace std;
class A {
public:
```

```
    A(A &src) { cout << "copying A..." << endl; }
```

```
    A(void) { }
```

```
    void Do(void) {
```

```
        cout << "A is doing something" << endl; }
```

```
};
```

```
class B {
```

```
public:
```

```
    B(B &src) { cout << "copying B..." << endl; }
```

```
    B(void){ }
```

```
    void Do(void) {
```

```
        cout << "B is doing something" << endl; }
```

```
};
```

```
copying A...
copying B...
A is doing something
B is doing something
```

```
class Compo {
public:
```

```
    Compo(void) { } ;
```

```
    A f1;
```

```
    B f2;
```

```
};
```

```
int main(void) {
```

```
    Compo co1;
```

```
    Compo co2 = co1;
```

```
    co2.f1.Do();
```

```
    co2.f2.Do();
```

```
    return 0;
```

```
}
```


6.5.12 Compositions vs. constructors (3)

```
#include <iostream>
using namespace std;
```

```
class A {
public:
    A(A &src) { cout << "copying A..." << endl; }
    A(void) { }
    void Do(void) { cout << "A is doing something" << endl; }
};
```

Copying Compo...
A is doing something
B is doing something

```
class B {
public:
    B(B &src) { cout << "copying B..." << endl; }
    B(void){ }
    void Do(void) { cout << "B is doing something" << endl; }
};
```

```
Compo(Compo &src) : f1(src.f1), f2(src.f2) {
    cout << "Copying Compo..." << endl; }
```

to call the copying constructor of A and B you must a line like this one

```
class Compo {
public:
    Compo(Compo &src) {
        cout << "Copying Compo..." << endl; }
    Compo(void) { };
    A f1;
    B f2;
};
```

The explicit copying constructor has invoked none of the component's copying constructors.

```
int main(void) {
    Compo co1;
    Compo co2 = co1;
    co2.f1.Do();
    co2.f2.Do();
    return 0;
}
```

It now has its own copying constructor. It'll be implicitly invoked once during the life of our program at the moment the `co2` object is created.

6.5.12 Compositions vs. constructors (3) suite

```
#include <iostream>
using namespace std;
class A {
public:
    A(A &src) { cout << "copying A..." << endl; }
    A(void) { }
    void Do(void) { cout << "A is doing
something" << endl; }
};
class B {
public:
    B(B &src) { cout << "copying B..." << endl; }
    B(void){ }
    void Do(void) { cout << "B is doing
something" << endl; }
};
```

```
class Compo {
public:
    Compo(Compo &src) : f1(src.f1), f2(src.f2) {
        cout << "Copying Compo..." << endl; }
    Compo(void) { } ;
    A f1;
    B f2;
};
int main(void) {
    Compo co1;
    Compo co2 = co1;
    co2.f1.Do();
    co2.f2.Do();
    return 0;
}
```

copying A...
copying B...
Copying Compo...
A is doing something
B is doing something

6.6 The const keyword

6.6.1 What is a constant anyway?

how the “C++” sees **constants**.

On l'a pas varier (changer)
Toujours même valeur malgré ceci non

```
const int size1 = 100;  
int const size2 = 100;  
size2 = size1;//non
```

In general, “C++” compilers don't think that “*const*” means “invariable”. They're more likely convinced that “*const*” is “unalterable”.

Note that the *const* keyword is located in different places in each line. **Both forms are acceptable.**

```
size1++;
```

```
size2 = size1;
```

```
const int size = 100;
```

```
int buffer[size];
```

```
const int size1 = 100;  
int const size2 = 100;
```

```
int size = 100;
```

```
int buffer[size];
```

(some compilers **allow** for the compilation of this example, because they have special extension implemented)

Note: you mustn't declare a *const* without initialization (think about this for a moment and you'll find it obvious). The following line will cause a compilation error:

```
const int size;
```

6.6.3 Constant aggregates

Aggregates (structures and arrays as well as arrays of structures and structures of arrays *et cetera*) may be declared as *const* too, although the effects are somewhat different.

```
const int points[5] = {1, 2, 4, 8, 16};
```

```
const struct {int key;} data = {10};
```

***points* and *data* are read-only variables and you mustn't modify them.**

```
--points[2];
```

```
data.key = 0;
```

Rappel:

```
const int size = 100;
```

```
int buffer[size];
```

contrary to the **previous examples**, you can't treat these symbols as literals.
Some of the "C++" compilers may consider the following line as incorrect:

```
int array[points[2] + data.key];
```

as the compiler may not be able to determine the number of the array's elements during the compile time

// Pointeur sur une constante

```
const int *a = &constante_entiere;
```

```
int const *b = &constante_entiere; // équivalent à la ligne précédente
```

```
a = &constante_entiere;
```

```
*a = 4; // ERREUR : on ne modifie pas une constante
```

// Pointeur constant sur une variable

```
int *const c = &entier;
```

```
int *const c; // ERREUR : initialisation manquante
```

```
c = &entier1; // ERREUR : on ne modifie pas une constante
```

```
*c = 5;
```

// Pointeur constant sur une constante

```
const int *const d = &constante_entiere;
```

```
int const *const e = &constante_entiere; // équivalent à la ligne précédente
```

```
const int *const f; // ERREUR : initialisation manquante
```

```
d = &entier1; // ERREUR : on ne modifie pas une constante
```

```
*d = 5; // ERREUR : on ne modifie pas une constante
```

6.6.4 Constant pointers (**Pointeur constant** sur une variable)

Pointers are allowed to be declared as `const` as well, **Note** that the *const* keyword is placed **after the *** and **before the variable name**

```
int arr[5] = {1, 2, 4, 8, 16};
```

```
int * const iptr = arr + 2;
```

iptr **mustn't be modified**. This means that the following lines will cause compilation errors: **--iptr;**

The entities pointed to by the **const pointers** may be modified with no restrictions. The following two lines will be accepted and successfully performed: ***iptr = 0;**

6.6.5 Pointers to constants(**Pointeur sur une constante**)

Constant pointers aren't equivalents for pointers to constants – don't forget that, it's important. The *const* keywords have changed their locations and now they're placed at the beginning of the declarations.

```
int arr[5] = {1, 2, 4, 8, 16};  
const int *iptr = arr + 2;  
const char *cptr = "Why?";
```

Note that the following form is correct too

Both *iptr* and *cptr* may be modified. The following lines are correct:

```
--iptr;  
++cptr;
```

```
int const *iptr = arr + 2;  
char const *cptr = "Why?";
```

In contrast, **the entities pointed to by these pointers cannot be modified any more.** The following two lines will not be accepted:

```
*iptr = 0;  
*cptr = 'T';
```


6.6.6 Constant pointers to constants (Pointeur constant sur une constante)

Both of the above variants can be mixed together giving a *const* pointer to a *const* value.

Take a look at the following snippet →

```
int arr[5] = {1, 2, 4, 8, 16};  
const int * const iptr = arr + 2;  
const char * const cptr = "Why?";
```

None of the following lines are correct in the scope of this declaration:

--iptr;

++cptr;

*iptr = 0;

*cptr = 'T';

6.6.7 Constant function parameters (1)

*Constant function parameters (passed **by value**)*

Any of the function parameters passed by value may be declared as *const*.

Note that the **effects** of these declarations are **only** observable **inside** the function and have no impact on the outside world.

The n parameter must not be modified by the function although that modification won't be propagated outside the function anyway.

This snippet is correct, as the n isn't modified in any way inside the function.

```
int fun(const int n) {  
    return n * n;  
}
```

6.6.8 Constant function parameters (2)

*Constant function parameters (passed **by reference**)*

Any of the function parameters passed by reference may be declared as *const*.

We can say that this is a **stronger form of the previous declaration**. We can understand it as a solemn **promise** made by the function: *I'm not going to modify your actual parameter.*

Take a look at the following snippet →

```
int fun(const int &n) {  
    return n++;  
}
```

The snippet is **incorrect**, as the function tries to break the promise. The compiler won't compromise on this.

6.6.9 Constant function results

Any function may declare its result as *const*.

This form doesn't make much sense when applied to the values of standard types (like *int* or *float*), but may be essential when used with **pointers**, **aggregates** and **objects**.

Declarations like these mean: *"I will give you a value that you mustn't change"*.

There are many reasons for this warning. One of the most obvious is that the value returned is located in the read-only memory.

```
const char *fun(void) {  
    return "Caution!";  
}
```

This line will be rejected by the compiler:

char *p = fun();

This one will be accepted:

const char *str = fun();

6.6.9 Constant function results (bis)

The *fun* function returns a pointer to the C-style string. We already know that the string is unalterable, so we emphasise this fact through the form of the function declaration. In effect, the result of the function **may be assigned only to a variable that guarantees safety**.

This line will be rejected by the compiler:

```
char *p = fun();
```

This one will be accepted:

```
const char *str = fun();
```

```
const char *fun(void) {  
    return "Caution!";  
}
```

6.6.10 Constant class variables

Any class may declare its field as *const*.

when applied to a regular variable. It means that the field is **unalterable during the object's life**, and nothing more. Different objects may have these fields assigned with different values. We can say that **constancy is limited to the boundaries of a single object**.

A *const* class field must be initialized inside an initialization list within any of the class constructors.

Any other assignment will be rejected.

All of the constructors initialize the *const field* with a different value.

All the initializations are valid.

```
class Class {  
private:  
    const int field;  
public:  
    Class(int n) : field(n) { };  
    Class(Class &c) : field(0) { };  
    Class(void) : field(1) { };  
    Class(double f) { field = f; }  
    void fun(int n) { field += n; }  
};
```

The following snippets, inserted inside the public part of the *Class*, will be recognized as invalid

6.6.10 Constant class variables

A *const* class field must be initialized inside an initialization list **within any of the class constructors**.

Any other assignment will be rejected.

```
class Class {  
private:  
    const int field;  
public:  
    Class(int n) : field(n) { };  
    Class(Class &c) : field(0) { };  
    Class(void) : field(1) { };  
    Class(double f) { field = f; } error: uninitialized const member in 'const int' [-fpermissive]  
    void fun(int n) { field += n; } error: assignment of read-only member 'Class::field'  
};
```

6.6.11 Constant objects

An object of any class may be declared as *const*. This means that the **object mustn't be modified during its life**. The compiler will protect the object from any attempts to try to change its state. So:

- directly modifying any object's fields is **prohibited**
- invoking any object's member functions is **prohibited**

```
Class o1(1);  
const Class o2(2);  
int i;
```

three lines will be rejected

```
o2.field = 3;  
o2.set(1);  
i = o2.get();
```

```
class Class {  
public:  
    int field;  
    Class(int n) : field(n) { };  
    Class(Class &c) : field(0) { };  
    Class(void) : field(1) { };  
    void set(int n) { field = n; }  
    int get(void) { return field; }  
};
```

They'll be considered valid if you replace 'o2' with 'o1'.

Note that **it doesn't matter if the member function modifies the state of the object** : invoking the *get()* function from the *o2* object is prohibited too, although the function only reads the object's field

6.6.12 Constant member functions

Any of the class's member functions may declare themselves as *const*.

This is a promise that **the function won't modify the state of the object**.

Note the *const* keyword is placed after the parameter list, like this:

type name(parameters) const ;	in declarations
type name(parameters) const { ... }	in definitions

the following line will be considered valid now: **i = o2.get();**

Note: the compiler will try to force the program to keep the promise. You shouldn't modify any of the class variables or invoke non-const functions inside the *get* function. These will be recognized as errors.

```
class Class {  
public:  
    int field;  
    Class(int n) : field(n) { };  
    Class(Class &c) : field(0) { };  
    Class(void) : field(1) { };  
    void set(int n) { field = n; }  
    int get(void) const { return field; }  
};
```

6.7 Friendship in the “C++” world



6.7.1 Friend or foe?

Classes can have friends too. A friend of a class may be:

- a **class** (it's called the **friend class**)

- a **function** (it's called the **friend function**)

A friend (class or function) can access those components hidden from others.

Friends are allowed to access or to use private and protected components of the class.

```
#include <iostream>
using namespace std;
```

6.7.2 How to say it?

the line starting with the phrase: **friend class** ... ;
may exist inside any of the class parts (**public**, **private** or **protected**),
but must be placed **outside** any function or aggregate.

```
class Class {
friend class Friend;
```

private:

The announcement works in one direction only

```
    int field;
    void print(void) {
        cout << "It's a secret, that field = " << field << endl; }
};
```

```
class Friend {
public:
```

```
    void Dolt(Class &c) { c.field = 100; c.print(); }
};
```

```
int main(void) {
    Class o;
    Friend f;

    f.Dolt(o);
    return 0;
}
```

```
It's a secret, that field = 100
```

an object of the *Friend* class is able to
manipulate the *Class*'s private fields
and invoke its private methods.

There are some additional rules that must be taken into account:

- a class may **be a friend of many classes**
- a class may **have many friends**
- a friend's friend **isn't my friend**
- friendship **isn't inherited** – the subclass has to define its own friendships

```
#include <iostream>
using namespace std;
class A {
friend class B;
friend class C;
private:
    int field;
protected:
    void print(void) {
        cout << "It's a secret, that field = "
            << field << endl; }
};
```

6.7.3 The rules

```
class C {
public:
    void Dolt(A &a) { a.print(); }
};
class B {
public:
    void Dolt(A &a, C &c) {
        a.field = 111; c.Dolt(a); }
};
int main(void) {
    A a; B b; C c;
    b.Dolt(a,c);
    return 0; }
```

```
It's a secret, that field = 111
```

```
#include <iostream>
using namespace std;
class A;
class C {
public:
```

```
    void dec(A &a);    };
class A {
    The A class has three friends
```

```
friend class B;
    the B class
```

```
friend void C::dec(A&);
    the member function dec (from the C class)
```

```
friend void Dolt(A&);
```

```
private:
```

```
    int field;
    the global DoIt() function
```

```
protected:
    void print(void) {
        cout << "It's a secret, that field = "
        << field << endl; }
};
```

```
void C::dec(A &a) { a.field--; }
```

6.7.4 Friend functions

```
class B {
public:
    void Dolt(A &a) { a.print(); }
```

```
};
void Dolt(A &a) {
    a.field = 99;
}
```

```
int main(void) {
    A a; B b; C c;
```

```
    Dolt(a);
    b.Dolt(a);
    return 0;
}
```

A **function** may be a class's friend too. Such a function may access all the private and/or protected components of the class.

The rules are a bit different from before:

- a friendship declaration must contain a **complete prototype of the friend function** (including return type); a function with the same name, but incompatible in the sense of the parameters' conformance, will not be recognized as a friend
- a class **may have many friend functions**
- a function **may be a friend of many classes**
- a class may recognize as friends **both global and member functions**

It's a secret, that field = 99

Assessments

Exercise 1

```
#include <iostream>
using namespace std;
class X {
private:
    int v;
};
class Y : public X {
    Y():v(0){}
};
int main() {
    Y y;
    cout<< y.v;
    return 0;
}
```

What happens when you attempt to compile and run the following code?

Select correct answer (single choice)

- It prints 1
- It prints 0
- Compilation fails
- It prints -1


```
#include <iostream>
using namespace std;
class X {
protected:
    int v;
};
class Y : protected X {
    Y(): v(0) {}
};
int main() {
    Y *y = new Y();
    cout << y->v;
    delete y;
    return 0;
}
```

Exercise 2

What happens when you attempt to compile and run the following code?

Select correct answer (single choice)

- It prints 1
- It prints 0
- Compilation fails
- It prints -1

```
#include <iostream>
using namespace std;
```

```
class X {};
class Y : public X {};
class Z : public X {};
int main() {
    Z *z = new Z();
    Y *y = new Y();
    z = y;
    cout << (z == y);
    return 0;
}
```

Exercise 3

What happens when you attempt to compile and run the following code?

Select correct answer (single choice)

- It prints 1
- It prints 0
- Compilation fails
- It prints -1

```
#include <iostream>
using namespace std;
```

Exercise 4

```
class X { };
class Y : public X {};
class Z : public X {};
int main() {
    Z *z = new Z();
    X *x = new X();
    x = z;
    cout << (x == z);
    return 0;
}
```

What happens when you attempt to compile and run the following code?

Select correct answer (single choice)

- It prints 1
- It prints 0
- Compilation fails
- It prints -1

```
#include <iostream>
using namespace std;
```

```
class X {
public:
    void shout() { cout << "X"; }
};
class Y : public X {
public:
    void shout() { cout << "Y"; }
};
```

```
int main() {
    X *x = new Y();
    x->shout();
    return 0;
}
```

Exercise 5

What happens when you attempt to compile and run the following code?

Select correct answer (single choice)

- It prints X
- It prints Y
- Compilation fails
- It prints nothing

```
#include <iostream>
using namespace std;
class X {
public:
    virtual void shout() { cout << "X"; }
};
class Y : public X {
public:
    void shout() { cout << "Y"; }
};
class Z : public Y {
public:
    void shout() { cout << "Z"; }
};
```

Exercise 6

```
int main() {
    Y *y = new Z();
    dynamic_cast<X *>(y) -> shout();
    return 0;
}
```

What happens when you attempt to compile and run the following code?

Select correct answer (single choice)

- It prints X
- It prints Z
- Compilation fails
- It prints nothing

Exercise 7

```
#include <iostream>
using namespace std;
class A {
public:
    A(): val(0) {}
    int val;
    virtual void run() { cout << val;}
};
class B : public A {
};
class C : public B {
public:
    void run() { cout << val + 2;}
};
```

```
void Do(A *a) {
    B *b;
    C *c;
    if (b = dynamic_cast<B *>(a)) b->run();
    if (c = dynamic_cast<C *>(a)) c->run(); a->run();
}
int main() {
    A *a = new C();
    Do(a);
    return 0;
}
```

Select correct answer (single choice)

- It prints 210
- It prints 222
- Compilation fails
- It prints 212

Exercise 8

```
#include <iostream>
using namespace std;
class A {
    int *val;
public:
    A() { val = new int; *val = 0;}
    A(A &a) { val = new int; *val = a.get();}
    int get() { ++(*val); return *val; }
};

int main() {
    A a,b = a;
    cout << a.get() << b.get();
    return 0;
}
```

What happens when you attempt to compile and run the following code?

Select correct answer (single choice)

- It prints 22
- It prints 20
- Compilation fails
- It prints 21

Exercise 9

```
#include <iostream>
using namespace std;
```

```
class A {
public:
    const int v;
    A(int x): v(x + 1) {}
    int get() {return ++v;}
};

int main() {
    A a(2);
    cout << a.get();
    return 0;
}
```

What happens when you attempt to compile and run the following code?

Select correct answer (single choice)

- It prints 2
- It prints 1
- Compilation fails
- It prints 3

Exercise 10

```
#include <iostream>
using namespace std;
class A {
friend void f();
private:
    int field;
public:
    int set(int x) { return field = ++x ;}
    int get() { return ++field;}
};
void f(A &a) {a.field /= 2;}
int main() {
    A a;
    a.set(2);
    cout << a.get();
    return 0;
}
```

What happens when you attempt to compile and run the following code?

Select correct answer (single choice)

- It prints 1
- It prints 0
- Compilation fails
- It prints 2

CPA: Module 6 Quiz

This quiz will help you prepare for Module 6 Test.
You will have 25 minutes to answer 10 questions.
Click Start Quiz to begin. Good luck!

What is the output of the following program?

```
#include <iostream>
using namespace std;
class A {
public:
    char c;
};
class B : A {
};
int main(void) {
    B b;
    A a;
    a.c = b.c = '?';
    cout << int(a.c - b.c) << endl;
    return 0;
}
```

☐ 2

☐ the program will cause a compilation error

☐ 1

☐ 4

What is the output of the following program?

```
#include <iostream>
using namespace std;
class A {
public:
    int x;
    void d() { x /= 2; }
};
class B : public A {
public:
    int y;
    void d() { y /= 4; }
};
int main(void) {
    B b;
    b.x = b.y = 4;
    b.d();
    cout << b.x / b.y << endl;
    return 0;
}
```

☐ 2

☐ the program will cause a compilation error

☐ 4

☐ 1

What is the output of the following program?

```
#include <iostream>
using namespace std;
class A {
public:
    int x;
    void d() { x /= 2; }
};
class B : public A {
public:
    int y;
    void d() { A::d(); }
};
int main(void) {
    B b;
    b.x = b.y = 4;
    b.d();
    cout << b.y / b.x << endl;
    return 0;
}
```

☐ 2

☐ the program will cause a compilation error

☐ 1

☐ 4

What is the output of the following program?

```
#include <iostream>
using namespace std;
class A {
public:
    int work(void) { return 4; }
};
class B : public A {
public:
    int relax(void) { return 2; }
};
class C : public A {
public:
    int relax(void) { return 1; }
};
int main(void) {
    A *a0 = new A, *a1 = new B, *a2 = new C;
    cout << a0 -> work() + static_cast<C*>(a2) -> relax() / static_cast<B*>(a1) -> relax() << endl;
    return 0;
}
```

☐ 1

☐ the program will cause a compilation error

☐ 4

☐ 2

What is the output of the following program?

```
#include <iostream>
using namespace std;
class A {
public:
    int p(void) { return 2; }
};
class B : public A {
public:
    int p(void) { return 1; }
};
int main(void) {
    A *a = new B;
    cout << static_cast<A*>(a)->p() << endl;
    return 0;
}
```

- ☐ the program will cause a compilation error
- ☐ 1
- ☐ 2
- ☐ 4

What is the output of the following program?

```
#include <iostream>
using namespace std;
void f(const int &v) {
    ++v;
    return v + 1;
}
int main(void) {
    int i = 1, j = f(i);
    cout << j - i << endl;
    return 0;
}
```

☐ 1

☐ the program will cause
a compilation error

☐ 4

☐ 2

What is the output of the following program?

```
#include <iostream>
using namespace std;
class A { friend class B;
    int a;
public:
    A() : a(1) {}
    int f() { return a; }
};
class B {
public:
    static void f(A &a) { a.a++; }
};
int main(void) {
    A a;
    B::f(a);
    cout << a.f() << endl;
    return 0;
}
```

☐ 2

☐ 4

☐ 1

☐ the program will cause a compilation error

What is the output of the following program?

```
#include <iostream>
using namespace std;
class A { friend void i(A*);
    int a;
public:
    A() : a(2) {}
    int f() { return a; }
};
void i(A *a){
    a->a *= 2;
}
int main(void) {
    A a;
    i(&a);
    cout << a.f() << endl;
    return 0;
}
```

☐ 4

☐ 1

☐ the program will cause a compilation error

☐ 2

What is the output of the following program?

```
#include <iostream>
using namespace std;
class A { friend void i(int);
    int a;
public:
    A() : a(4) {}
    int f() { return a; }
};
void i(int a){
    a /= 2;
}
int main(void) {
    A a;
    i(a.a);
    cout << a.f() << endl;
    return 0;
}
```

☐ 2

☐ 1

☐ 4

☐ the program will cause a compilation error

What is the output of the following program?

```
#include <iostream>
using namespace std;
class B;
class A {
    friend class B;
    int a;
public:A() : a(4) {}
    void f(B &b,A &a);
    int out(void) { return a; }
};
class B {
    friend class A;
    int b;
public:B() : b(2) {}
    void f(A &a) { a.a /= b; }
};
void A::f(B &b,A &a){ b.f(*this); }
int main(void) {
    A a;
    B b;
    a.f(b,a);
    cout << a.out() << endl;
    return 0;
}
```

☐ the program will cause a compilation error

☐ 4

☐ 1

☐ 2

Fin