Object Oriented Methodology

Week – *3*, Lecture – *3*

Classes in C++ - Part 2

(Static Members, Copy Constructors, Single Inheritance and Friend Functions)

SAURABH SRIVASTAVA

VISITING FACULTY

IIIT LUCKNOW

What we know so far …

Namespaces are collections of declarations and definitions

- As long as things are unique within a namespace, it is fine !!

Classes can have fields and methods which could be visible to or hidden from the outside world

- The former are called public members and the latter are called private members
- The default behaviour is that members are private, unless defined in the public section
- We did not discuss the static members of the class last time, so we will do it now

Constructors are called whenever a new object for a class is created

- A class can have more than one constructor – the job of all of them is to initialise the object's fields
- You can provide initial values for some or all of the fields of the class
- We will cover one more type of constructor in this part of the lecture

Static Members of a class

Static members of a class – fields or methods – are essentially associated with the class …

- … as against the non-static members, which are rather associated with an object

Only one copy of a static field is maintained for the whole class

- Static variables are stored in a separate memory section and initialised before any object is created

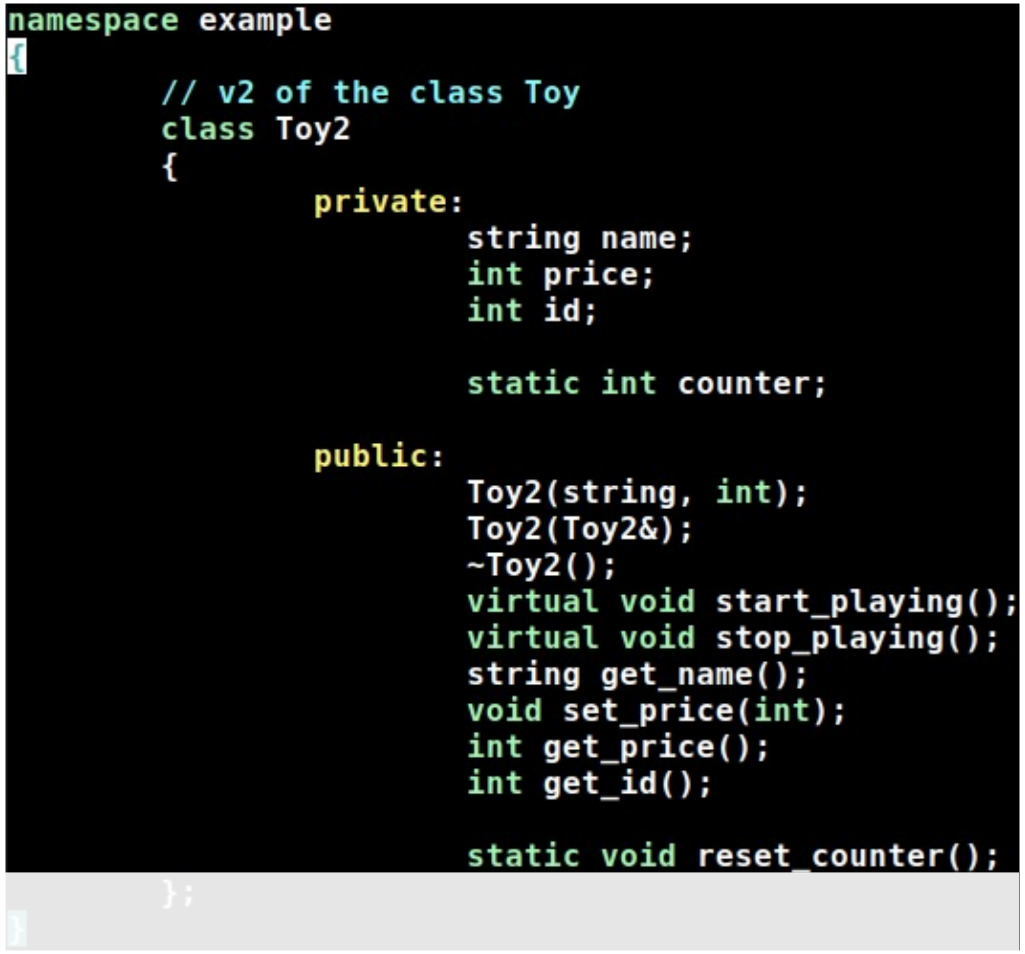
Static methods can only work with static fields of a class

- They can however, access the non-static members (including private members) through objects

Static members of a class can be accessed using the class's name with scope resolution operator

- For instance, `<class name>::<static field>` or `<class name>::<static method>`
- By the way, you can access them using an object as well, but logically, it doesn't make much sense !!
- The access rules (`public` or `private`) apply to static members in a similar fashion

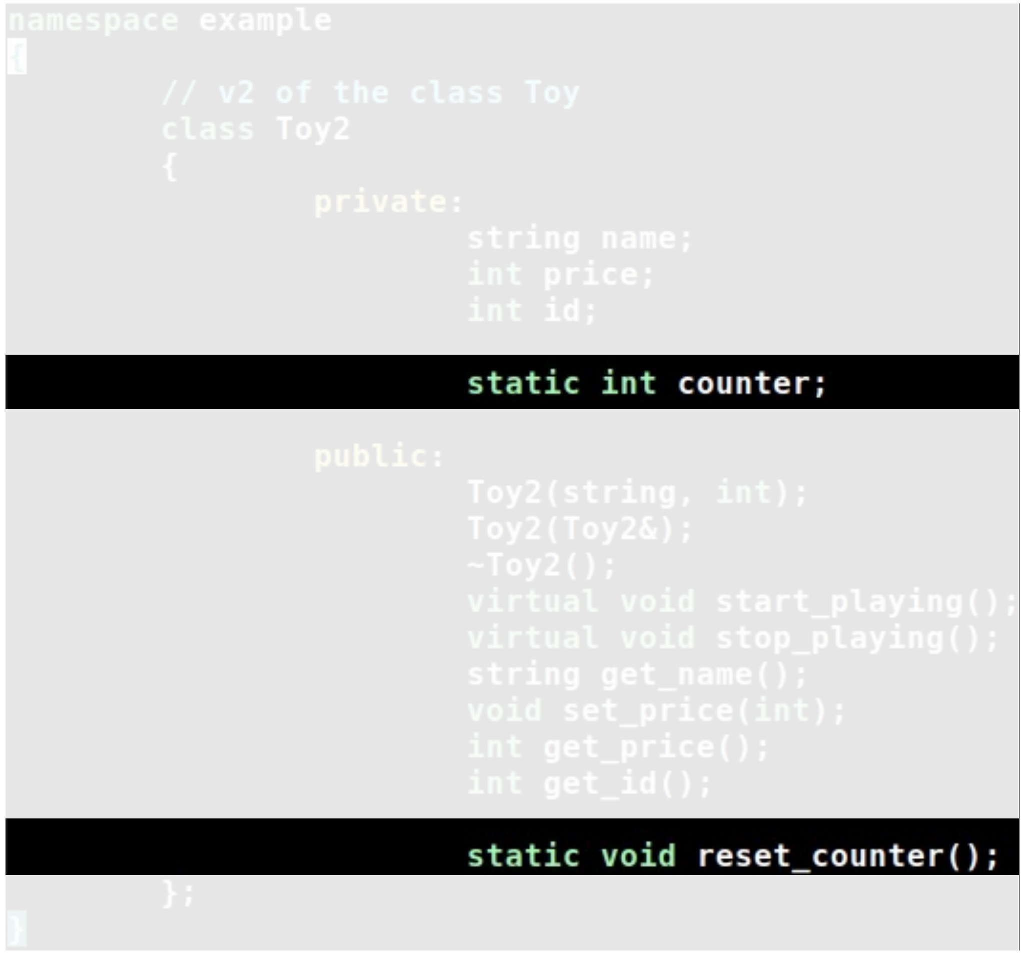
Extended Toy Example – static members



```
namespace example
{
        // v2 of the class Toy
        class Toy2
        {
                private:
                        string name;
                        int price;
                        int id;

                        static int counter;

                public:
                        Toy2(string, int);
                        Toy2(Toy2&);
                        ~Toy2();
                        virtual void start_playing();
                        virtual void stop_playing();
                        string get_name();
                        void set_price(int);
                        int get_price();
                        int get_id();

                        static void reset_counter();
        };
}
```

This is an extended version of the Toy class we saw last time

Extended Toy Example – static members



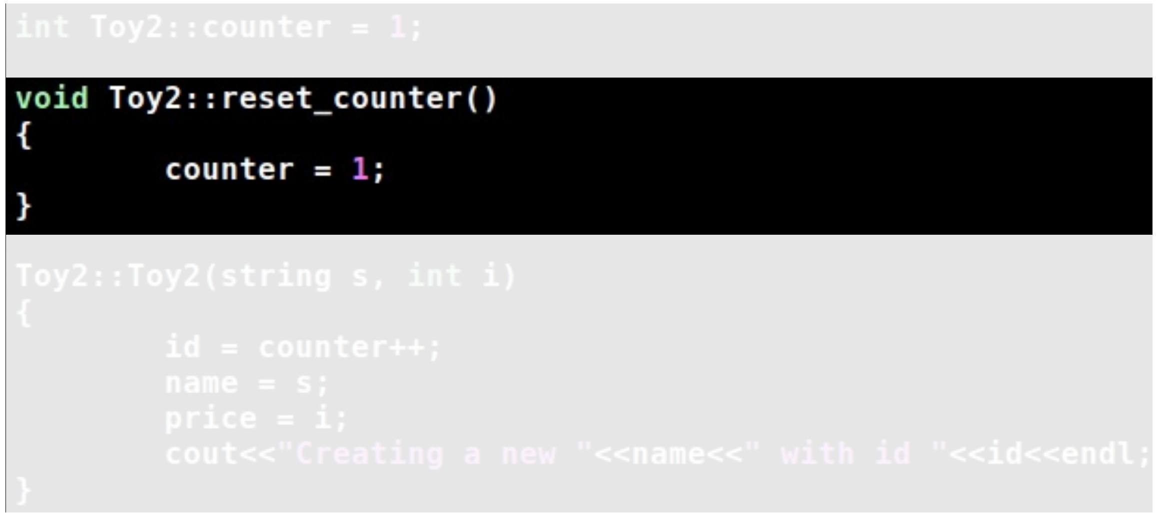
```
namespace example
{
        // v2 of the class Toy
        class Toy2
        {
                private:
                        string name;
                        int price;
                        int id;

                        static int counter;

                public:
                        Toy2(string, int);
                        Toy2(Toy2&);
                        ~Toy2();
                        virtual void start_playing();
                        virtual void stop_playing();
                        string get_name();
                        void set_price(int);
                        int get_price();
                        int get_id();

                        static void reset_counter();
        };
}
```

This is an extended version of the Toy class we saw last time

These are the static members of the class

A static field of a class is initialized by default with 0 (or an equivalent value)

Extended Toy Example – static members

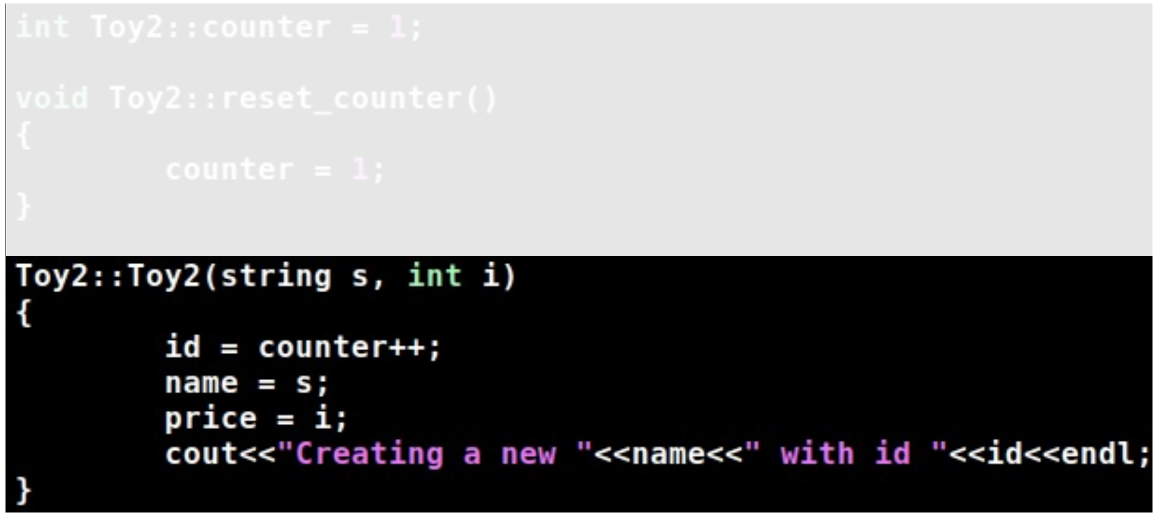


```
int Toy2::counter = 1;

void Toy2::reset_counter()
{
        counter = 1;
}

Toy2::Toy2(string s, int i)
{
        id = counter++;
        name = s;
        price = i;
        cout<<"Creating a new "<<name<<" with id "<<id<<endl;
}
```

At the time of definition, the static methods are defined exactly like a non-static method

It is just that it can only use the static fields of the class (because static fields too, are the property of the class and not associated with a particular object)

Extended Toy Example – static members



```
int Toy2::counter = 1;

void Toy2::reset_counter()
{
        counter = 1;
}
Toy2::Toy2(string s, int i)
{
        id = counter++;
        name = s;
        price = i;
        cout<<"Creating a new "<<name<<" with id "<<id<<endl;
}
```

Here, we are using the static field id as a way to provide a new identification value to each object of the class, as and when they are created

This is a fairly common use for static fields – keeping a count of the number of objects created

Copy Constructors

There is a special type of constructor, that is also added by added by the compiler implicitly

- However, a Copy Constructor can never be used for the first object of a class …
- … or for any subsequent objects, unless the required action is – creating a copy of an existing object

The constructor, called a *Copy Constructor* can also be defined explicitly

In most cases, it is not required to provide a user-defined copy constructor

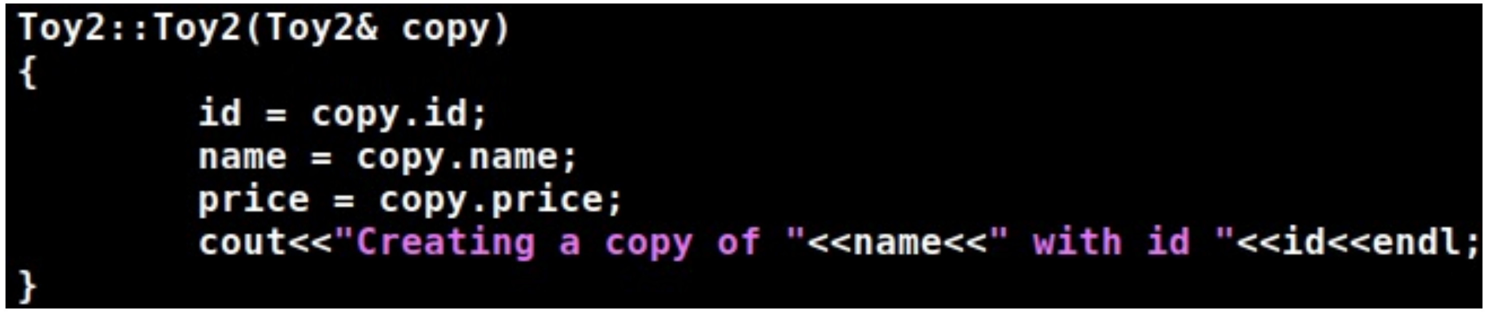
- There are some peculiar cases, where you may need to define them (check the homework)

A copy constructor is called implicitly whenever the compiler needs to create a copy of an object

- For example, when you pass an object to a function by value

You can also use it to create a copy of an object explicitly in your code

Extended Toy Example – copy constructer



```
Toy2::Toy2(Toy2& copy)
{
        id = copy.id;
        name = copy.name;
        price = copy.price;
        cout<<"Creating a copy of "<<name<<" with id "<<id<<endl;
}
```

This is what a constructor looks like

C++ has another way to pass parameters as compared to C – via a *reference* (check homework)

Basically, we pass an object of the same type to the constructor by reference

We can then copy the fields of the passed object to the newly created object

(Single) Inheritance in C++

Assume that you now want to make the class Toy a little more "specific"

- You also wish to store information about the type and number of batteries required to run the toy
- But, not all toys require batteries – some can be played with, without any battery as well

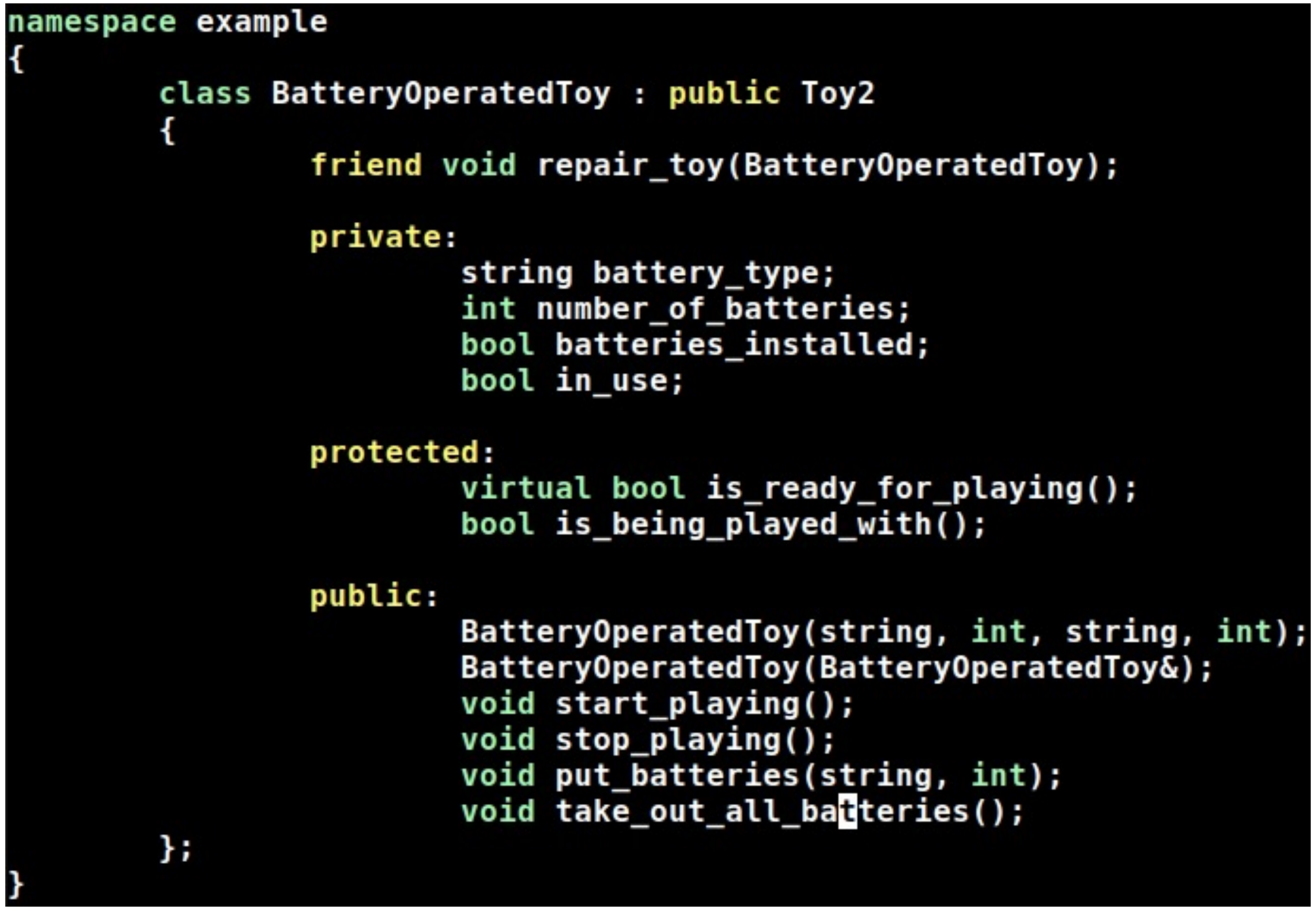
One option is to add some fields and methods in the Toy class to cover this aspect

- But then, we will need to adopt a convention like "Battery not required" for some Toys

Another option is to *extend* the Toy class and create a new class for "battery operated toys"

- Basically, a battery operated toy has all the properties and behaviour of a toy …
- … plus, some more properties and behaviour
- There is no point creating a copy of all fields and methods of the Toy class …
- … instead, through *Inheritance* we "import" this behaviour, and add more to the new class

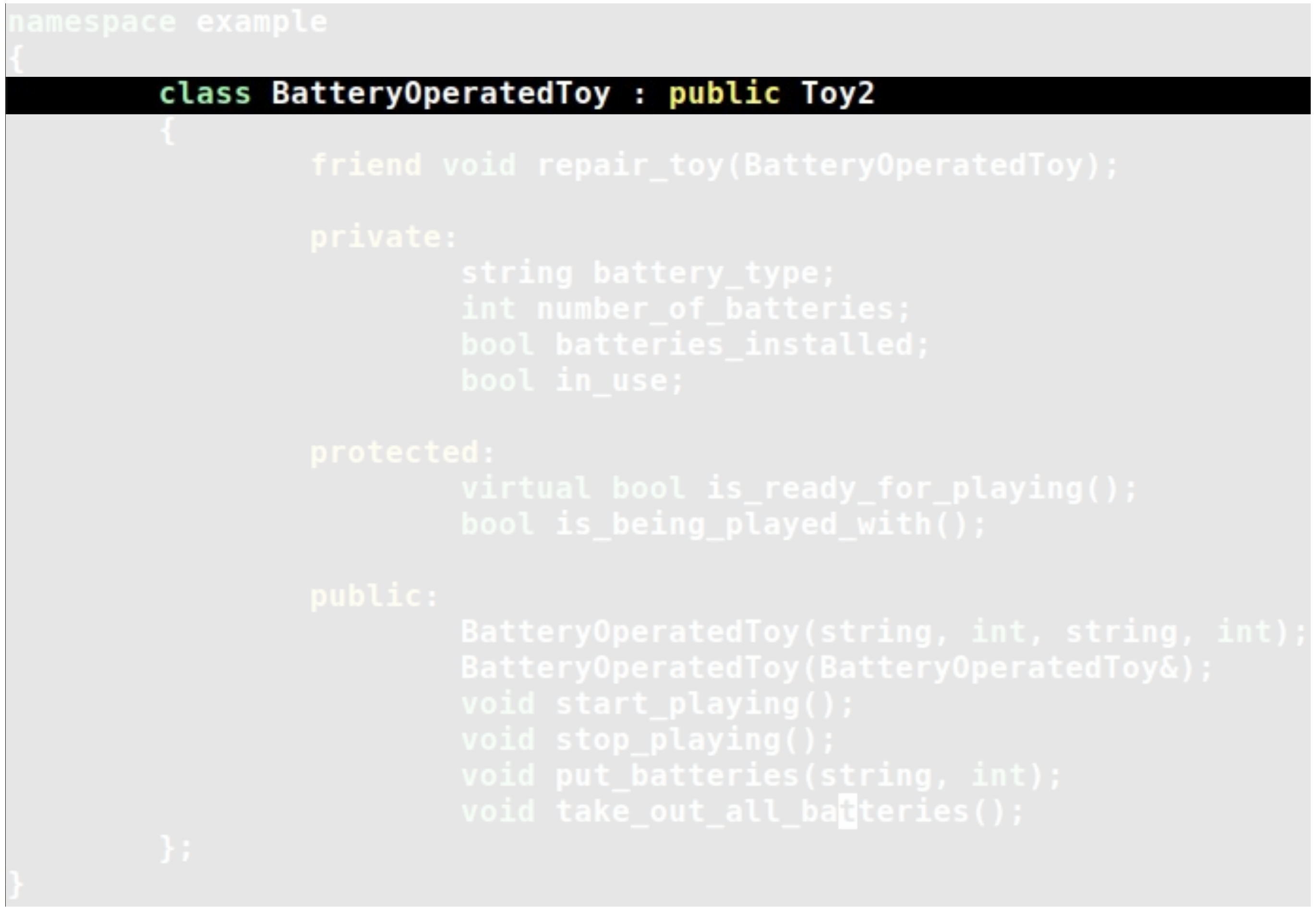
The Battery Operated Toy Example



```
namespace example
{
        class BatteryOperatedToy : public Toy2
        {
                friend void repair_toy(BatteryOperatedToy);

                private:
                        string battery_type;
                        int number_of_batteries;
                        bool batteries_installed;
                        bool in_use;

                protected:
                        virtual bool is_ready_for_playing();
                        bool is_being_played_with();

                public:
                        BatteryOperatedToy(string, int, string, int);
                        BatteryOperatedToy(BatteryOperatedToy&);
                        void start_playing();
                        void stop_playing();
                        void put_batteries(string, int);
                        void take_out_all_batteries();
        };
}
```

Here, we created another class to as a specific type of toys – those that require batteries

The Battery Operated Toy Example



```
namespace example
{
        class BatteryOperatedToy : public Toy2
        {
                friend void repair_toy(BatteryOperatedToy);

                private:
                        string battery_type;
                        int number_of_batteries;
                        bool batteries_installed;
                        bool in_use;

                protected:
                        virtual bool is_ready_for_playing();
                        bool is_being_played_with();

                public:
                        BatteryOperatedToy(string, int, string, int);
                        BatteryOperatedToy(BatteryOperatedToy&);
                        void start_playing();
                        void stop_playing();
                        void put_batteries(string, int);
                        void take_out_all_batteries();
        };
}
```

This is how we *inherit* the class `Toy2` in `BatteryOperatedToy`

The syntax is
`class <D> : <a> <B>`
where `D` is the derived class, `B` is the base class and `a` could be `public`, `private` or `protected`

We will discuss the `protected` access specifier shortly …

The access specifier here decides the access rights of the "inherited members"

The `protected` access specifier

Other than `public` and `private`, there is one more access specifier in C++ for class members

The specifier, `protected`, is similar to the specifier `private`, if there is no inheritance

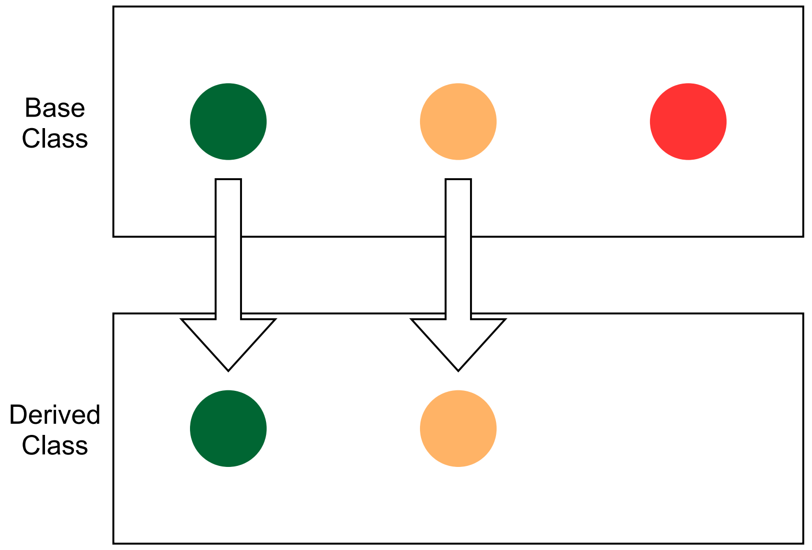
The `private` members of a class cannot be inherited by any derived class

- One way to let a member be inherited is by making it `public`
- But, it also provides access to the member to the outside world
- A mid path is to make these members `protected`

A `protected` member behaves the same way as a `private` member, with just one exception

- The `protected` members can be inherited by a derived class
- To reiterate, for every other case, `protected` and `private` mean exactly the same

Access specifiers in inheritance



● public members

● protected members

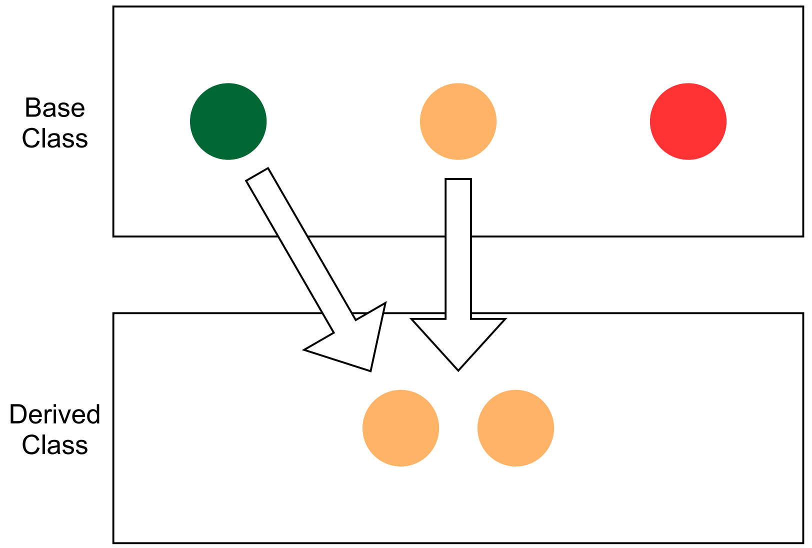
● private members

Scenario:
`class <D> : public <B>`

When the inheritance access specifier is `public`, the `public` members join the `public` section of the derived class, and the `protected` members join the `protected` section of the derived class

Remember, `private` members are not inherited

Access specifiers in inheritance



● public members

● protected members

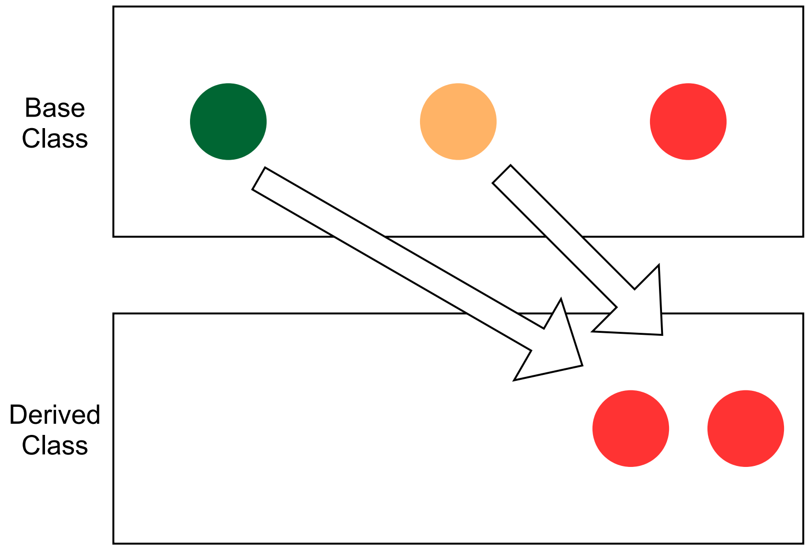
● private members

Scenario:
`class <D> : protected <B>`

When the inheritance access specifier is `protected`, both `public` as well as `protected` members join the `protected` section of the derived class

Remember, `private` members are not inherited

Access specifiers in inheritance



● public members

● protected members

● private members

Scenario:
`class <D> : private <B>`

When the inheritance access specifier is `private`, both `public` as well as `protected` members join the `private` section of the derived class

Remember, `private` members are not inherited

Constructors and Inheritance

When a class inherits another class, there is a *hierarchical* creation of the objects

This involves, creation of an object of the base class first, and then adding more fields to it …

- … thus making it an object of the derived type
- Keep in mind that space is allocated in the memory for even the private fields of the base class …
- … it is just that they are not accessible directly to the derived class

Thus, when a constructor of the derived class is called, "some" base class constructor is also invoked

- The base class constructor must be invoked before the derived class constructor is executed

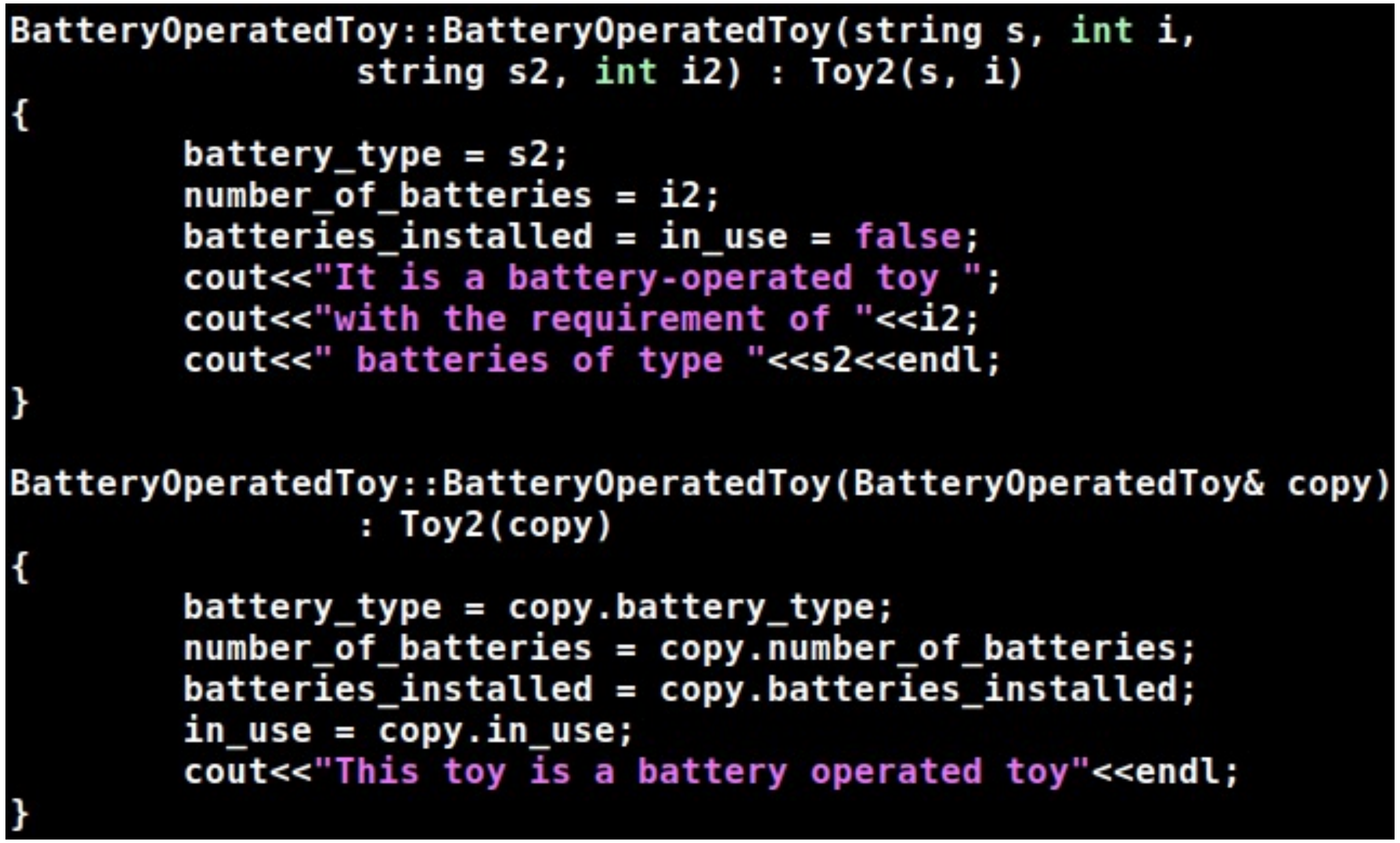
If there is a default constructor available in the base class, no additional measures are required …

- … the default constructor is automatically invoked, when any derived class constructor is called

However, if a default constructor is not present in the base class, some additional work is required

- Basically, all derived class constructors *must* make a call to some base class constructor

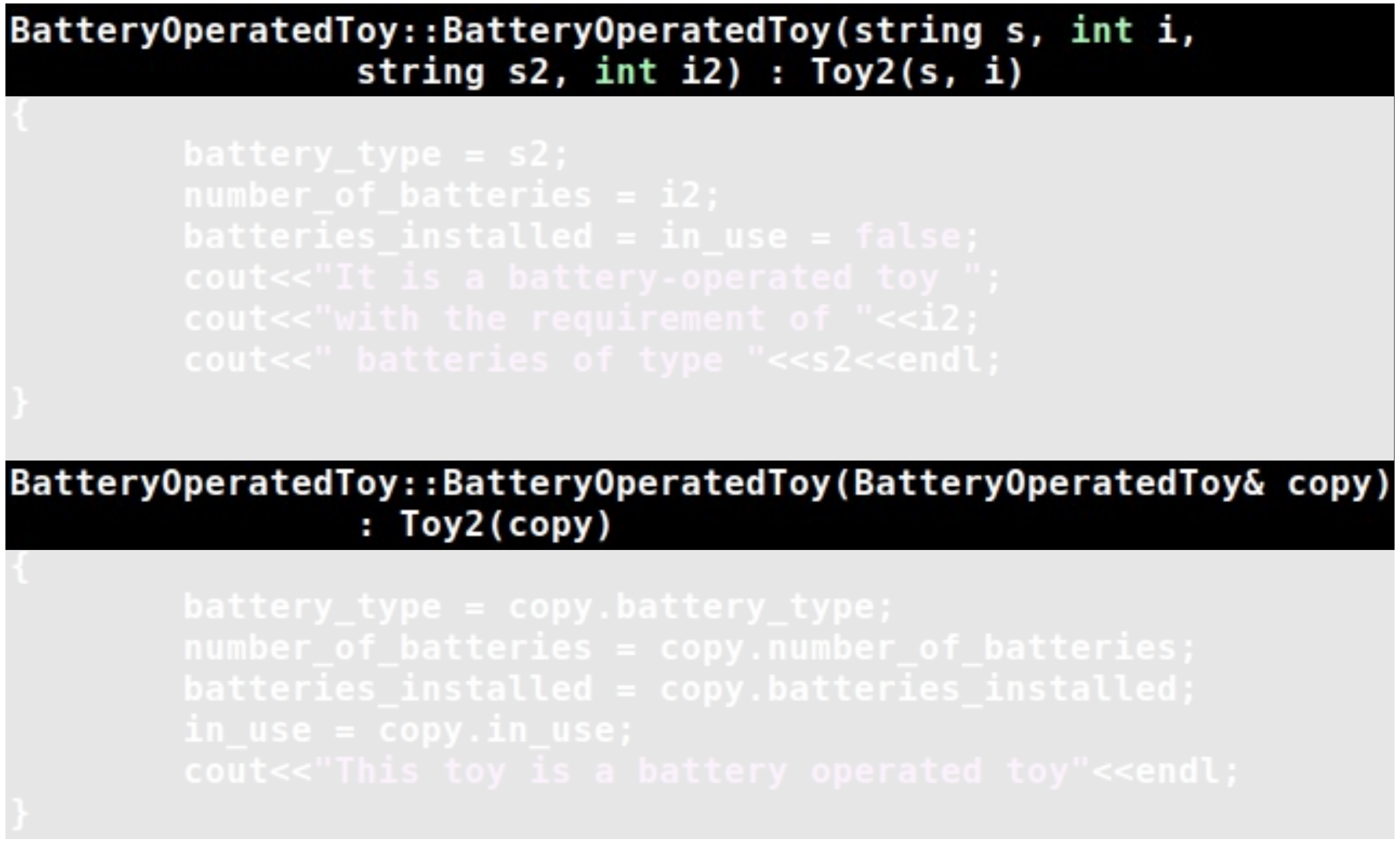
Calling base class constructors



```
BatteryOperatedToy::BatteryOperatedToy(string s, int i,
                string s2, int i2) : Toy2(s, i)
{
        battery_type = s2;
        number_of_batteries = i2;
        batteries_installed = in_use = false;
        cout<<"It is a battery-operated toy ";
        cout<<"with the requirement of "<<i2;
        cout<<" batteries of type "<<s2<<endl;
}

BatteryOperatedToy::BatteryOperatedToy(BatteryOperatedToy& copy)
                : Toy2(copy)
{
        battery_type = copy.battery_type;
        number_of_batteries = copy.number_of_batteries;
        batteries_installed = copy.batteries_installed;
        in_use = copy.in_use;
        cout<<"This toy is a battery operated toy"<<endl;
}
```

This is how an explicit call to a base class constructor is achieved in the derived class

Calling base class constructors



```
BatteryOperatedToy::BatteryOperatedToy(string s, int i,
                string s2, int i2) : Toy2(s, i)
{
        battery_type = s2;
        number_of_batteries = i2;
        batteries_installed = in_use = false;
        cout<<"It is a battery-operated toy ";
        cout<<"with the requirement of "<<i2;
        cout<<" batteries of type "<<s2<<endl;
}
BatteryOperatedToy::BatteryOperatedToy(BatteryOperatedToy& copy)
                : Toy2(copy)
{
        battery_type = copy.battery_type;
        number_of_batteries = copy.number_of_batteries;
        batteries_installed = copy.batteries_installed;
        in_use = copy.in_use;
        cout<<"This toy is a battery operated toy"<<endl;
}
```

Syntax:
`derived constructor` *`header`* `:`
`base constructor` *`call`*

Notice the difference – *header* vs *call*
We are making a “call” to a base constructor from the ”header” of the derived constructor

The only parameters that can be passed in the base constructor call are either the formal parameters of the derived constructor, or constants

Accessing `private` members externally

The general rule of thumb is that `private` members of a class are inaccessible externally

However, in some peculiar cases, you may wish to make an exception …

- … i.e. allow external access to the private members of the class

To do so, you have the option of creating *friendships* in you code !!

In C++, a non-member function can be declared as a `friend` of a class

A `friend` function has access to all the members of the class, even the `private` ones

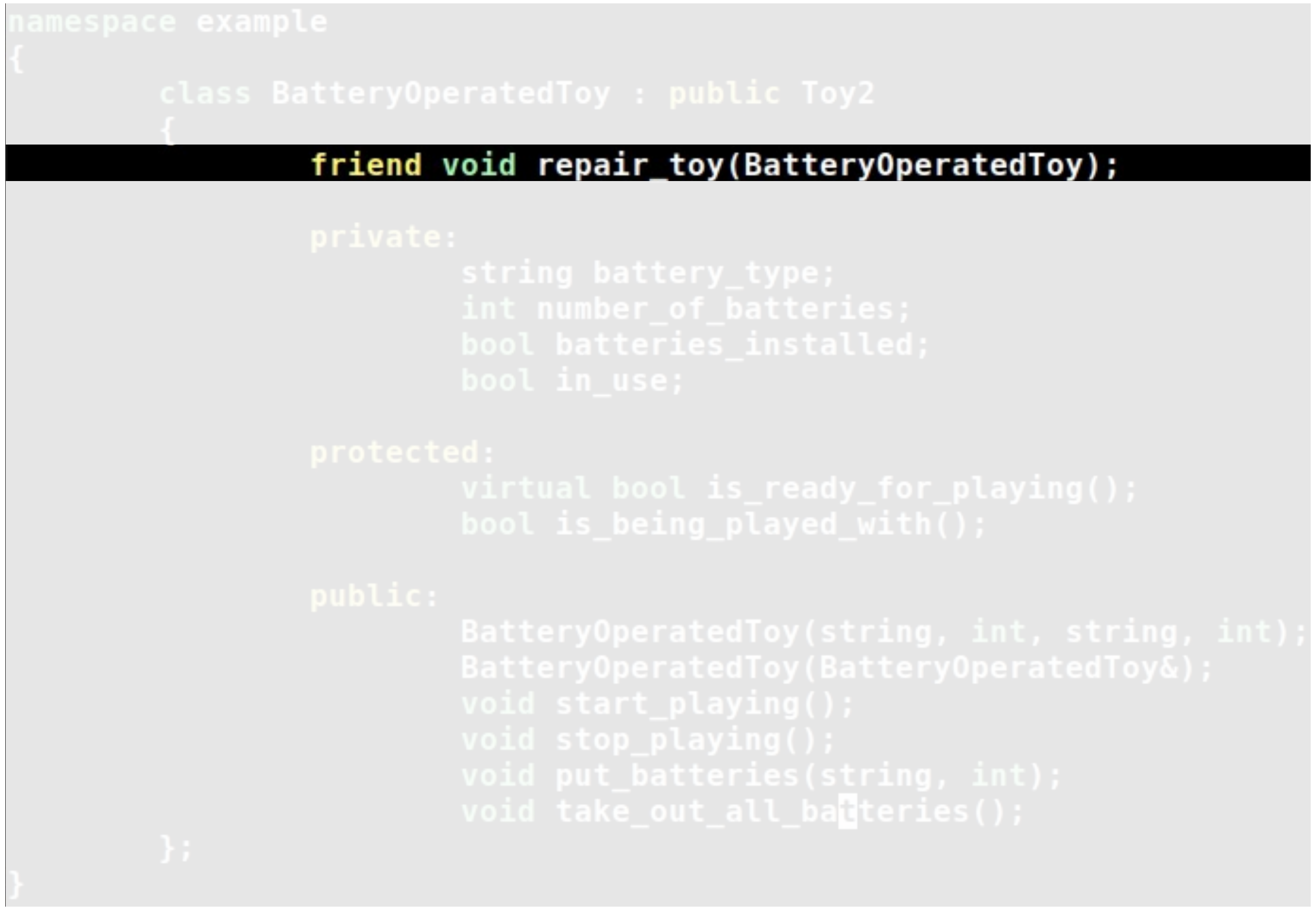
- It cannot, however, access the members like a member function – an object or class name is still required

A member function of one class, can be a `friend` of another class

If you wish to make all the member functions of a class as `friend`s of a class …

- … you can also make this entire class as a `friend` for the said class (called a `friend` class)

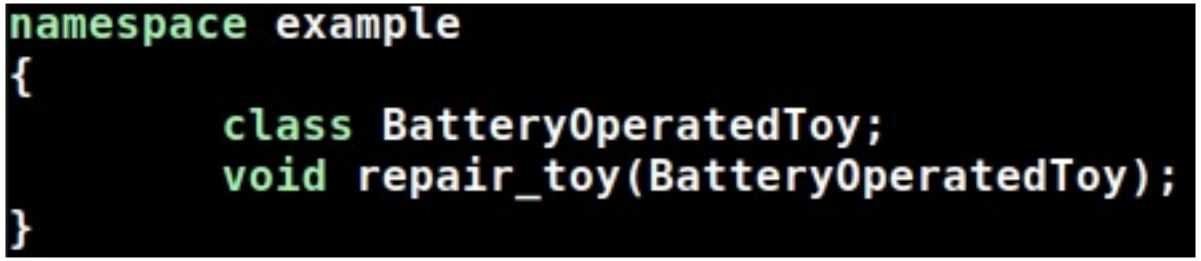
Friend Functions



```
namespace example
{
        class BatteryOperatedToy : public Toy2
        {
                friend void repair_toy(BatteryOperatedToy);

                private:
                        string battery_type;
                        int number_of_batteries;
                        bool batteries_installed;
                        bool in_use;

                protected:
                        virtual bool is_ready_for_playing();
                        bool is_being_played_with();

                public:
                        BatteryOperatedToy(string, int, string, int);
                        BatteryOperatedToy(BatteryOperatedToy&);
                        void start_playing();
                        void stop_playing();
                        void put_batteries(string, int);
                        void take_out_all_batteries();
        };
}
```

This is how you specify a function as a friend of a class – it looks similar to a function declaration, preceded by the keyword `friend`

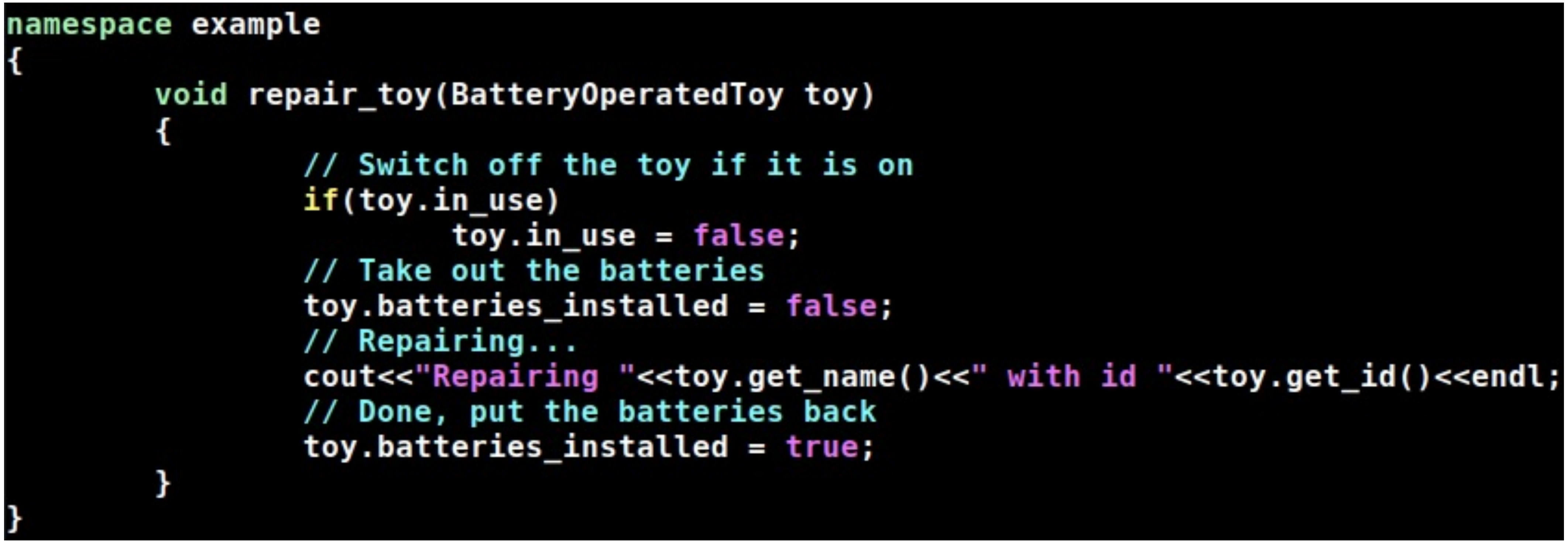
Friend Functions



```
namespace example
{
        class BatteryOperatedToy;
        void repair_toy(BatteryOperatedToy);
}
```

While declaring the function, it is a good practice to provide a declaration of the class for which this function will be a friend (called a *forward declaration*)

Friend Functions



```
namespace example
{
        void repair_toy(BatteryOperatedToy toy)
        {
                // Switch off the toy if it is on
                if(toy.in_use)
                        toy.in_use = false;
                // Take out the batteries
                toy.batteries_installed = false;
                // Repairing...
                cout<<"Repairing "<<toy.get_name()<<" with id "<<toy.get_id()<<endl;
                // Done, put the batteries back
                toy.batteries_installed = true;
        }
}
```

Friend functions can access the private members of a class using the `<object>.<member>` notation

Behaviour Extension Points – Virtual Methods

Sometimes, a method may be defined for a base class …

- … but it's definition may be "more precise" (or "different") with respect to a derived class

In such cases, what we may wish is …

- … if we have a base class object, the expected behaviour should be the one defined in the base class …
- … but we have a derived class object, the behaviour expected should be the one defined in the derived class

To achieve this in C++, the method can be defined as *virtual* in the base class

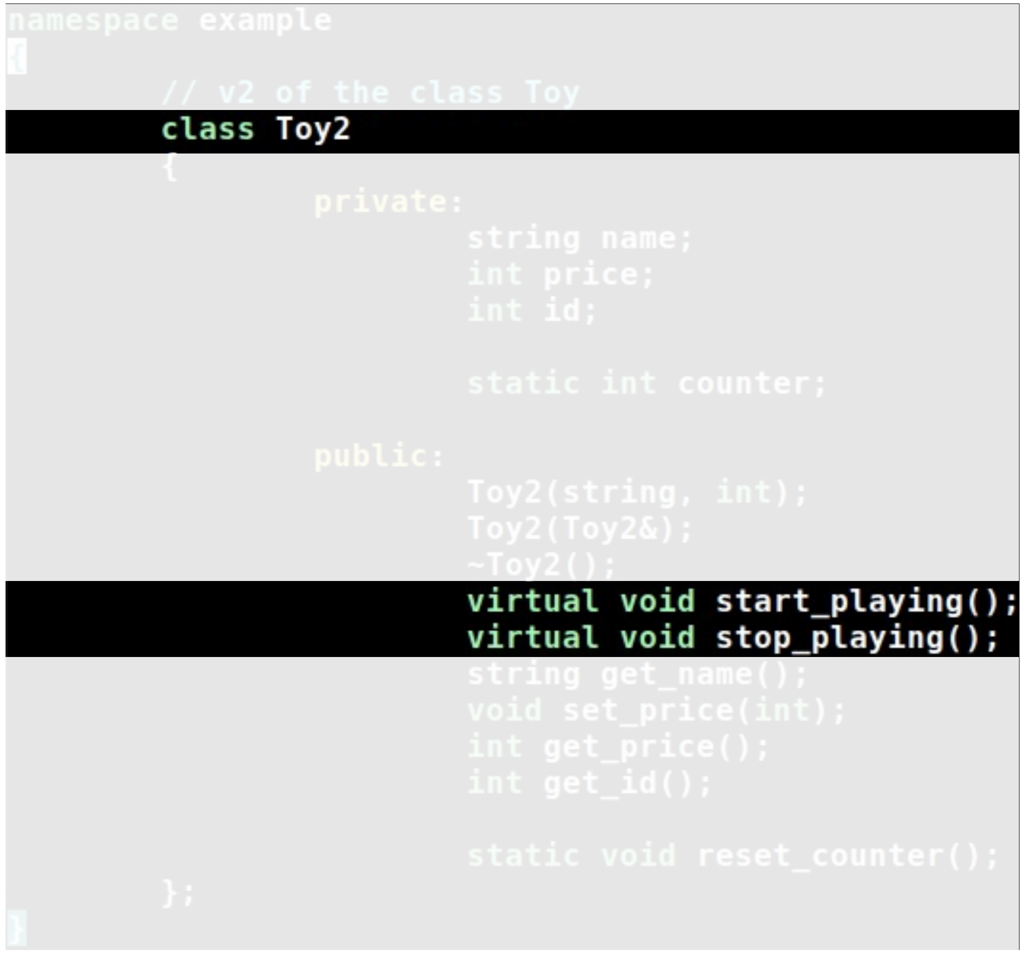
A virtual method indicates that multiple definitions of this method "may be" found at the runtime

The definition that must be invoked is decided by the type of the object

- If the object is of base class type, the definition from the base class is executed, and …
- … if the object is of derived class type, the definition executed is the one provided in the derived class

Keep in mind that only non-static methods can be declared as virtual

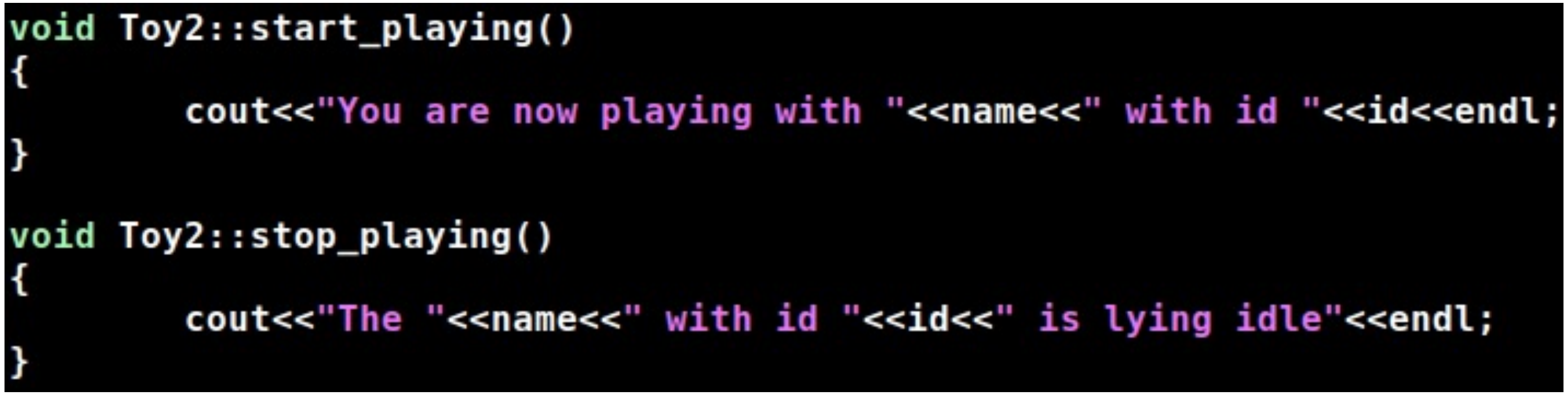
Virtual Functions



```
namespace example
{
        // v2 of the class Toy
        class Toy2
        {
                private:
                        string name;
                        int price;
                        int id;

                        static int counter;

                public:
                        Toy2(string, int);
                        Toy2(Toy2&);
                        ~Toy2();
                        virtual void start_playing();
                        virtual void stop_playing();
                        string get_name();
                        void set_price(int);
                        int get_price();
                        int get_id();

                        static void reset_counter();
        };
}
```

This is how a virtual function can be declared – by prepending the keyword `virtual`

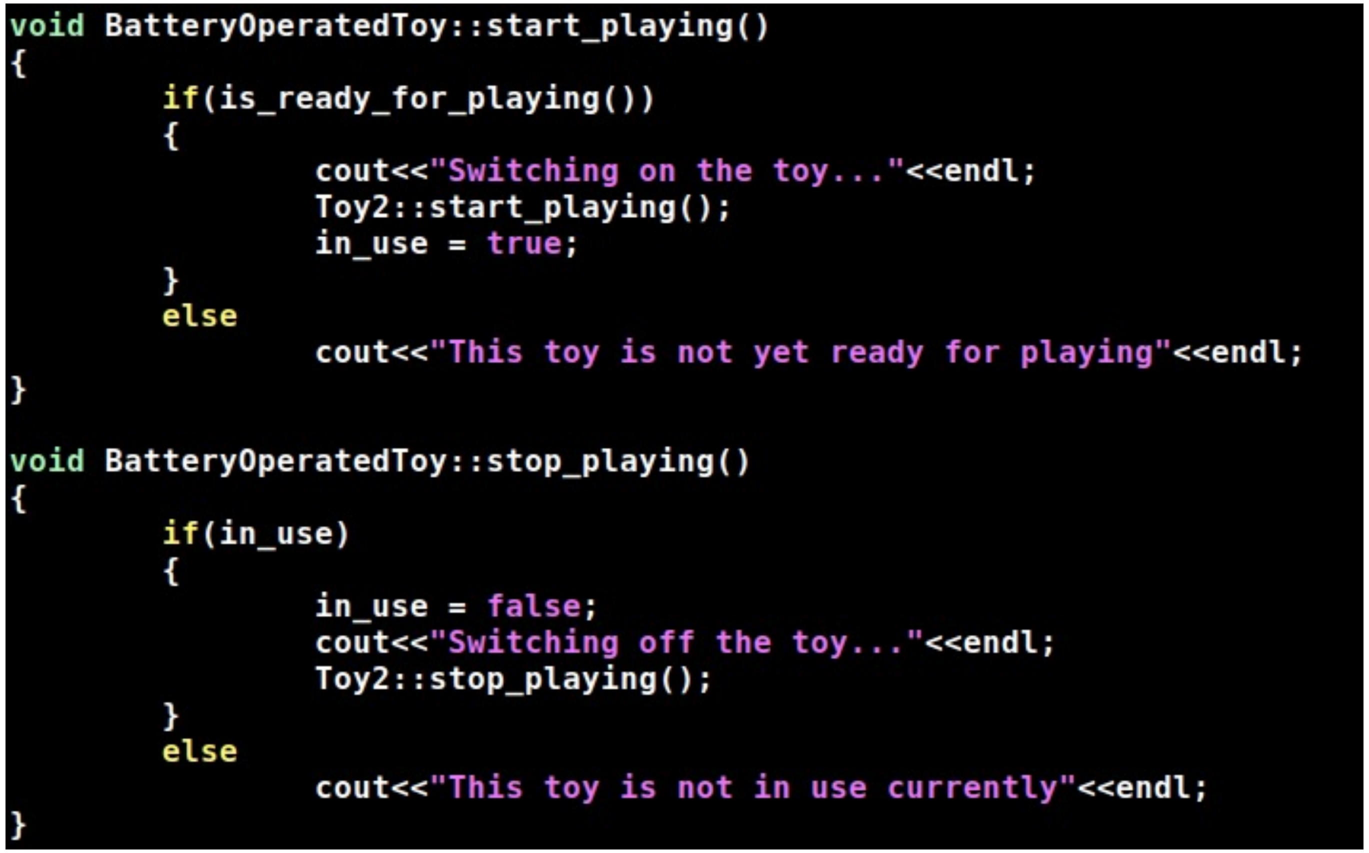
Virtual Functions



```
void Toy2::start_playing()
{
        cout<<"You are now playing with "<<name<<" with id "<<id<<endl;
}

void Toy2::stop_playing()
{
        cout<<"The "<<name<<" with id "<<id<<" is lying idle"<<endl;
}
```

These are the base class definitions for these methods …

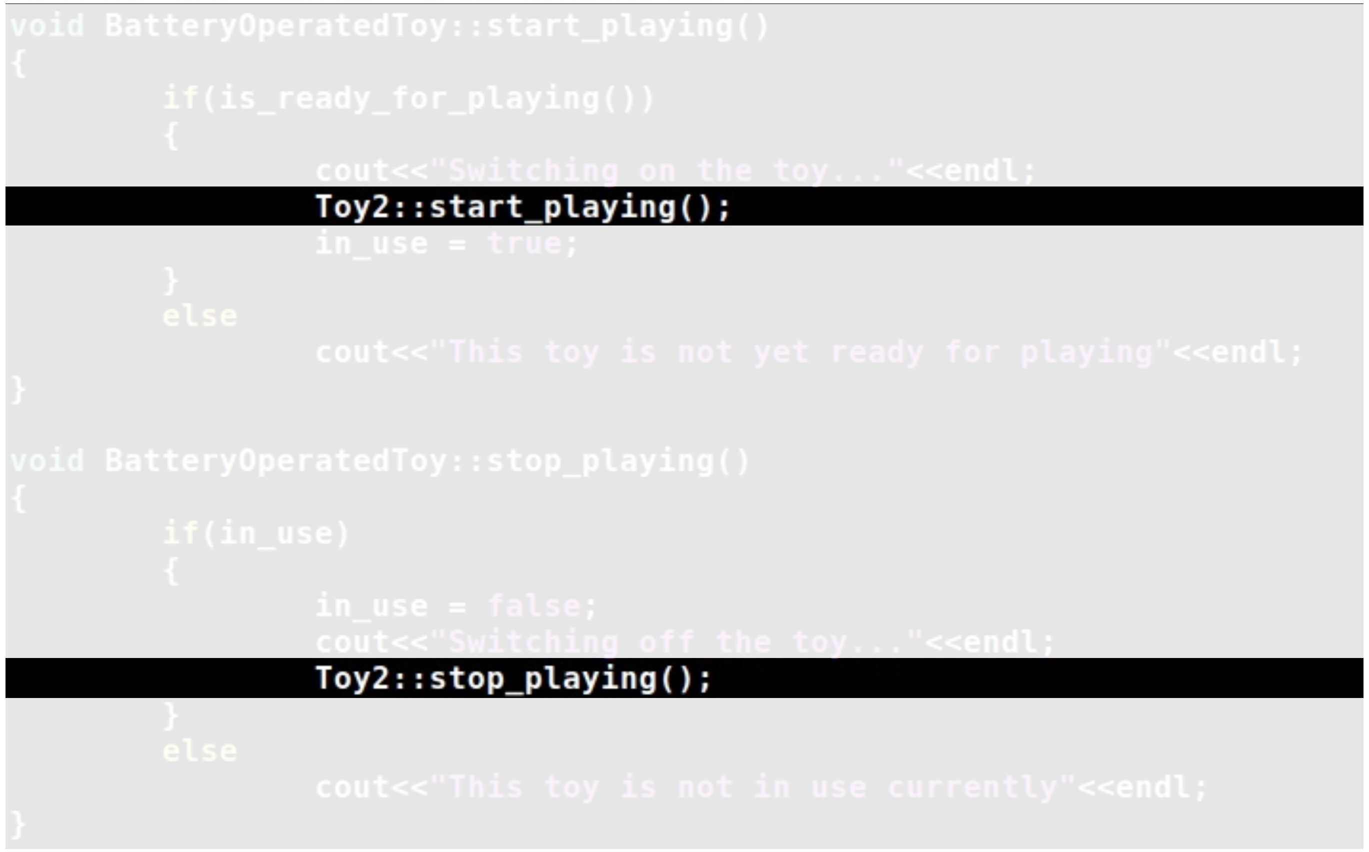
Virtual Functions



```
void BatteryOperatedToy::start_playing()
{
        if(is_ready_for_playing())
        {
                cout<<"Switching on the toy..."<<endl;
                Toy2::start_playing();
                in_use = true;
        }
        else
                cout<<"This toy is not yet ready for playing"<<endl;
}

void BatteryOperatedToy::stop_playing()
{
        if(in_use)
        {
                in_use = false;
                cout<<"Switching off the toy..."<<endl;
                Toy2::stop_playing();
        }
        else
                cout<<"This toy is not in use currently"<<endl;
}
```

… and these are the definitions for the methods in the derived class

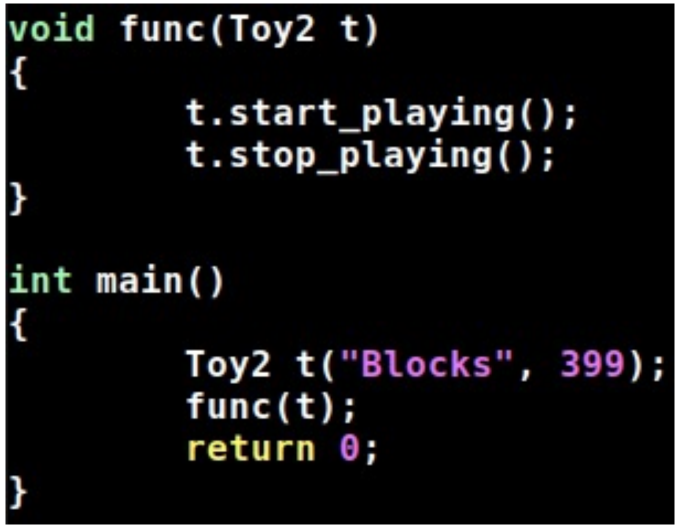
Virtual Functions



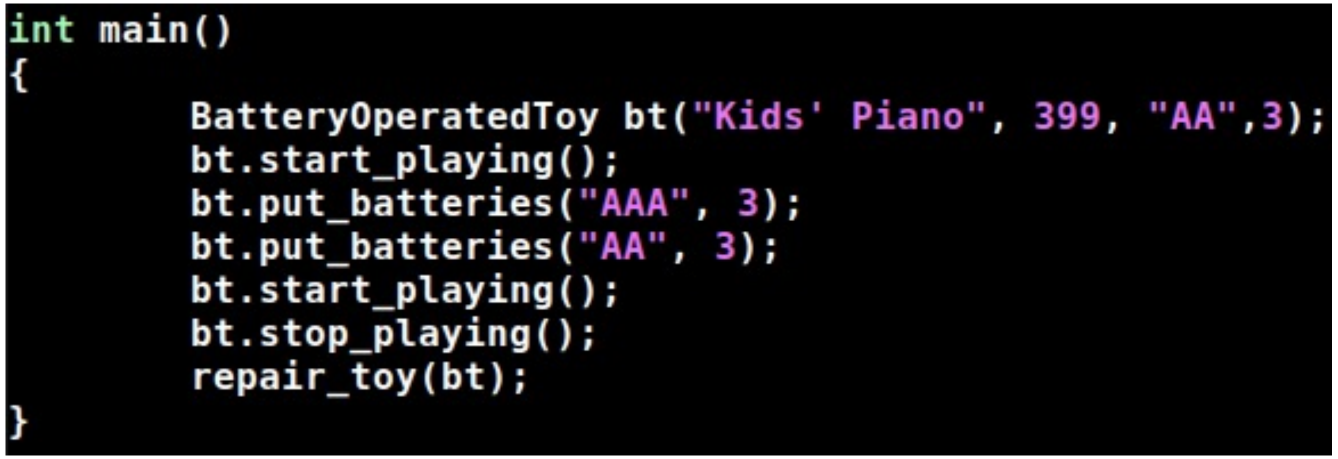
```
void BatteryOperatedToy::start_playing()
{
        if(is_ready_for_playing())
        {
                cout<<"Switching on the toy..."<<endl;
                Toy2::start_playing();
                in_use = true;
        }
        else
                cout<<"This toy is not yet ready for playing"<<endl;
}

void BatteryOperatedToy::stop_playing()
{
        if(in_use)
        {
                in_use = false;
                cout<<"Switching off the toy..."<<endl;
                Toy2::stop_playing();
        }
        else
                cout<<"This toy is not in use currently"<<endl;
}
```

The base class versions can also be accessed, using the scope resolution operator

Virtual Functions



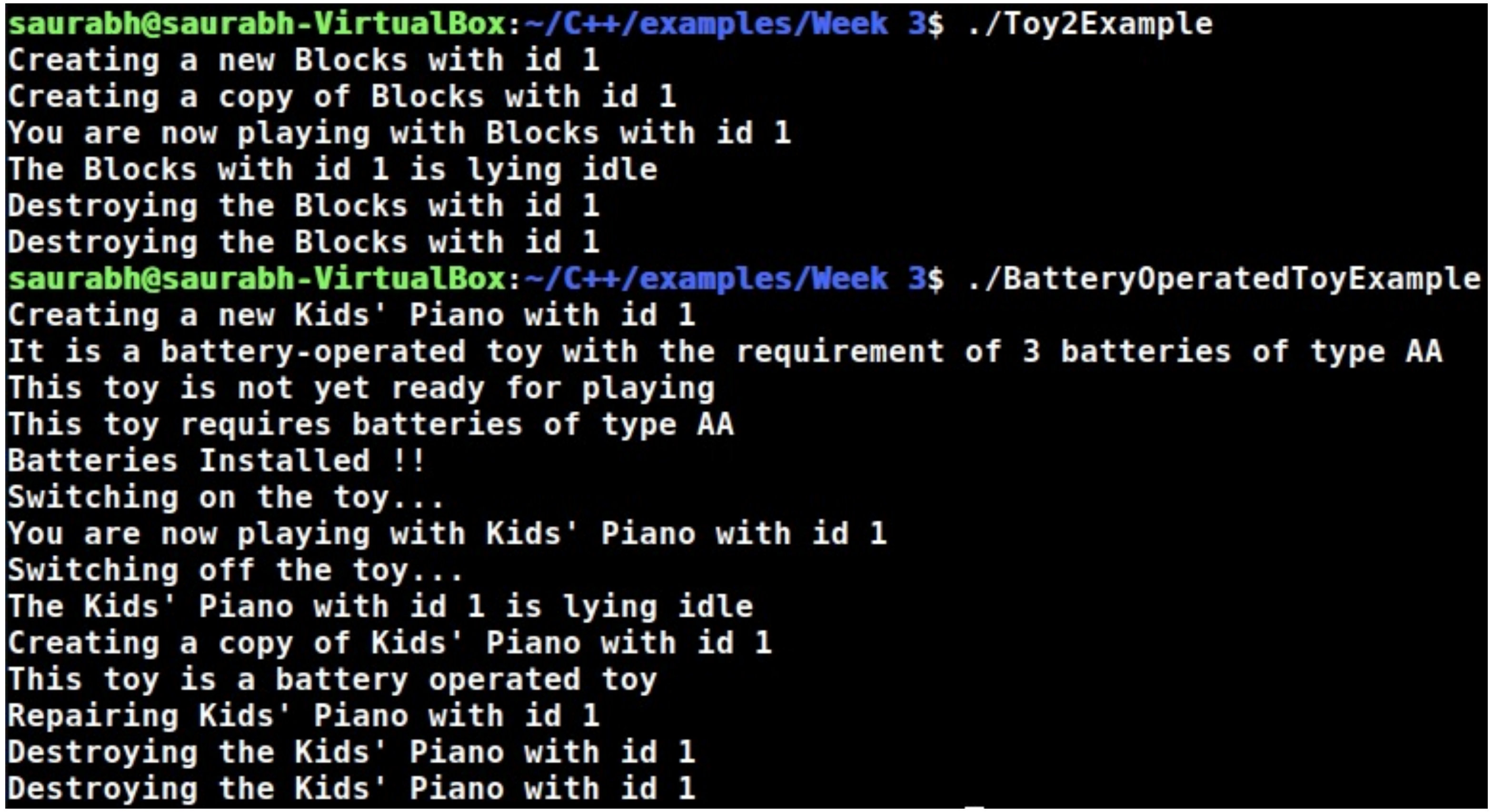
```
void func(Toy2 t)
{
        t.start_playing();
        t.stop_playing();
}

int main()
{
        Toy2 t("Blocks", 399);
        func(t);
        return 0;
}
```



```
int main()
{
        BatteryOperatedToy bt("Kids' Piano", 399, "AA",3);
        bt.start_playing();
        bt.put_batteries("AAA", 3);
        bt.put_batteries("AA", 3);
        bt.start_playing();
        bt.stop_playing();
        repair_toy(bt);
}
```

The methods can be invoked either with the base class object or derived class object

Virtual Functions



```
saurabh@saurabh-VirtualBox:~/C++/examples/Week 3$ ./Toy2Example
Creating a new Blocks with id 1
Creating a copy of Blocks with id 1
You are now playing with Blocks with id 1
The Blocks with id 1 is lying idle
Destroying the Blocks with id 1
Destroying the Blocks with id 1
saurabh@saurabh-VirtualBox:~/C++/examples/Week 3$ ./BatteryOperatedToyExample
Creating a new Kids' Piano with id 1
It is a battery-operated toy with the requirement of 3 batteries of type AA
This toy is not yet ready for playing
This toy requires batteries of type AA
Batteries Installed !!
Switching on the toy...
You are now playing with Kids' Piano with id 1
Switching off the toy...
The Kids' Piano with id 1 is lying idle
Creating a copy of Kids' Piano with id 1
This toy is a battery operated toy
Repairing Kids' Piano with id 1
Destroying the Kids' Piano with id 1
Destroying the Kids' Piano with id 1
```

You can observe how the output of the code changes when the same method is invoked with different types of objects

This is also an example of overriding – something we discussed before

We will revisit virtual functions after we have covered *Polymorphism*

Note: To build these executables, run the command `make clean all` in the example directory

Homework !!

Copy Constructors are “usually” not required to be defined explicitly

- It was certainly not required in our example, but in some cases it may be necessary
- Try to find out about these cases; may be give this article a read: https://www.geeksforgeeks.org/copy-constructor-in-cpp/

If you remember our discussion about “passing values by reference” in the previous semester …

- … we briefly discussed how the same term means slightly different things in C and C++
- Basically, C++ has an explicit way to create references – without the use of pointers
- You can, of course, also use the C way to pass parameters by reference, i.e. via pointers
- But using references are more popular (and usually convenient)
- Read more about these; you may start with these links: https://www.geeksforgeeks.org/references-in-c/ https://stackoverflow.com/questions/8627956/ways-of-passing-arguments-value-vs-reference-vs-pointer

On the last slide, try to find out why the destructor is being called twice?