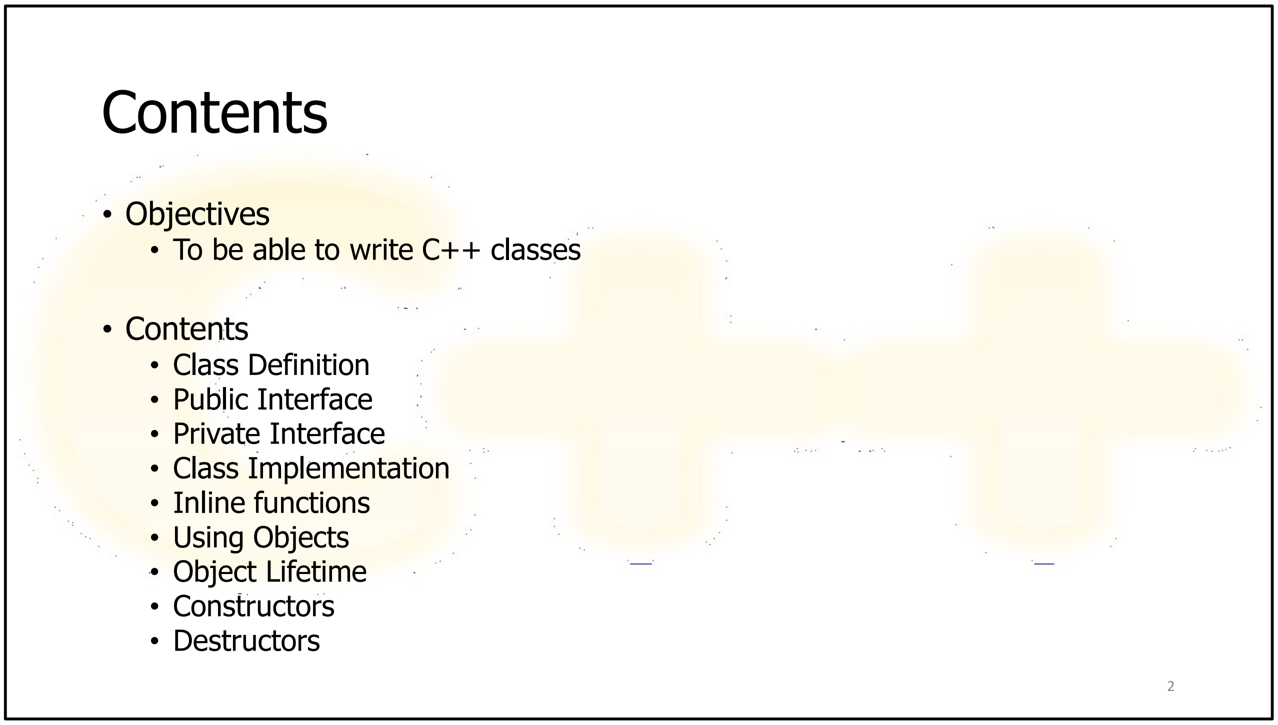
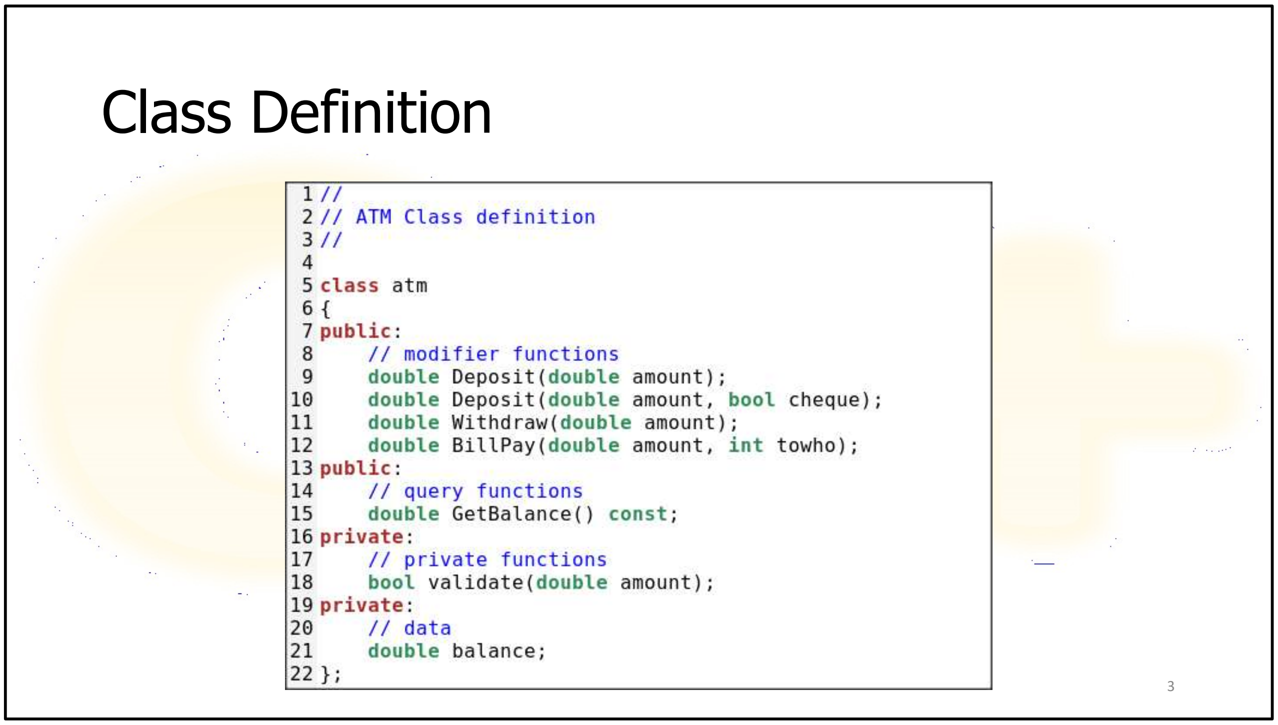
Classes & Objects



Contents

- Objectives
  - To be able to write C++ classes
- Contents
  - Class Definition
  - Public Interface
  - Private Interface
  - Class Implementation
  - Inline functions
  - Using Objects
  - Object Lifetime
  - Constructors
  - Destructors

2



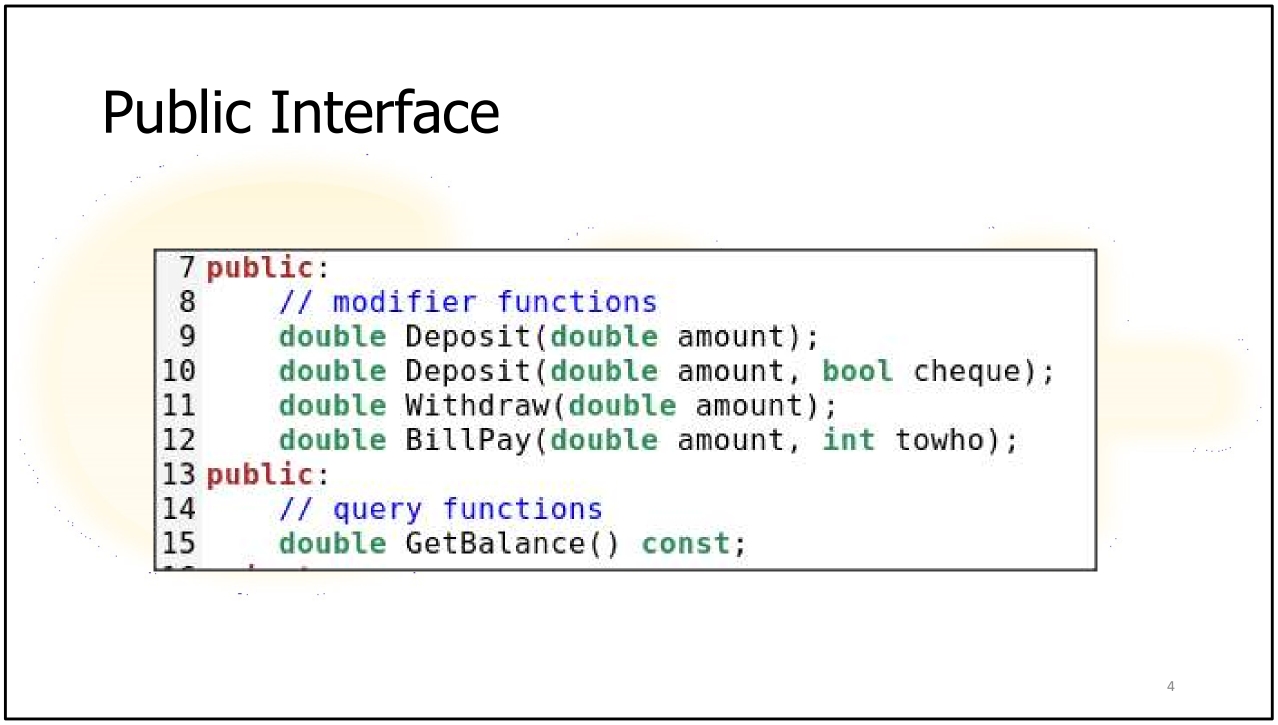
Class Definition

```cpp
//
// ATM Class definition
//

class atm
{
public:
    // modifier functions
    double Deposit(double amount);
    double Deposit(double amount, bool cheque);
    double Withdraw(double amount);
    double BillPay(double amount, int towho);
public:
    // query functions
    double GetBalance() const;
private:
    // private functions
    bool validate(double amount);
private:
    // data
    double balance;
};
```

3

Classes in C++ are normally declared in a header file. The keyword class is used to declare a class.   The class declaration usually contains function declarations & data declarations. The code for the functions is usually placed in a cpp file. The header & source file names should by convention be the same as the class name.

There may be several sections in a class definition. These are public, private and protected. The protected section will be discussed later in the course.



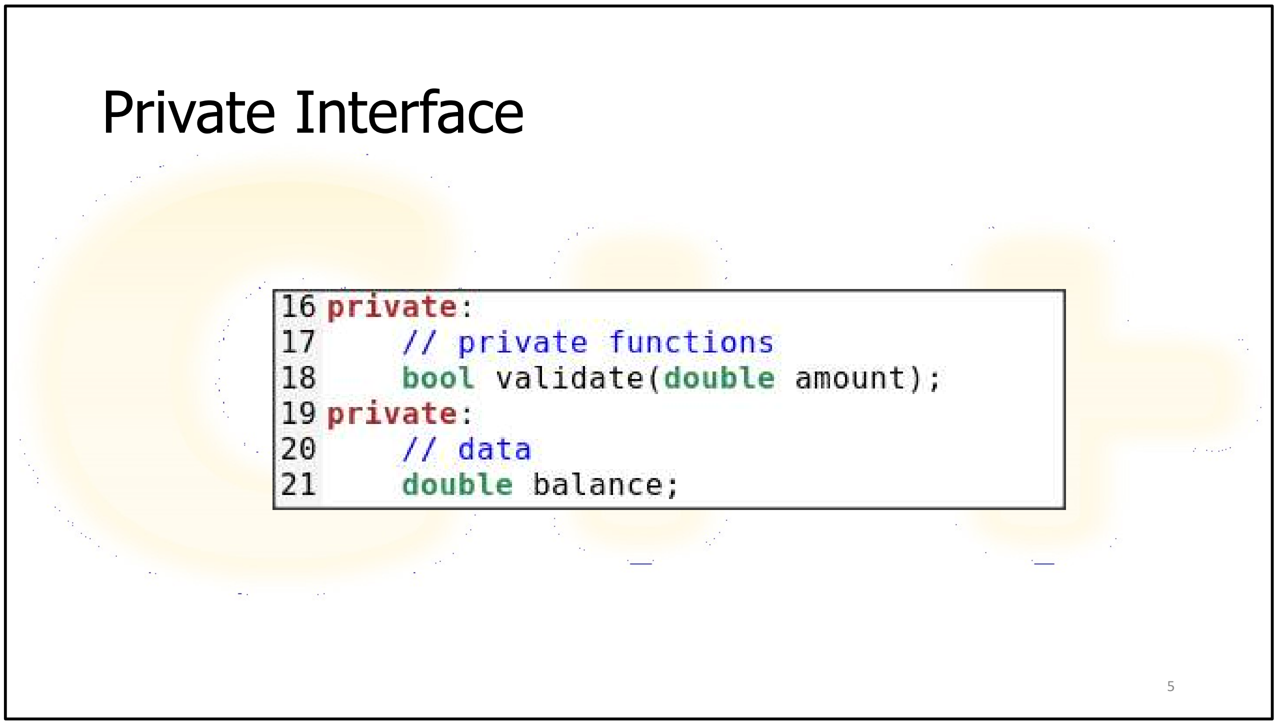
Public Interface

```
public:
    // modifier functions
    double Deposit(double amount);
    double Deposit(double amount, bool cheque);
    double Withdraw(double amount);
    double BillPay(double amount, int towho);
public:
    // query functions
    double GetBalance() const;
```

The public interface typically consists of three types of function prototype. Class modifier functions, class query functions and class constructors. Class modifier functions are those which modify data encapsulated within the class. Class query functions do not modify the data. Constructors perform initialisation for objects created from the class.

The public section is identified by using the public keyword followed by a colon. There may be more than one public section in a class definition.

This example also includes a member function – GetBalance that is declared as a const member function. const member functions are also called query functions and do not (cannot) change the state of the object. There are restrictions placed on const member functions. They cannot call modifier member functions. I.e. they can only call other const member functions.

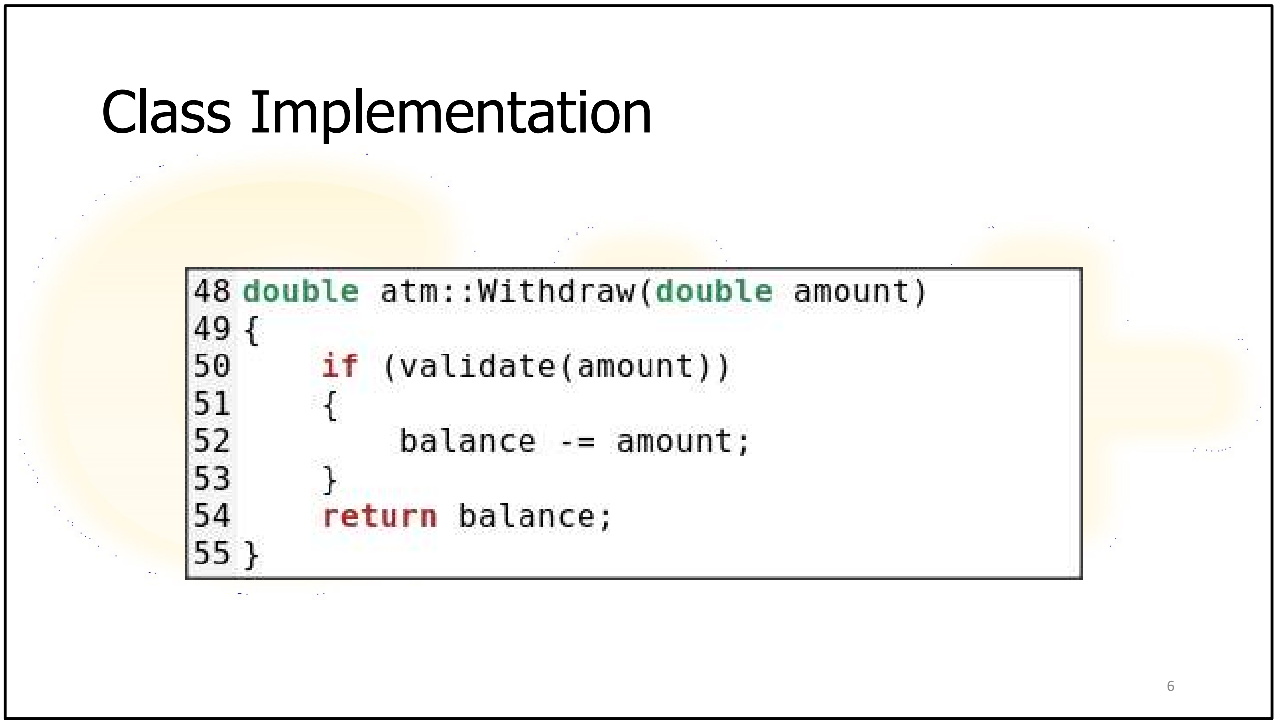


Private Interface

```
private:
    // private functions
    bool validate(double amount);
private:
    // data
    double balance;
```

5

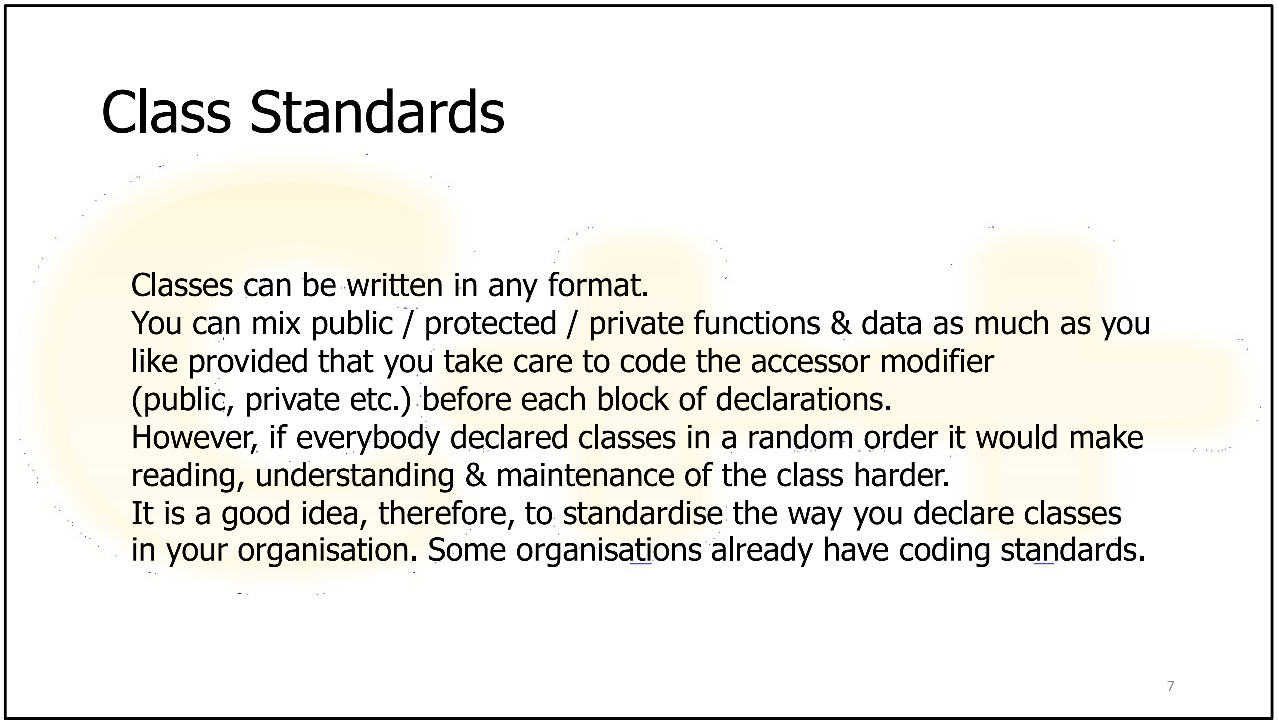
The private interface typically contains two types of definitions. It may contain private member function declarations such as the validate function for our ATM. It will also usually contain the declarations for encapsulated data items. As with the public section, you can have more than one private section in the class header.



Class Implementation

```
double atm::Withdraw(double amount)
{
    if (validate(amount))
    {
        balance -= amount;
    }
    return balance;
}
```

A member function definition is similar to any other function definition except that we must tell the compiler that it is part of one of our classes. This is done by explicitly declaring it’s scope using the scope resolution operator.

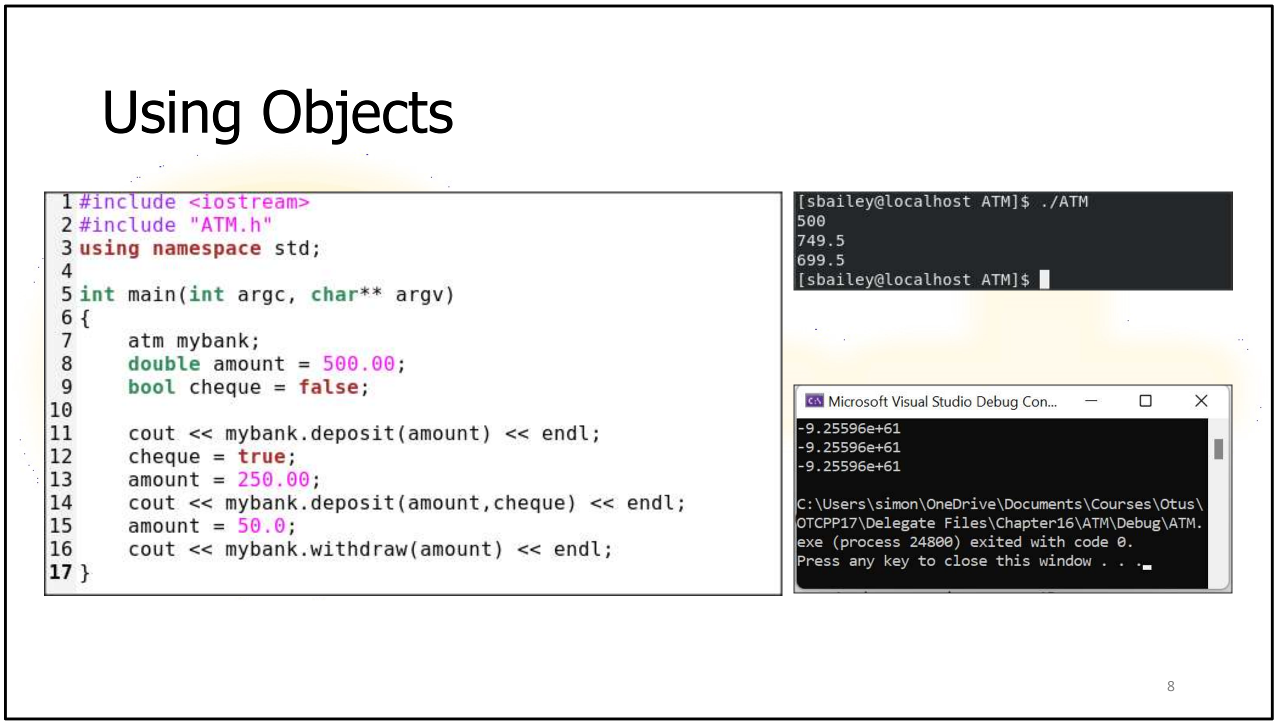


Class Standards

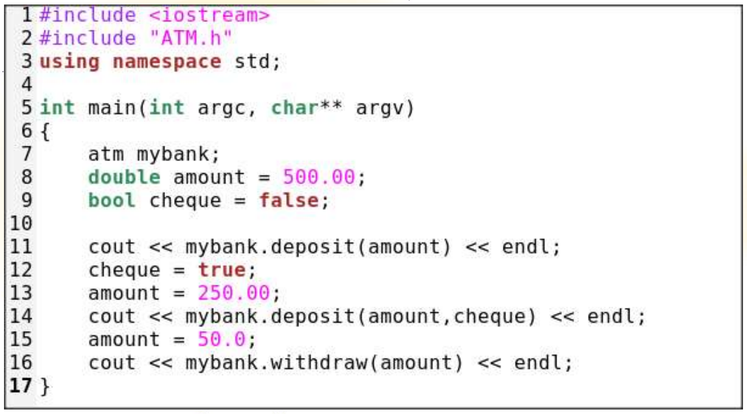
Classes can be written in any format.
You can mix public / protected / private functions & data as much as you like provided that you take care to code the accessor modifier (public, private etc.) before each block of declarations.
However, if everybody declared classes in a random order it would make reading, understanding & maintenance of the class harder.
It is a good idea, therefore, to standardise the way you declare classes in your organisation. Some organisations already have coding standards.

7

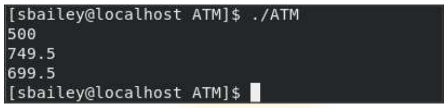
I have included a set of suggested programming standards at the end of the course document. Of course, following any standard is entirely optional & is not checked by the compiler.



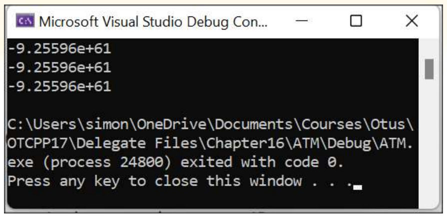
Using Objects



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

int main(int argc, char** argv)
{
    atm mybank;
    double amount = 500.00;
    bool cheque = false;

    cout << mybank.deposit(amount) << endl;
    cheque = true;
    amount = 250.00;
    cout << mybank.deposit(amount,cheque) << endl;
    amount = 50.0;
    cout << mybank.withdraw(amount) << endl;
}
```



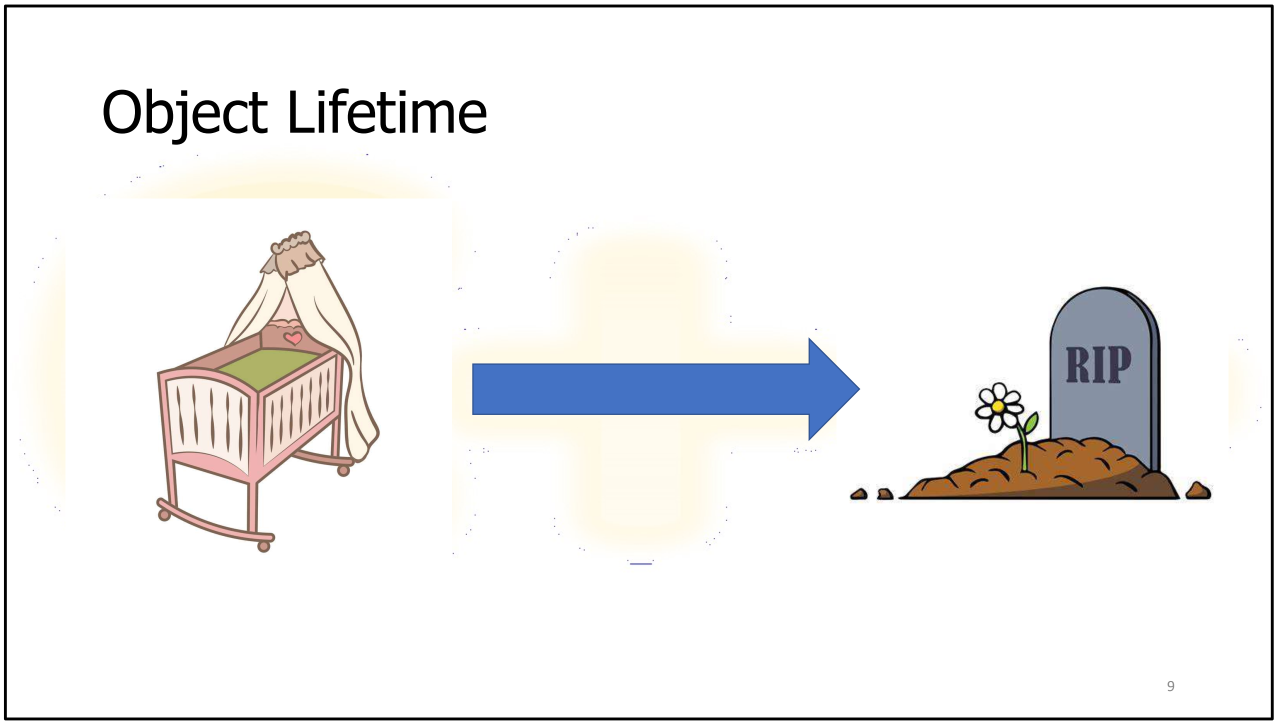
```
[sbailey@localhost ATM]$ ./ATM
500
749.5
699.5
[sbailey@localhost ATM]$
```



```
-9.25596e+61
-9.25596e+61
-9.25596e+61

C:\Users\simon\OneDrive\Documents\Courses\Otus\
OTCPP17\Delegate Files\Chapter16\ATM\Debug\ATM.
exe (process 24800) exited with code 0.
Press any key to close this window . . .
```

8

Two outputs are shown for this code. The first from Linux & the second from Visual Studio on Windows. The C++ language specification states that the initial value of variables is undefined. You cannot, therefore, rely on the compiler initialising the balance to 0. In fact, the Microsoft Visual Studio output demonstrates this.



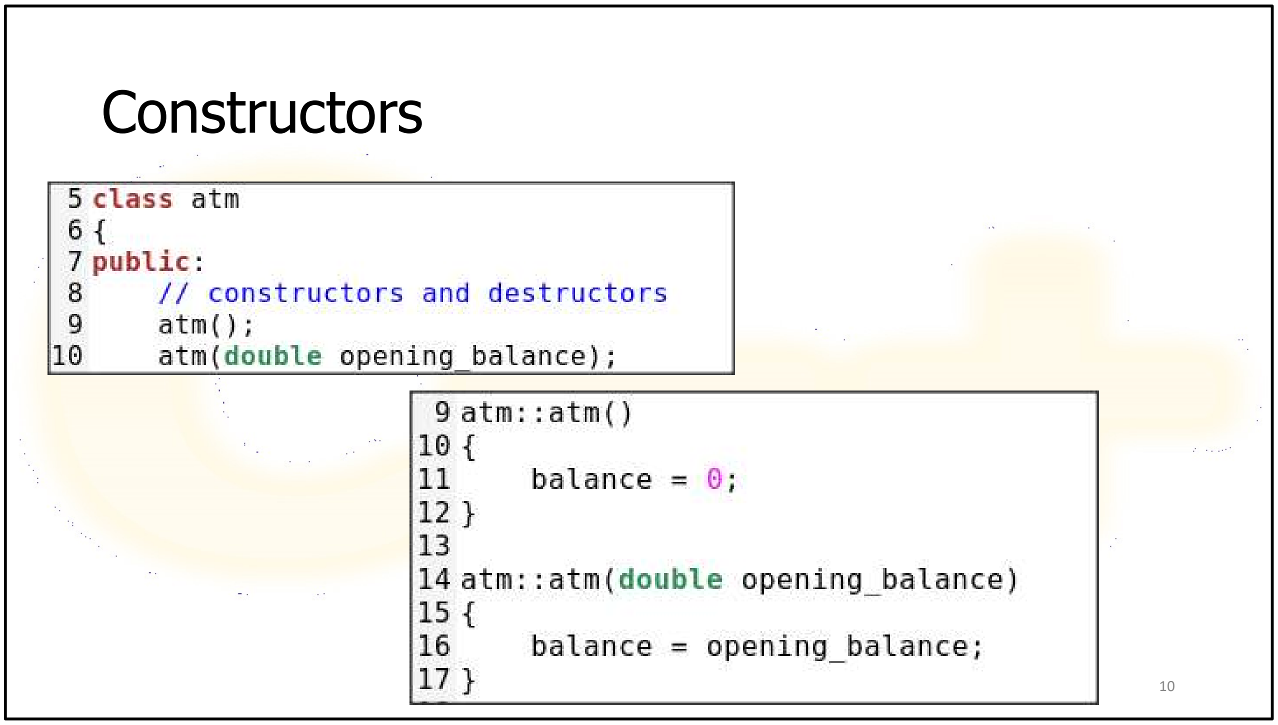
Object Lifetime

Object oriented programs are usually designed to reflect real life. Objects have a definite life. They are created, live for a time and then are destroyed. This is not too dissimilar to a human life. A human being is born, lives to carry out some purpose & sometime later dies. In real life, certain actions take place. For example, when you are born you are given a name and the birth is registered. When you die your death is registered & your assets disposed of in some way.

The OO paradigm is designed to enforce initialisation & disposal of objects. Although we have written no code to do so, when we created the mybank object in the previous slides initialisation did take place. The only problem is, it was default initialisation which did precisely nothing. Also, at the end of the program the mybank object was destroyed. A default destruction routine was also called even though we have not provided one ourselves.

Initialisation is done by a special member function of the class called a constructor. If you do not code your own constructor then the compiler will, under most circumstances, provide one for you.

Disposal of an object is performed by a special member function called a destructor. Again, the compiler will provide a default destructor if you do not provide one yourself.



Constructors

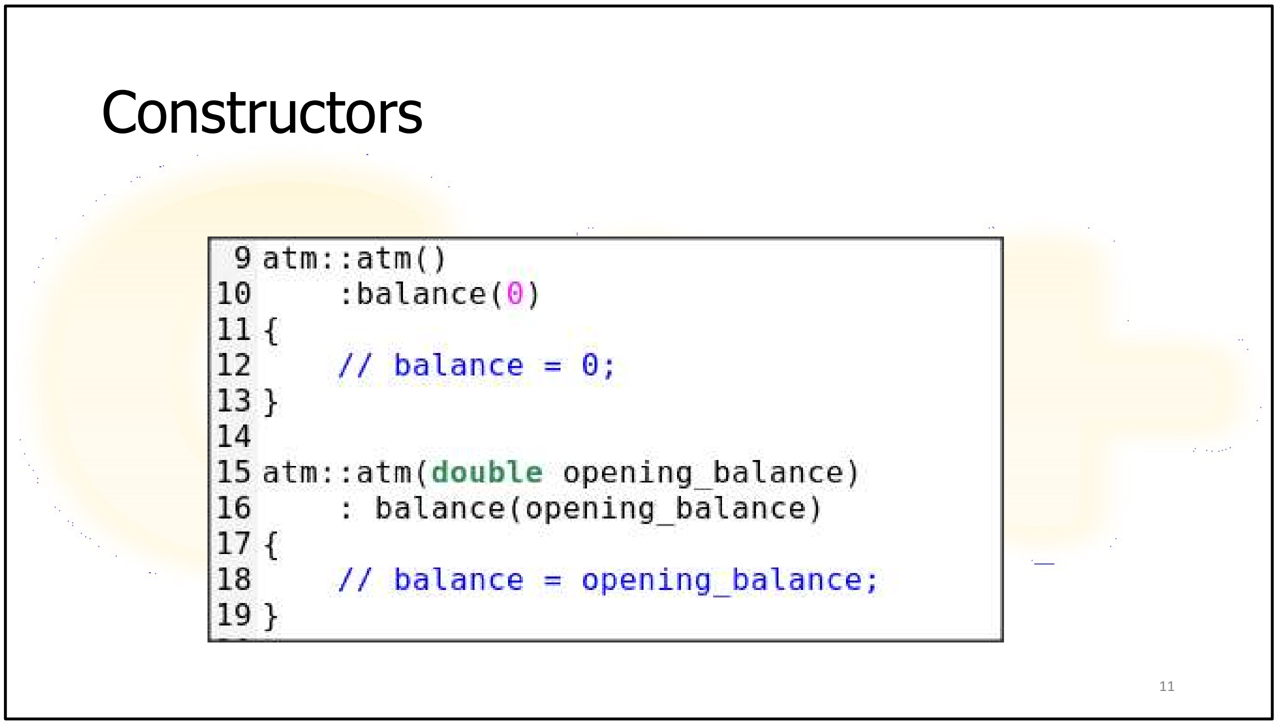
```cpp
class atm
{
public:
    // constructors and destructors
    atm();
    atm(double opening_balance);
```

```cpp
atm::atm()
{
    balance = 0;
}

atm::atm(double opening_balance)
{
    balance = opening_balance;
}
```

10

Constructors are called automatically when an object is created. They have a special format. A constructor for a class must have the same name as the class, and it has no return type, not even void.

Two constructors have been added to the class shown here. The first is known as the default constructor. The second is an overloaded constructor. If you provide any other constructor than the default constructor you should also provide the default too if required as the compiler will only provide the default if no other constructors are present. Note that constructor overloading follows the same rules as function overloading discussed earlier.



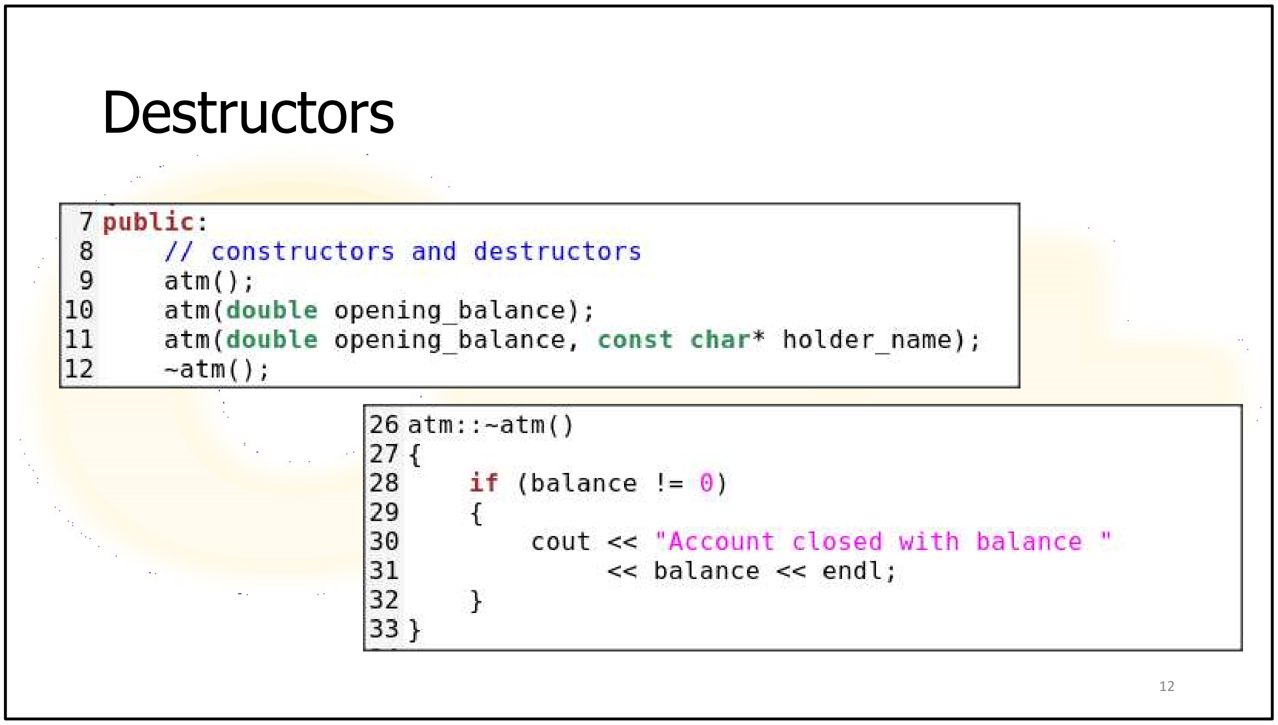
Constructors

```cpp
atm::atm()
    :balance(0)
{
    // balance = 0;
}

atm::atm(double opening_balance)
    : balance(opening_balance)
{
    // balance = opening_balance;
}
```

11

Some people would consider the code on the previous slide to be bad coding practice. They would prefer to see the code as shown on this slide using an initialiser.

Whilst many people prefer initializer lists over assignment, and most texts recommend it, initialization cannot be used in all cases. You cannot initialize all member variables using an initializer. For example, initialising pointer variables is a two step process & must be done in the constructor body. i.e. Allocate memory & then initialise the memory.



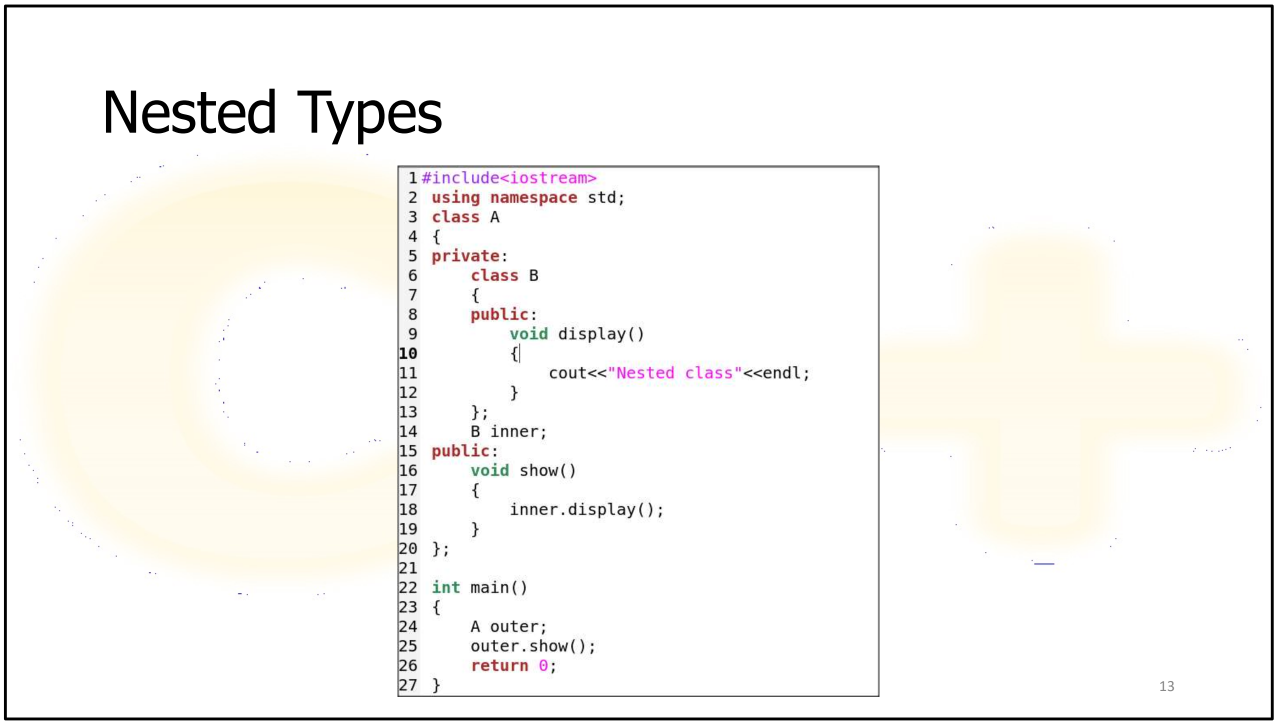
Destructors

```cpp
public:
    // constructors and destructors
    atm();
    atm(double opening_balance);
    atm(double opening_balance, const char* holder_name);
    ~atm();
```

```cpp
atm::~atm()
{
    if (balance != 0)
    {
        cout << "Account closed with balance "
             << balance << endl;
    }
}
```

12

A class may contain only one destructor member function. It may not be overloaded. It also has a special syntax. The name of the destructor function is the same name as the class preceded by the tilde (~) character. It has no arguments. Destructors are often used to de-allocate resources which may have been allocated in a constructor.



Nested Types

```
#include<iostream>
using namespace std;
class A
{
private:
    class B
    {
    public:
        void display()
        {
            cout<<"Nested class"<<endl;
        }
    };
    B inner;
public:
    void show()
    {
        inner.display();
    }
};

int main()
{
    A outer;
    outer.show();
    return 0;
}
```

13

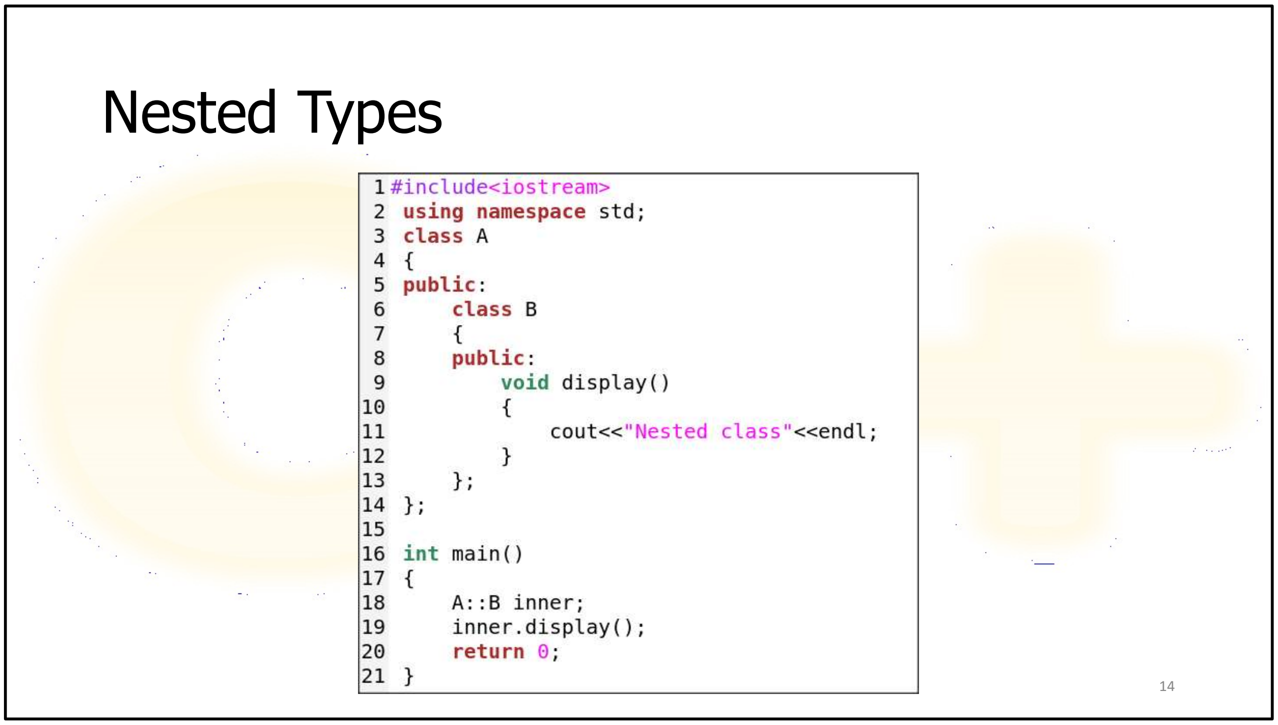
You can declare and define types within a class definition. Accessing these nested types is similar to using types declared within a namespace; the class name serves as the namespace name.

Prior to C++11 declarations in a nested class can use only type names, static members, and enumerators from the enclosing class.

From C++11, declarations in a nested class can use any members of the enclosing class, following the usual usage rules for the non-static members.

A nested class is a class defined inside another class. The class which contains the nested class is called as Enclosing class.

The nested class can be defined as private member of enclosing class. The object of enclosing class can be used to access the member function of the nested class:

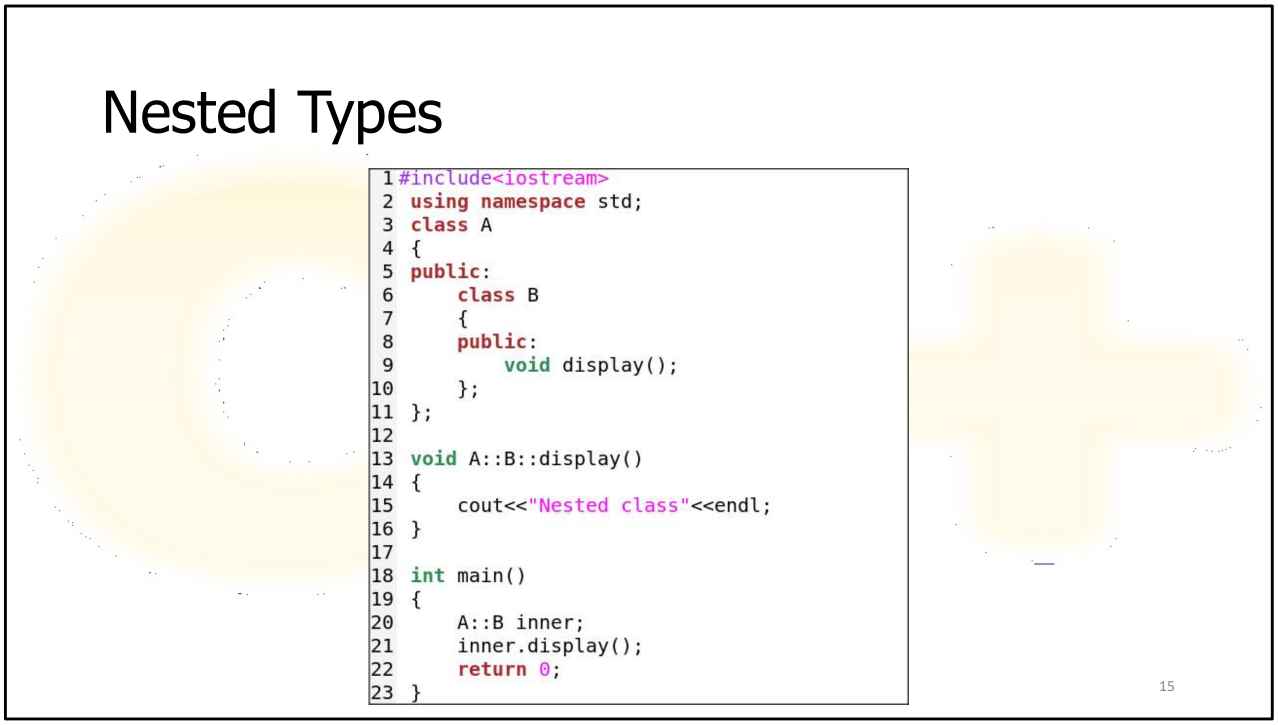


Nested Types

```
#include<iostream>
using namespace std;
class A
{
public:
    class B
    {
    public:
        void display()
        {
            cout<<"Nested class"<<endl;
        }
    };
};

int main()
{
    A::B inner;
    inner.display();
    return 0;
}
```

14

We can define the nested class as a public member of enclosing class. In this case, the public member function of nested class can be accessed from the object of enclosing class directly:

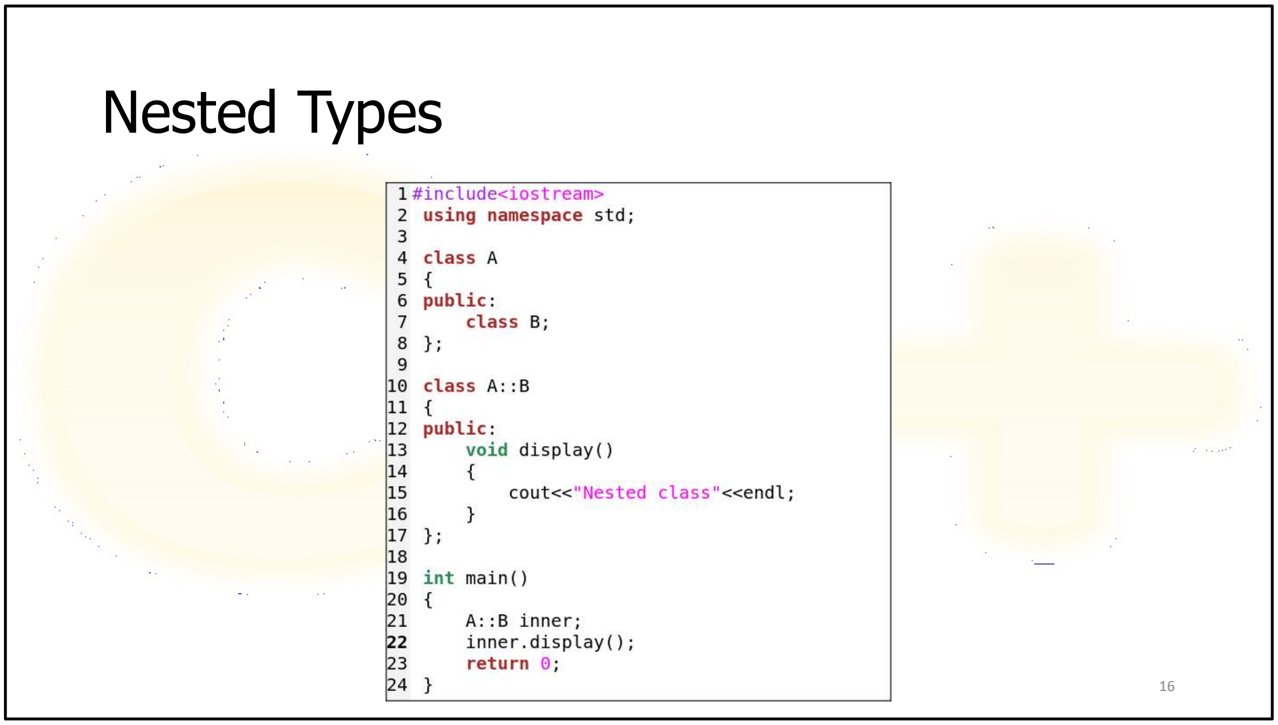


Nested Types

```
#include<iostream>
using namespace std;
class A
{
public:
    class B
    {
    public:
        void display();
    };
};

void A::B::display()
{
    cout<<"Nested class"<<endl;
}

int main()
{
    A::B inner;
    inner.display();
    return 0;
}
```

15

The member function of the nested class can be defined outside the enclosing class. Here we need to use the scope resolution operator:

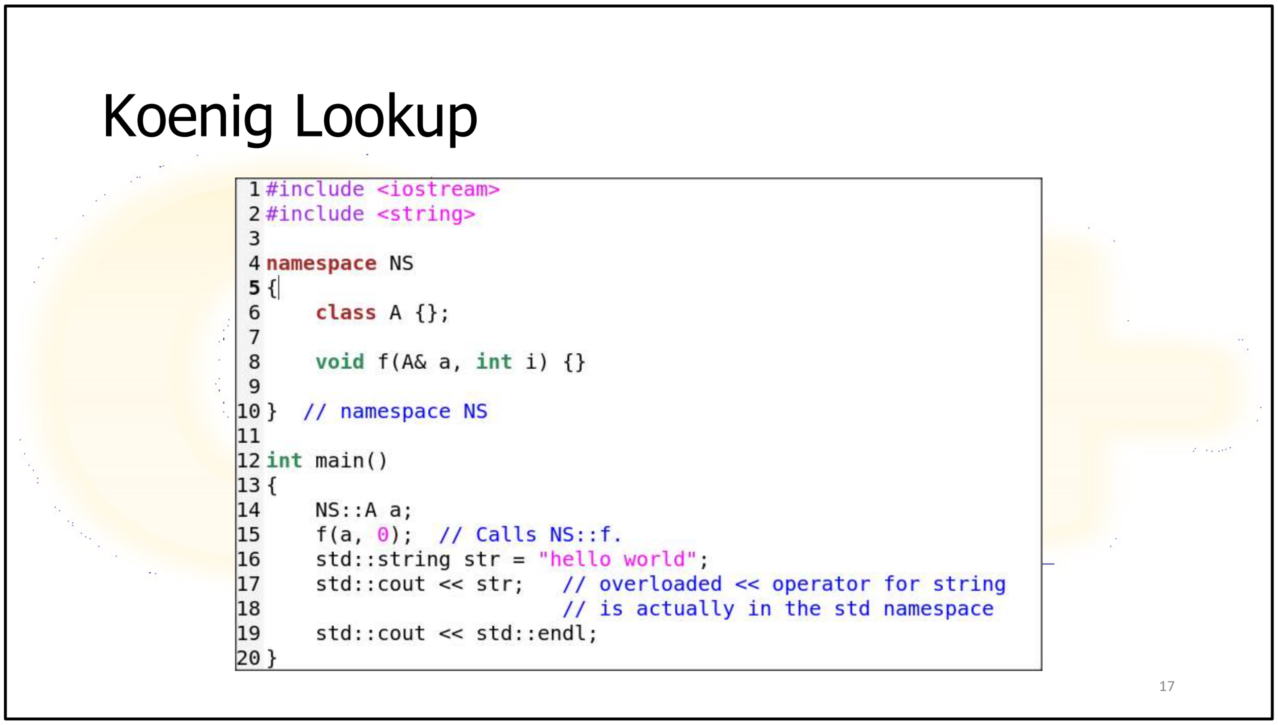


Nested Types

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

class A
{
public:
    class B;
};

class A::B
{
public:
    void display()
    {
        cout<<"Nested class"<<endl;
    }
};

int main()
{
    A::B inner;
    inner.display();
    return 0;
}
```

16

The nested class can be defined outside the enclosing class. Here we need to use forward declaration and the scope resolution operator.



Koenig Lookup

```
 1 #include <iostream>
 2 #include <string>
 3
 4 namespace NS
 5 {
 6     class A {};
 7
 8     void f(A& a, int i) {}
 9
10 }  // namespace NS
11
12 int main()
13 {
14     NS::A a;
15     f(a, 0);  // Calls NS::f.
16     std::string str = "hello world";
17     std::cout << str;   // overloaded << operator for string
18                         // is actually in the std namespace
19     std::cout << std::endl;
20 }
```

17

In the C++ programming language, argument-dependent lookup (ADL), or argument-dependent name lookup, applies to the lookup of an unqualified function name depending on the types of the arguments given to the function call. This behavior is also known as Koenig lookup, as it is often attributed to Andrew Koenig, though he is not its inventor.

In order to use an item from namespace NS as in line 14 above, we have to explicitly scope the item NS::. However, line 15 uses a function in the NS namespace but there is no NS:: before the function call. This does not cause a compiler error; it works because of Koenig Lookup. Similarly, on line 17 cout is used with the std::string data type. The overloaded << operator that deals with strings is actually coded within the std namespace. In theory the compiler would not find this version of the operator, but thanks to ADL it finds the correct operator & no further scoping is required.

While ADL makes it practical for functions defined outside of a class to behave as if they were part of the interface of that class, it makes namespaces less strict and so can require the use of fully qualified names when they would not otherwise be needed. For example, the C++ standard library makes extensive use of unqualified calls to std::swap to swap two values. The idea is that then one can define an own version of swap in one's own namespace and it will be used within the standard library algorithms.

Exercise