SWINBURNE UNIVERSITY OF TECHNOLOGY

COS30008 Data Structures and Patterns

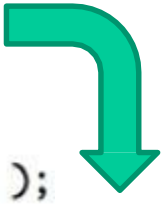
Lecture 3 Introduction to C++ Part 3

Inheritance

- A mechanism for specialization
- A mechanism for reuse
- Fundamental to supporting polymorphism

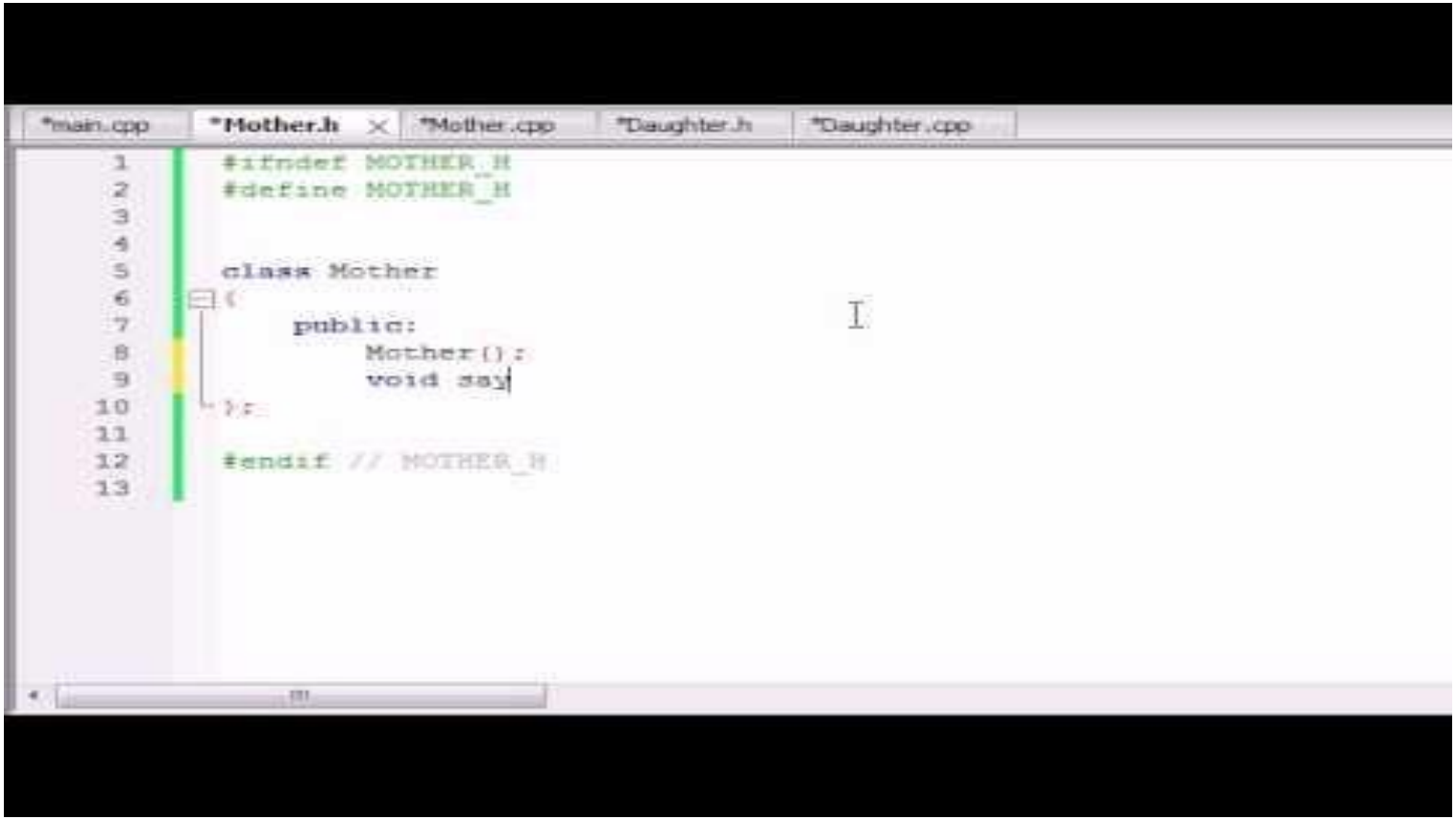
Example

```
4  class Account
5  {
6  private:
7    double fBalance;
8
9  public:
10   Account( double aBalance );
11   virtual ~Account() {}
12
13   void deposit( double aAmount );
14   virtual void withdraw( double aAmount );
15   double getBalance();
16 };
```



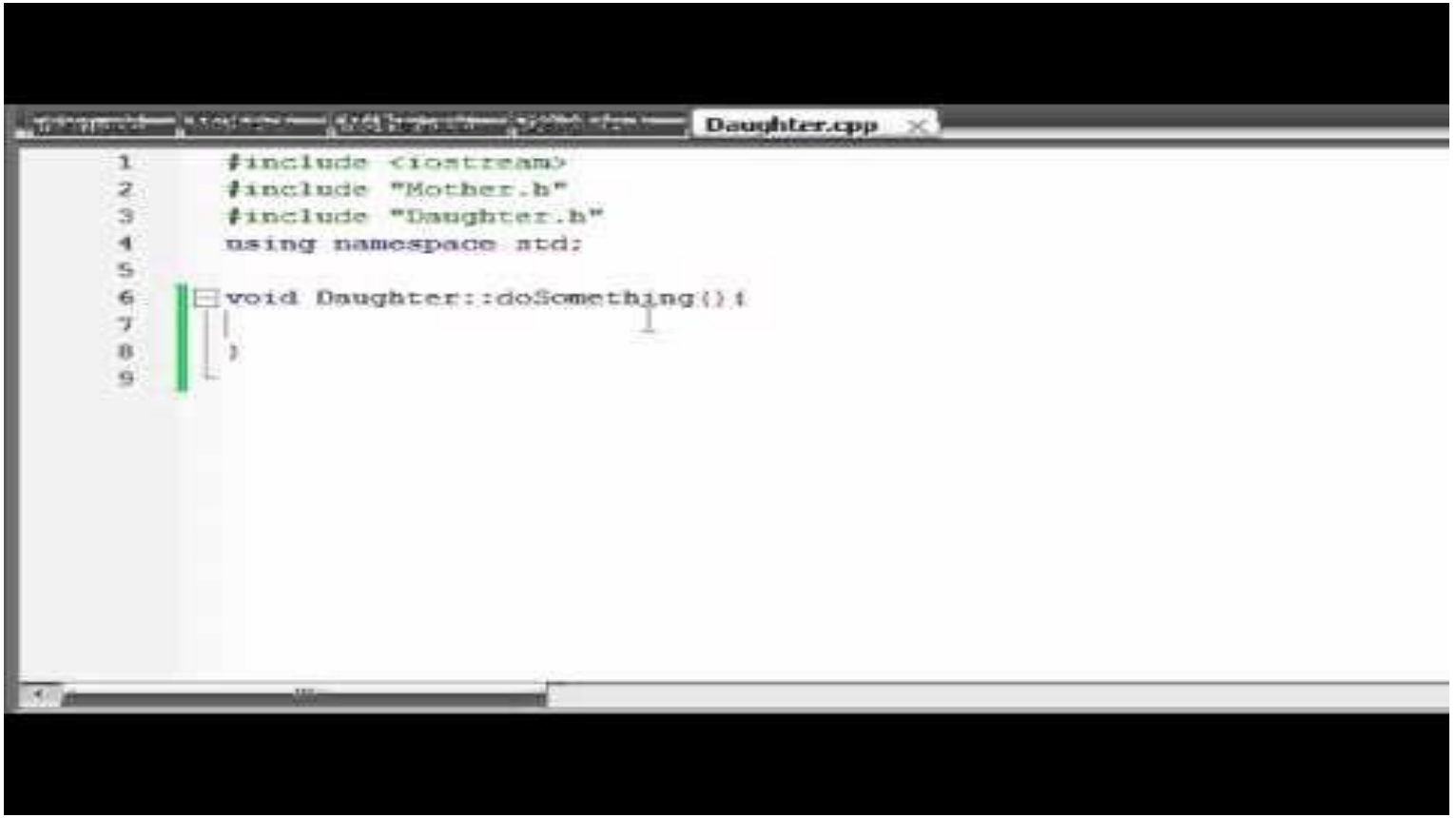
```
18 class BankAccount: public Account
19 {
20 private:
21   double fInterestRate;
22
23 public:
24   BankAccount( double aRate );
25   ~BankAccount() {}
26
27   virtual void withdraw( double aAmount );
28   void addInterest();
29   void chargeFee( double aAmount );
30 };
```

Class Inheritance



URL: https://www.youtube.com/watch?v=gq2Igdc-OSI

Protected members



URL: https://www.youtube.com/watch?v=DHAAy4GJ684

Access Levels for Class Inheritance

- public:
  - Public members in the base class remain public.
  - Protected members in the base class remain protected.
- protected:
  - Public and protected members in the base class are protected in the derived class.
- private:
  - Public and protected members in the base class become private in the derived class.

https://www.tutorialspoint.com/cplusplus/cpp_class_access_modifiers.htm

Constructors and Inheritance

- Whenever an object of a derived class is instantiated, multiple constructors are called so that each class in the inheritance chain can initialize itself.
- The constructor for each class in the inheritance chain is called beginning with the base class at the top of the inheritance chain and ending with the most recent derived class.

Base Class Initializer

Default Value

```
3  Account::Account( double aBalance = 0.0 ) : fBalance(aBalance)
4  {}
```

```
21 BankAccount::BankAccount( double aRate ) : Account(), fInterestRate(aRate)
22 {}
```

Direct super class

Base Class Initializers

- If a base class does not have a default constructor, the derived class must provide a base class initializer for it.
- Base class initializers frequently appear alongside member initializers, which use similar syntax.
- If more than one argument is required by a base class constructor, the arguments are separated by comma.
- Reference members need to be initialized using a member initializer.

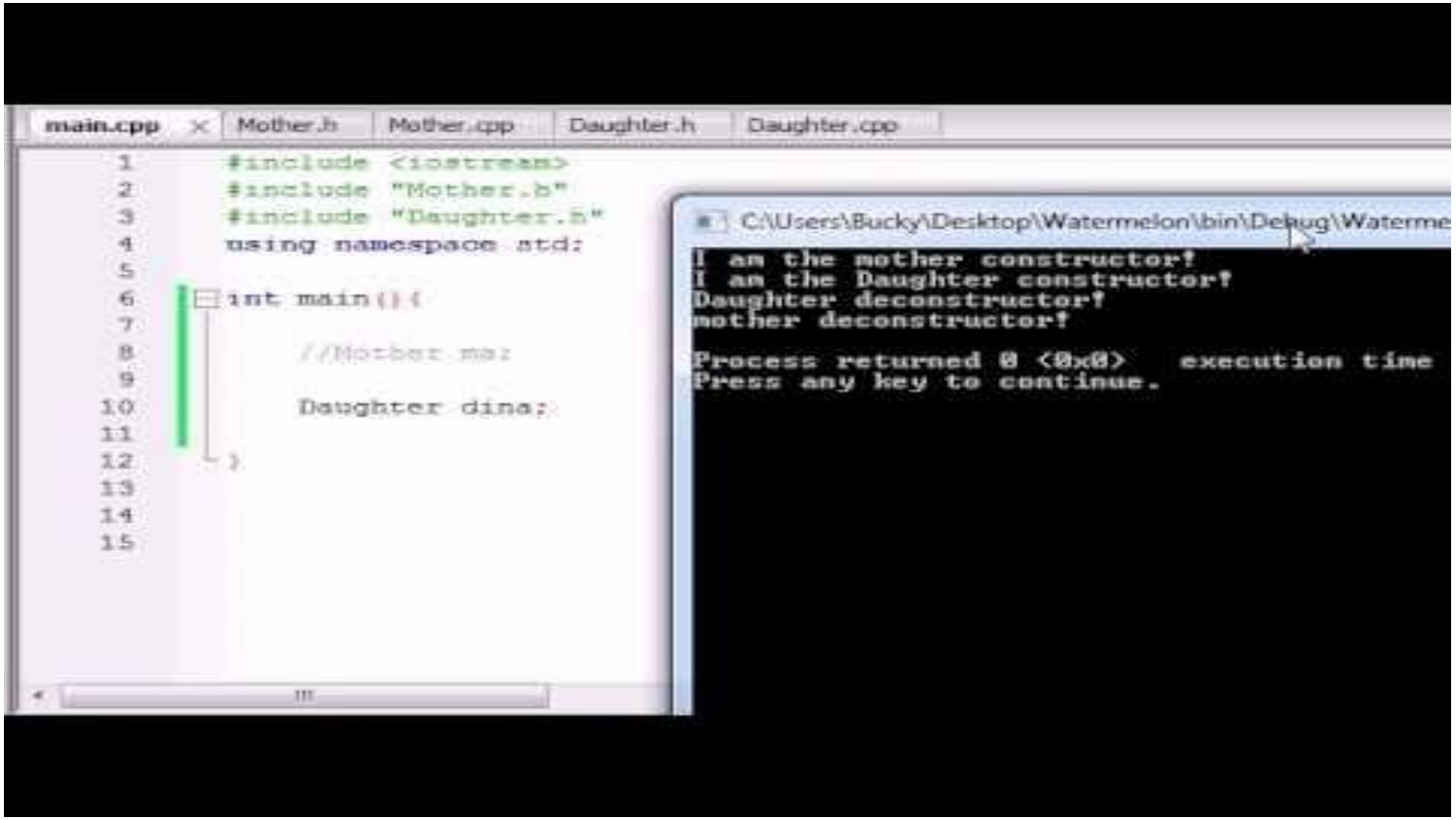
Destructors

- A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.
- A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.
  - Do not accept arguments.
  - Cannot specify any return type (including void).
  - Cannot return a value using the return statement.

Destructors and Inheritance

- Whenever an object of a derived class is destroyed, the destructor for each class in the inheritance chain, if defined, is called.
- The destructor for each class in the inheritance chain is called beginning with the most recent derived class and ending with the base class at the top of the inheritance chain.

Derived class constructors and destructors



URL: https://www.youtube.com/watch?v=Z_vJEKU9WTg

Virtual Destructor

```
class Base
{
 public:
 ~Base() {cout << "Base Destructor\t"; }
};

class Derived:public Base
{
 public:
 ~Derived() { cout<< "Derived Destructor"; }
};

int main()
{
 Base* b = new Derived;      //Upcasting
 delete b;
}
```

Output :

Base Destructor

- **delete b** will only call the Base class destructor, which is undesirable because, then the object of Derived class **remains undestructed**, because its destructor is never called. This results in memory leak.

Virtual Destructor

```
1  class Base
2  {
3   public:
4   virtual ~Base() {cout << "Base Destructor\t"; }
5  };
6
7  class Derived:public Base
8  {
9   public:
10  ~Derived() { cout<< "Derived Destructor"; }
11 };
12
13 int main()
14 {
15  Base* b = new Derived;      //Upcasting
16  delete b;
17 }
```

Output :

Derived Destructor
Base Destructor

- When we have Virtual destructor inside the base class, then first Derived class's destructor is called and then Base class's destructor is called, which is the desired behaviour.

Example of Destructors

```
4  class Account
5  {
6  private:
7    double fBalance;
8
9  public:
10   Account( double aBalance );
11   virtual ~Account() {}
12
13   void deposit( double aAmount );
14   virtual void withdraw( double aAmount );
15   double getBalance();
16 };
```

```
18 class BankAccount: public Account
19 {
20 private:
21   double fInterestRate;
22
23 public:
24   BankAccount( double aRate );
25   ~BankAccount() {}
26
27   virtual void withdraw( double aAmount );
28   void addInterest();
29   void chargeFee( double aAmount );
30 };
```

Virtual Destructors

- When deleting an object using a base class pointer of reference, it is essential that the destructors for each of the classes in the inheritance chain get a chance to run:

```
BankAccount *BAptr;
Account *Aptr;
BAptr = new BankAccount( 2.25 );

Aptr = BAptr; … // upper casting
…
delete Aptr;
```

Virtual Destructor

```
class Account
{
public:
    virtual ~Account() { ... }
};
```

Not declaring a destructor virtual is a common source of memory leaks in C++ code.

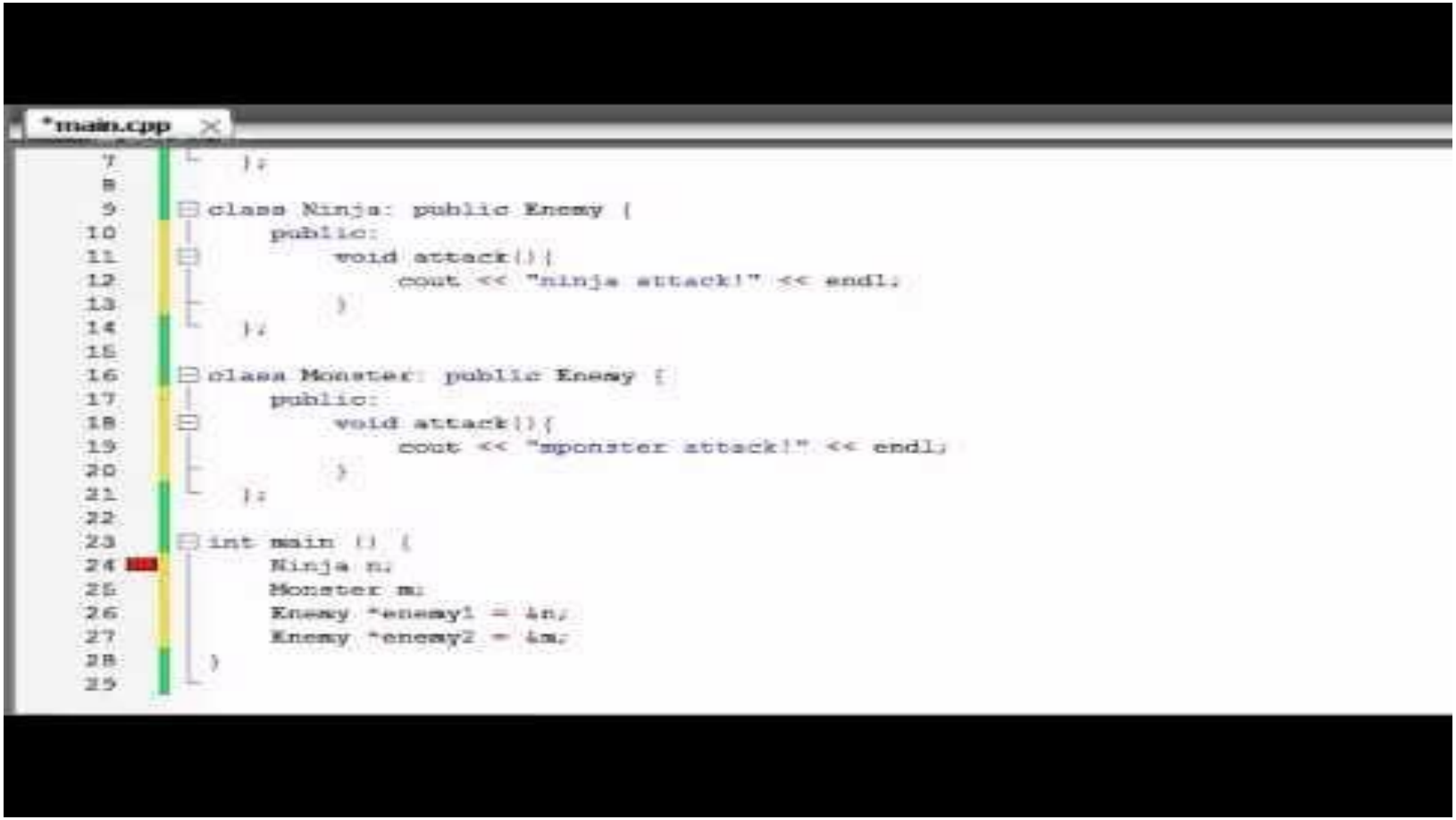
Polymorphism

- The word **polymorphism** means having many forms. Typically, **polymorphism** occurs when there is a hierarchy of classes and they are related by inheritance.
- **C++ polymorphism** means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.
- Polymorphism allows derived classes to use the same functions but have different outcomes. This is done by **function overriding**.

Polymorphism

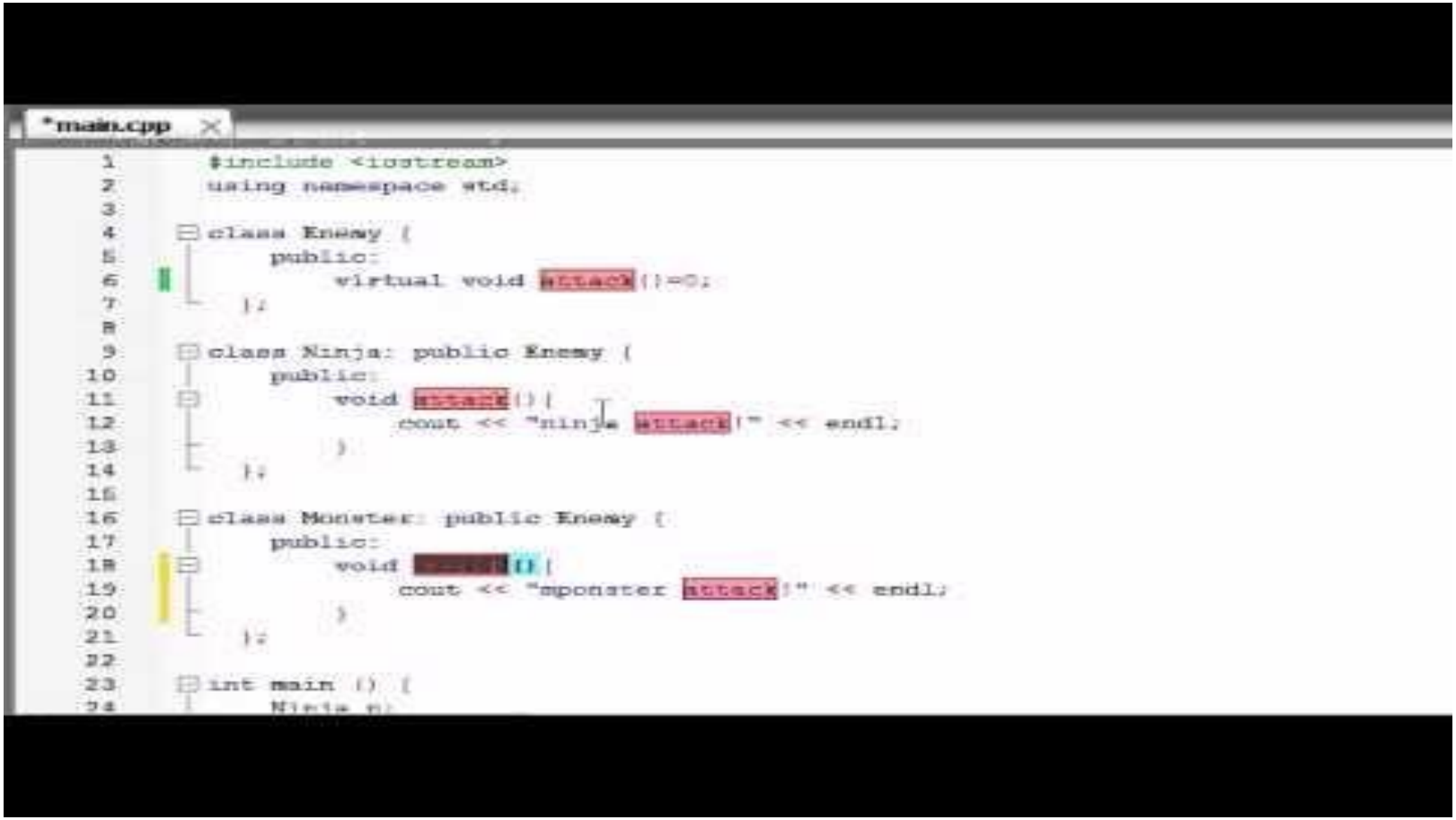
URL: https://www.youtube.com/watch?v=R_PPA9eejDw

Virtual Functions



URL: https://www.youtube.com/watch?v=DudHooleNVg

Abstract Classes and Pure Virtual Functions



URL: https://www.youtube.com/watch?v=ndz3EHpFEZc

Virtual Member Functions

- To give a member function from a base class new behavior in a derived class, one **overrides** it.
- To allow a member function in a base class to be overridden, one must declare the member function virtual.
- When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class' version of the function.

Virtual Example:

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

class base
{
public:
    virtual void print ()
    { cout<< "print base class" <<endl; }

    void show ()
    { cout<< "show base class" <<endl; }
};

class derived:public base
{
public:
    void print ()
    { cout<< "print derived class" <<endl; }

    void show ()
    { cout<< "show derived class" <<endl; }
};

int main()
{
    base *bptr;
    derived d;
    bptr = &d;

    //virtual function, binded at runtime
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();
}
```

Output:

print derived class
show base class

The difference between using virtual method and just overriding the methods

- A method can be defined in a base class and overridden in the derived class whether it is a virtual method or not.
- If the overridden method is called directly from the derived class, the derived class' method will be used.
- However, when a **base class pointer points to a derived class object**, it will call the **base class' method** if the it is **not a virtual method**, but will use the **derived class' method** if it is **virtual**.
- Declaring a method 'virtual' means that C++ will use mechanisms to support polymorphism and check to see if there is a more derived version of the method when you call via a base class pointer.

More details here:
https://stackoverflow.com/questions/11067975/overriding-non-virtual-methods

Virtual withdraw Method

- Back to the account example

```
4  class Account
5  {
6  private:
7    double fBalance;
8
9  public:
10   Account( double aBalance );
11   virtual ~Account() {}
12
13   void deposit( double aAmount );
14   virtual void withdraw( double aAmount );
15   double getBalance();
16 };
```

Accounts.h

```
18 class BankAccount: public Account
19 {
20 private:
21   double fInterestRate;
22
23 public:
24   BankAccount( double aRate );
25   ~BankAccount() {}
26
27   virtual void withdraw( double aAmount );
28   void addInterest();
29   void chargeFee( double aAmount );
30 };
```

Overriding the withdraw Method

```
24  void BankAccount::withdraw( double aAmount )
25  {
26      if ( (this->getBalance() - aAmount) > 0.0 )
27          Account::withdraw( aAmount );
28  }
29
```

Call inherited method via **`this->`**

Call overridden

SWINBURNE UNIVERSITY OF TECHNOLOGY

Calling a Virtual Method

```
1  #include "Accounts.h"
2
3  int main()
4  {
5      BankAccount lBankAccount( 2.25 );
6      Account* Aptr = &lBankAccount;
7
8      Aptr->withdraw( 50.0 );
9
10     return 0;
11 }
```

Calls withdraw in derived class, i.e. BankAccount

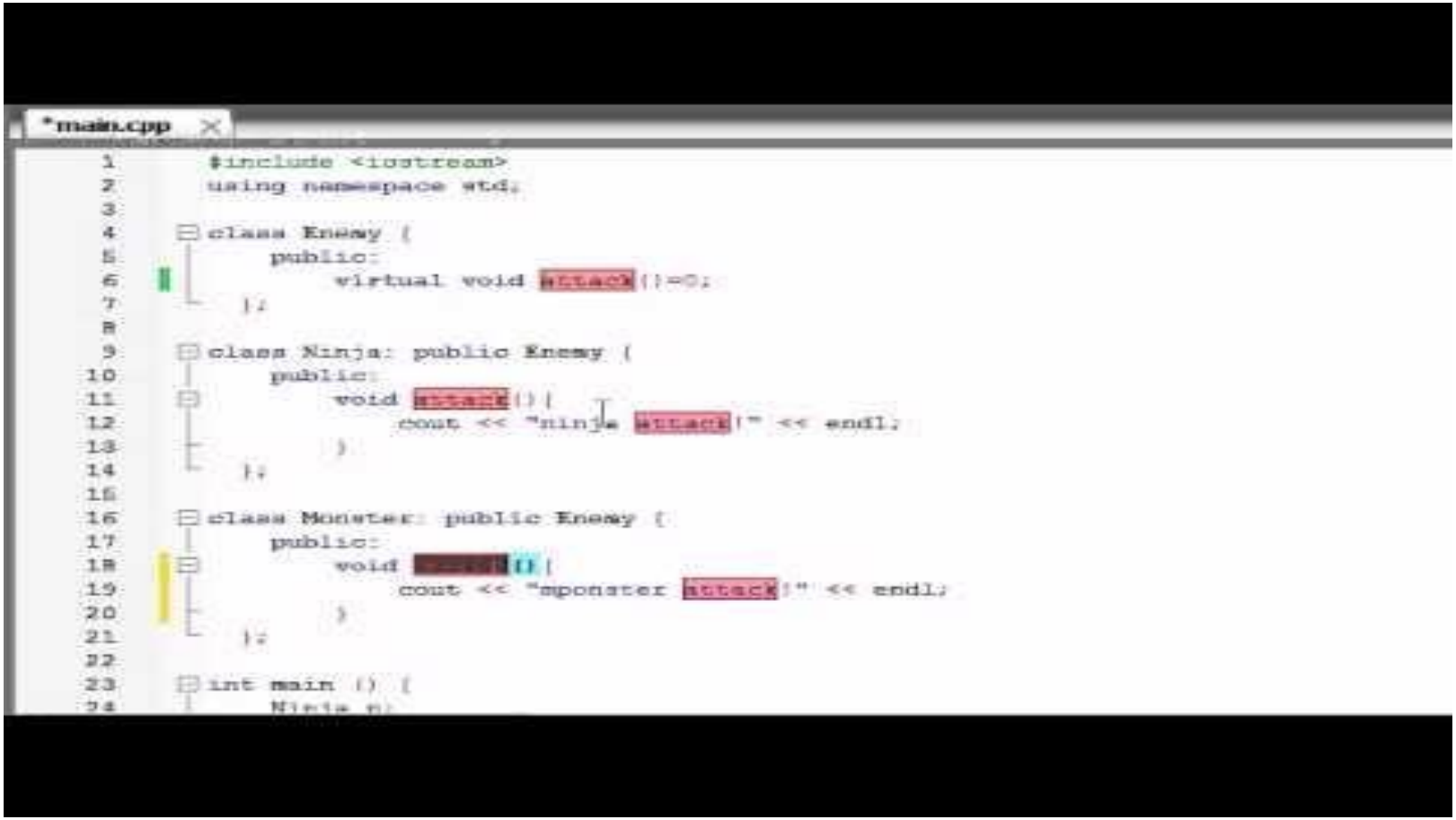
Facts About Virtual Members

- Parameter and result types must match to properly override a virtual member function.
- If one declares a non-virtual member function virtual in a derived class, the new member function hides the inherited member function.
- You can declare a private member function virtual to enable polymorphism within the scope of the declaring class

Facts About Virtual Members

- Constructors cannot be virtual.
- Once a member function has been declared virtual, it remains virtual.
- Declaring a member function virtual does not require that this function must be overridden in derived classes, except when the member function is declared **pure virtual**.

Abstract Classes and Pure Virtual Functions



URL: https://www.youtube.com/watch?v=ndz3EHpFEZc

About pure virtual function

- A pure virtual function is a virtual function whose declaration ends in “=0”, it's just syntax!

```
class Base {
   // ...
   virtual void f() = 0;
   // ...
}
```

- A pure virtual function implicitly makes the class it is defined as *abstract*

example of pure virtual

```
1  #include<iostream>
2  using namespace std;
3
4  class Base
5  {
6     int x;
7  public:
8      virtual void fun()=0;
9      int getX() { return x; }
10 };
11
12 // This class inherits from Base and implements fun()
13 class Derived: public Base
14 {
15     int y;
16 public:
17     void fun() { cout << "fun() called"; }
18 };
19
20 int main(void)
21 {
22     Derived d;
23     //Base e;
24     d.fun();
25     return 0;
26 }
```

why ”=0”?
it is just syntax!

The curious =0 syntax was chosen ... because at the time I saw no chance of getting a new keyword accepted. - *Bjarne Stroustrup*

if this line is enabled, the compiler will return some error like:
error C2259: 'Base' : cannot instantiate
abstract class
due to following members:
'void Base::fun(void)' : is abstract

Output :

fun() called

What is an abstract class?

- The purpose of an **abstract class** (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit.
- Abstract classes cannot be used to instantiate objects and serves only as an **interface**. Attempting to instantiate an object of an abstract class causes a compilation error.
- Thus, if a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the ABC.
- Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error.

Read more:
https://www.tutorialspoint.com/cplusplus/cpp_interfaces.htm

SWINBURNE UNIVERSITY OF TECHNOLOGY

Some other features of C++

SWINBURNE UNIVERSITY OF TECHNOLOGY

Enumerations

- Enumerations provide a mechanism for defining constants and grouping them into sets of integral types.
- An enumeration is defined using the **enum** keyword, followed by an optional enumeration name, and a comma-separated list of enumerators enclosed in braces.

```
enum CardSuit { Blank, Club, Diamond, Heart, Spade };
```

- Enumerators are `const` values. If not otherwise specified, the first enumerator equals 0, whereas the others are implicitly assigned the increment of its predecessor.

C++ Enums

URL: https://www.youtube.com/watch?v=b_OpPRWG-T8

Type Definitions

- Type definitions using the **typedef** keyword let us introduce a synonym for a type:

```
typedef char byte;
```

- Typedefs are commonly used for three purposes:
  - To hide the implementation of a given type.
  - To streamline complex type definitions making them easier to understand, and
  - To allow a single type to be used in different contexts under different names.
    - Type definitions establish a nominal equivalence between types.

Const Qualifier

- What are the problems with

```
for ( int index = 0; index < 128; index++ ) { … }
```

- We can do better

```
for ( int index = 0; index < BufferSize; index++ ) { … }
```

- Defining a const object:

```
const int BufferSize = 128; // initialized at compile time
```

- Unlike macro definitions, const objects have an address!

```
#define BUF_SIZE 128 // macro definition
const int BufferSize = BUF_SIZE; // initialized at compile
time
```

Constant Reference Parameters

- C++ uses call-by-value as default parameter passing mechanism.

```
void Assign( int aPar, int aVal ) { aPar = aVal; }
Assign( val, 3 ); // val unchanged
```

- A reference parameter yields call-by-reference:

```
void AssignR( int& aPar, int aVal ) { aPar = aVal; }
AssignR( val, 3 ); // val is set to 3
```

- A const reference parameter yields call-by-reference, but the value of the parameter is read-only:

```
void AssignCR( const int& aPar, int aVal ) { aPar =
aVal; } // error
```

Constant References

- A reference introduces a new name for an object:

```
int BlockSize = 512;

int& BufferSize = BlockSize;

// BufferSize is an alias
```

- A constant reference yields a new name for a constant object:

```
const int FixedBlockSize = 512;

const int& FixedBufferSize = FixedBlockSize;
```

- A constant reference defines an alias to an object.

Reference-based Objects

- Reference-based objects require pointer variables and an explicit new and delete:

```
Card* AceOfDiamond = new Card( Diamond, 14 );
Card* TestCard = new Card( Diamond, 14 );


if ( TestCard == AceOfDiamond )
cout << "The test card is " << TestCard->getName() << endl;


delete AceOfDiamond;
delete TestCard;
```

Reference Data Members

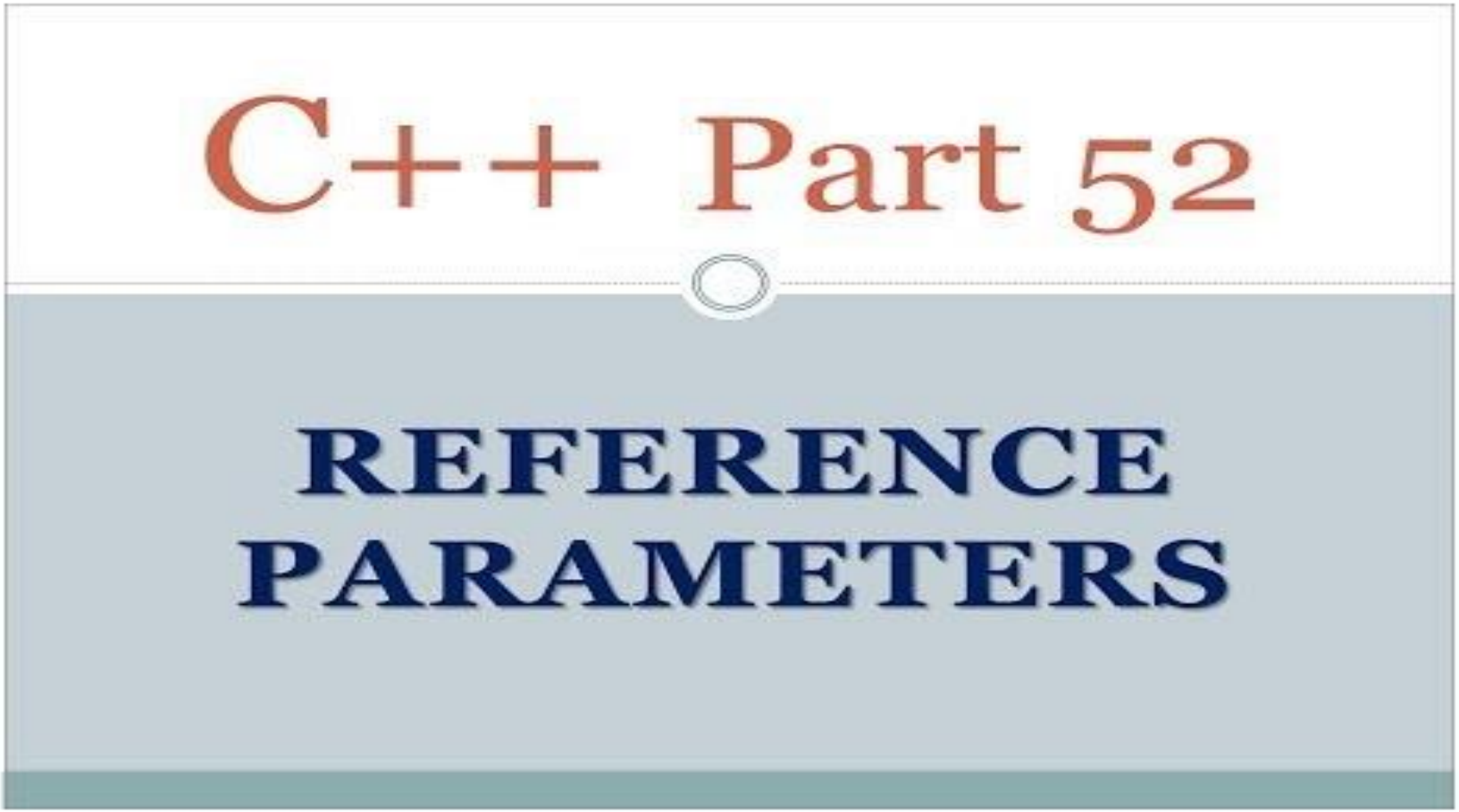
- Constructor initializers are optional, but there are cases in which they are required.
- Reference data members require a constructor initializer:

```
class RefMember

{
private:
OtherClass& fRef;
public:
RefMember( OtherClass& aRef ) : fRef(aRef) { ... }
};
```

Reference data members must be initialized before the contructor body is entered!

Reference Parameters



URL: https://www.youtube.com/watch?v=X9-a-MiMwEI

End of Introduction to C++ Part 3

This is the last part of Intro to C++ :D We'll be looking into Data Structures next week (Finally..)