Lecture X — Inheritance and derived classes

Suppose we want to write a class of footballers who play for Arsenal

A footballer who plays for Arsenal is a Premiership footballer, and so each object in this new class is also an object in the class of Premiership footballers

The "is a" relationship allows us to derive the class of footballers who play for Arsenal from the class of Premiership footballers

Each object in the class of footballers who play for Arsenal inherits many properties from the class of Premiership footballers

Inheritance allows us to re–use code

Premiership footballers is the base class and Arsenal footballers is the derived class

The header file for the class of Arsenal footballers, `ArsenalFootballer.hpp` may be written

```
#include "PremiershipFootballer.hpp"
class ArsenalFootballer: public PremiershipFootballer
{
public:
  ArsenalFootballer();
};
```

The word "public" in the first line of code on the previous slide has the effect that

- `public` members of `PremiershipFootballer` are public members of `ArsenalFootballer`
- `protected` members of `PremiershipFootballer` are protected members of `ArsenalFootballer`
- `private` members of `PremiershipFootballer` are hidden from `ArsenalFootballer`

These access privileges may be changed by using `protected` or `private`

The file `ArsenalFootballer.cpp` that contains the functions of the class is given by

```
#include "ArsenalFootballer.hpp"
ArsenalFootballer::ArsenalFootballer()
: PremiershipFootballer()
{
  club = "Arsenal";
}
```

This constructor sets the club of all members of the class of Arsenal footballers to "Arsenal"

Example code using this class is on the next slide

Note that we can still use the functions and variables of the class `PremiershipFootballer` when using an object of type `ArsenalFootballer`

The code - which may be written on one line - on the previous slide

```
ArsenalFootballer::ArsenalFootballer()
: PremiershipFootballer()
```

indicates which constructor in the class `PremiershipFootballer` we want to call when creating an object of the class `ArsenalFootballer`

To re-use a constructor of the class `PremiershipFootballer` that required two integers as input we would have

```
ArsenalFootballer::ArsenalFootballer(int a, int b)
: PremiershipFootballer(a, b)
{
  lines of code
}
```

in the file `ArsenalFootballer.cpp`

a constructor

```
ArsenalFootballer(int a, int b);
```

declared in the file `ArsenalFootballer.hpp`

and a constructor

```
PremiershipFootballer(int a, int b);
```

declared in the file `PremiershipFootballer.hpp`

```cpp
#include <iostream>
#include "PremiershipFootballer.hpp"
#include "ArsenalFootballer.hpp"
int main()
{
  ArsenalFootballer twalcott;
  std::cout << "Club is " << twalcott.club << "\n";
  twalcott.surname = "Walcott";
  twalcott.SetWeeklyWage(70000);
  std::cout << "Surname is " << twalcott.surname << "\n";
  std::cout << twalcott.GetWeeklyWage() << "\n";
}
```

Polymorphism

Polymorphism may be used when a number of classes are derived from the base class, and for some of these derived classes we want to override one of the functions of the base class

For example, consider a class of guests who stay at a hotel The class guest will have variables such as name, room type, arrival date, number of nights booked, minibar bill, telephone bill

This class will also have a function that computes the total bill

Suppose the hotel has negotiated special rates for individuals from particular organisations. The function that computes the total bill will be different for these clients

This can be handled with `if` (or `switch`) statements, but this can get messy. A more practical solution is to use virtual functions where the function used to compute the total bill does different things for different derived classes before returning the value of the total bill

This is known as *run-time polymorphism*

The function should be defined as `virtual` in the base class as shown on the next slide

The class Guest has header file `Guest.hpp`

```
#ifndef GUEST__
#define GUEST__

#include <string>

class Guest
{
public:
   std::string name, roomType, arrivalDate;
   int numberOfNights;
   double minibarBill, telephoneBill;
   virtual double CalculateBill();
};

#endif
```

The file `Guest.cpp` is

```
#include "Guest.hpp"
double Guest::CalculateBill()
{
  double room_bill, total;
  room_bill = ((double)(numberOfNights)) * 50;
  total = room_bill + minibarBill + telephoneBill;
  return total;
}
```

Suppose the hotel have negotiated a deal with a company that reduces the room rate to £45 for the first night and £40 for subsequent nights that a guest stays in the hotel

The header file for our derived class `SpecialGuest.hpp` is

```
#ifndef SPECIALGUEST__
#define SPECIALGUEST__

#include "Guest.hpp"

class SpecialGuest : public Guest
{
public:
  double CalculateBill();
};

#endif
```

The file SpecialGuest.cpp is

```
#include "SpecialGuest.hpp"
double SpecialGuest::CalculateBill()
{
  double room_bill, total;
  room_bill = ((double)(numberOfNights - 1)) * 40 + 45;
  total = room_bill + minibarBill;
  return total;
}
```

Example use of the class SpecialGuest is

```
#include <iostream>
#include "Guest.hpp"
#include "SpecialGuest.hpp"
int main()
{
  SpecialGuest harry;
  harry.numberOfNights = 2;
  harry.minibarBill = 30;
  std::cout << "Harry's bill = " << harry.CalculateBill() <
  return 0;
}
```

Note that declaring the function `CalculateBill()` as virtual in the class `Guest` does not require that this function must be redefined in derived classes - instead it gives us the option to redefine it

If the function wasn't redefined then objects of the class `SpecialGuest` would use the function `CalculateBill()` defined in the class `Guest`

Note that the function `CalculateBill()` could have been declared as `virtual` in the class `SpecialGuest`

This would allow any class derived from `SpecialGuest` to redefine the function `CalculateBill()` if desired

When using derived classes, the destructor for the base class should always be a *virtual function*

Example of polymorphism in action. (Imagine an array of pointers to Guests.)

```
Guest* p_gu1 = new Guest;
Guest* p_gu2 = new Guest;
Guest* p_gu3 = new SpecialGuest; //Pointer is of different

//Set the three guests identically
p_gu1->numberOfNights = 3;
p_gu2->numberOfNights = 3;
p_gu3->numberOfNights = 3;

std::cout << "Bill 1 = " << p_gu1->CalculateBill() << "\n"
std::cout << "Bill 2 = " << p_gu2->CalculateBill() << "\n"
std::cout << "Bill 3 = " << p_gu3->CalculateBill() << "\n"
// The last one gets a smaller bill
```