COMP6771
Advanced C++ Programming

Week 3
Part One: Classes - Overview, this Pointer and Constructors

2017

www.cse.unsw.edu.au/˜cs6771

What is Object-based Programming?

A class uses data abstraction and encapsulation to define an abstract data type:

- Interface: the operations used by the user (an API)
- implementation:
  - the data members
  - the bodies of the functions in the interface and any other functions not intended for general use
- Abstraction: separation of interface from implementation
- Encapsulation: enforcement of this via information hiding

Example: Sales Data - Sales_data.h (interface), Sales_data.cpp (implementation), user code (knows the interface).

Class Interface Declaration: Sales_data.h

```
#ifndef SALES_DATA_H
#define SALES_DATA_H

#include <iostream>

class Sales_data {
public:
  // constructors
  Sales_data(const std::string &s): bookName{s} { }
  Sales_data(const std::string &s, unsigned n, double p):
             bookName{s}, units_sold{n}, revenue{p*n} { }
  Sales_data() : Sales_data{"", 0, 0} { }
  Sales_data(std::istream &);
  // operations on Sales_data objects
  std::string getBookName() const;
  Sales_data& combine(const Sales_data&);
  double avg_price() const;
private:
  std::string bookName;
  unsigned units_sold {0}; // in-class initialiser
  double revenue {0.0};    // in-class initialiser
  mutable unsigned no_of_times_called {0};
};
#endif
```

Class Implementation Definition: Sales_data.cpp

```
#include "Sales_data.h"

// member functions
Sales_data::Sales_data(std::istream &is) {
  read(is, *this);
}

std::string Sales_data::getBookName() const {
  ++no_of_times_called;
  return bookName;
}

double Sales_data::avg_price() const {
  if (units_sold)
          return revenue/units_sold;
  else
          return 0;
}

Sales_data& Sales_data::combine(const Sales_data &rhs) {
  units_sold += rhs.units_sold;
  revenue += rhs.revenue;
  return *this;
}
```

Class Implementation Definition: Sales_data.cpp

Non-member class releated function

(note: these need to be delcared as a 'friend' to work)

```
std::istream& read(std::istream &is, Sales_data &item) {
  double price {0}; // unit price
  is >> item.bookName >> item.units_sold >> price;
  item.revenue = price * item.units_sold;
  return is;
}

std::ostream& print(std::ostream &os, const Sales_data &item) {
  os << item.bookName << " " << item.units_sold << " "
     << item.revenue << " " << item.avg_price();
  return os;
}
```

User Code

```
#include <iostream>

#include "Sales_data.h"

int main() {
  Sales_data curBook{"Harry Potter"};
  if (read(std::cin, curBook))  {
    Sales_data newBook{""};
    while(read(std::cin, newBook)) {
      if (curBook.getBookName() == newBook.getBookName()) {
        curBook.combine(newBook);
      } else {
        print(std::cout, curBook) << std::endl;
        curBook = newBook;
      }
   }
   print(std::cout, curBook) << std::endl;
  } else {
    std::cerr << "No data?!" << std::endl;
  }
}
```

Compilation

- Create the Sales_data.o file from Sales_data.cpp and Sales_data.h
  `g++ -std=c++14 -Wall -Werror -O2 -c Sales_data.cpp`
- Create the Sales_data_user.o from from Sales_data_user.cpp and Sales_data.h
  `g++ -std=c++14 -Wall -Werror -O2 -c Sales_data_user.cpp`
- Link the two .o files to form the Sales_data_user executable
  `g++ -std=c++14 -Wall -Werror -O2 -o Sales_data_user Sales_data_user.o Sales_data.o`

C++ Classes

A class:

- defines a new type and a new scope
- is created using the keywords `class` or `struct`
  a `struct` is a class whose members are all public
- defines zero or more type (alias), data and function members
- may contain zero or more `public` and `private` sections
- is instantiated through a constructor

A member function:

- must be declared inside the class
- may be defined inside the class (it is then `inline` by default)
- may be declared `const`, when it doesn't modify the data members

The data members are private, representing the state of an object.

Abstraction and Encapsulation

Abstraction and Encapsulation

We see that abstraction relies on separating the interface from the implementation. The concept of encapsulation refers to hiding details about class representation and implementation: an object's state can only be accessed/modified via the public interface.

Abstraction and encapsulation provide two advantages:

- object state is protected from user-level errors
- class implementation may evolve over time

Declarations vs Definitions

A class may be defined only once in a given source file. If defined in multiple files, the definitions must be identical.

Class definitions should be placed in header files to ensure uniformity. Header guards guarantee that class definitions are seen only once in each file (or only once in each program by using the preprocessor variables).

It is possible to just declare a class, using a forward declaration:

```
class Foo;
```

Incomplete Types

An incomplete type may only be used to define pointers and references, as well as in function declarations (but not definitions).

Declarations vs Definitions

Because of the restriction on incomplete types, a class cannot have data members of its own type. The following is illegal:

```
struct Node {
  int data;
  Node next, prev;
};
```

But the following is legal, since a class is considered declared once its class name has been seen:

```
struct Node {
  int data;
  Node *next, *prev;
};
```

Member Access Control

```
class Foo {
public:
  Members accessible by everyone

private:
  Accessible only by members & friends
}
```

- C++ classes support (in this way):
  - encapsulation
  - information hiding

Class Scope

```
Sales_data& Sales_data::combine(const Sales_data &rhs) {
  units_sold += rhs.units_sold;
  revenue += rhs.revenue;
  return *this;
}
```

- Provides a definition of `Sales_data& Sales_data::combine(const Sales_data &)`
- After the `::`, the body of the function is in the scope of class `Sales_data`, so that the usual name resolution process can be used

The Implicit `this` Pointer

A member function has an extra implicit parameter, named `this`, which is a pointer to the object on behalf of which the function is called. A member function does not explicitly define it, but may explicitly use it. The compiler treats an unqualified reference to a class member as being made through the `this` pointer.

The Implicit `const` pointer: `this`

The non-const member function combine is conceptually defined as:

```
Sales_data& combine(Sales_data* const this, Sales_data &rhs) {
  this->units_sold += rhs.units_sold;
  this->revenue += rhs.revenue;
  return *this;
}
```

- Can be invoked on a non-const object:

```
Sales_data data1("123", 2, 44.0), data2("123", 1, 22.0);
data1.combine(data2); // combine(data1&, data2);
```

- But cannot be invoked on a const object:

```
const Sales_data data3("123", 1, 22.0);
data3.combine(data1); // error
```

The Implicit `const` pointer: `this`

The `const` member function getBookName:

```
std::string getBookName() const {
  return bookName;
}
```

Is conceptually defined as:

```
std::string getBookName(const Sales_data* const this) const {
  return this->bookName;  // (*this).bookName
}
```

- Call be invoked on a const object:

  ```
  const Sales_data data1("123", 2, 44.0);
  data1.getBookName() // getBookName(&data1)
  ```

- Call also be invoked on a non-const object:

  ```
  Sales_data data2("123", 2, 44.0);
  data2.getBookName() // ok, too
  ```

Returning the Object

It is possible to use this to return a reference to the object:

```
Sales_data& Sales_data::combine(const Sales_data &rhs) {
  units_sold += rhs.units_sold;
  revenue += rhs.revenue;
  return *this;
}

const Sales_data& Sales_data::print(ostream &os) const {
  os << data << endl;
  return *this;
}
```

Note that the return statement is the same, but the return type is different!

Returning the Object

Are the following correct?

```
Sales_data a("Harry Potter");
Sales_data b("Harry Potter");

a.combine(b).print(std::cout);

a.print(std::cout).combine(b);
```

Returning the Object

Are the following correct?

```
Sales_data a("Harry Potter");
Sales_data b("Harry Potter");

a.combine(b).print(std::cout);

a.print(std::cout).combine(b);
```

The `combine/print` is fine, but the `print/combine` fails since `print` returns a `const` reference through which we cannot call a nonconst member.

Two solutions: overloading based on `const`, and mutable data members.

Overloading based on `const`

Recall that a function can be overloaded based on the `const`ness of its pointer arguments, and since the `const`ness of the implicit `this` argument changes, we can overload `print`:

```
const Sales_data& Sales_data::print(ostream &os) const
Sales_data& Sales_data::print(ostream &os)
```

NB

A `const` object can use only the const member. A non`const` object could use either member—but the non`const` version is a better match.

Mutable Data Members

```
class Sales_data {
public:
  std::string getBookName() const {  // ok
    ++no_of_times_called;
    return bookNo;
  }
private:
  std::string bookName;
  unsigned units_sold = 0;
  double revenue = 0.0;
  // counts no. of times getBookName() called on a Sales_data object
  mutable unsigned no_of_times_called = 0;
};
```

- allows a data member of a const object to be modified.
- The mutable data members represent the (logical) state of an object
- const-correctness for the logical not physical state of an object

Constructors

- Constructors define how class data members are initalised.
- A constructor has the same name as the class and no return type.
- Default initalisation is handled through the default constructor.
- Unless we define our own constructors the compile will define a default constructor.
- This is known as the synthesized default constructor.

The Compiler-Synthesized Default Constructor

```
   for each data member in declaration order
     if it has an in-class initialiser
       Initialise it using the in-class initialiser
    else
      if it is of a built-in type
          it is undefined // local scope
      else
          Initialise it using its default constructor
```

- The synthesised constructor makes j undefined

```
class A {
public:
  A() { }
};
```

```
class B {
private:
  int i {1};  // in-class initialiser
  int j;
  A a;
};
```

B b;

- Also we cannot synthesise the default constructor for `B` if `A()` is replaced by `A(int)`.

The Compiler-Synthesized Default Constructor

Don't rely on this unless you are confident about its behavior.

In addition, a synthesised constructor

- Is generated for a class only if it declares no constructors
- Is incorrect if some members of built-in types have no in-class initialisers
- Is unlikely to be correct if some members are raw pointers (as will be clear when we look at Copy Control)
- Cannot be generated if for a member of a user-defined type, the type doesn't have a default constructor

Constructor Initialiser List!

```
class A {
A() :            { }
}
```

Constructor Phases

The initialisation phase occurs *before* the body of the constructor is executed, regardless of whether the initialiser list is supplied.

A constructor will:

1. initialise all data members: each data member is initialised using the constructor initialiser or by default value (using the same rules as those used to initialise variables); then
2. execute the body of constructor: the code may assign values to the data members to override the initial values

(Overloaded) Constructors

- Constructor initialiser list:

```
Sales_data(const std::string &s, unsigned n, double p)
: bookNo{s}, units_sold{n}, revenue{p*n} { }
```

  1. First, the data members are initialised
  2. Then, the (empty) body is executed

- Explicitly initialise bookNo only and implicitly initialise the other two using the in-class initialisers

```
Sales_data(const std::string &s): bookNo{s} { }
// SAME AS
Sales_data(const std::string &s)
: bookNo{s}, units_sold{0}, revenue{0.0} { }
```

- Do the initialisation in the body:

```
Sales_data(std::istream &) { ... }
```

  Due to the lack of the initialiser list, the data members will be initialised first as per Slide 23 before this 3rd constructor is called.

Constructor Initialiser List

- The last three must be initialised on the initialiser list:

```
class ConstRef {
public:
  ConstRef(int ii) : i{ii}, ci{ii}, ri{i}, o{...} { }
private:
  int i;
  const int ci;  // const
  int &ri;  // reference
  noDefaultConstructor o;
};
```

- Don't do this:

```
ConstRef::ConstRef(int ii) {
 i = ii;  // ok -- uninitialized (by default initialisation)
 ci = ii; // error: cannot assign to a const
 ri = ii; // error: was never initialised
}         // error: noDefaultConstructor() doesn't exist
```

Order, Order!

- This doesn't work as expected!

```
#include<iostream>

class X {
public:
  X(int val) : j{val}, i{j} { }
private:
  int i;
  int j;
};
```

The members are always initialised in declaration order:

- Correct solutions:

```
X(int val) : i{val}, j{val} { }
// or
X(int val) : i{val}, j{i} { }
```

- Why? Because a class has only one destructor! The destructor of a class calls the destructors of its members in the reverse declaration order.

Readings

- Chapter 7
- C++ Object Initialisation:
  http://stackoverflow.com/questions/3127454/how-do-c-class-me
  http://stackoverflow.com/questions/12927169/how-can-i-initia