# COSC 151: Intro to Programming: C++

Chapter 7
Classes

# Chapter 7: Classes

- **Objectives**
  - Have the ability to write our own data types as classes
  - Understand the importance of **data abstraction** and **encapsulation**
    - In terms of separating an **interface** from an **implementation**
  - Define member data objects and member functions for our data types

# Why Classes?

- **Why do we even want to write our own data types?**
  - Modeling our applications on real world objects makes code easier to understand and reason about
  - Encapsulating data (keeping all related data together) is a great deal simpler than if we had to track it all separately

# C7: Defining Members

- **Classes we write can have data members, and member functions**

```
struct Sales_data
{
    // operations on Sales_data objects (member functions)
    std::string isbn() const { return bookNo; }
    Sales_data& combine(const Sales_data&);
    double avg_price() const;

    // Unchanged from 2.6.1 --
    // data members of Sales_data objects
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

# C7: Members and `this`

- **Inside a member function, we have access to the data members associated with a specific instance of the object**
  - There is an implicit parameter available in member functions: `this`
- **Inside a member function we can refer directly to the members of the object on which the function was called.**
  - Below, we don't need to use any member access operator to access the `bookNo` member variable inside our `Sales_data` class.

```
std::string isbn() { return bookNo; }
```

# C7: const Member Functions

- **const member functions cannot change the object on which they are called**

```
struct Foo
{
   void change_x() const
   {
      x = 1; // Compile error, attempt to
             // assign a const value since change_x
             // here is const, x is const
   }

   void change_x()
   {
      x = 1;  // OK, change_x is non-const, x is non-const
   }

   int x = 0;
}
```

# C7: Class Scope and Members

- **A class is itself a scope**
  - Definitions of date members and member functions are nested inside that scope
- **Classes are parsed by the compiler in two steps**
  - Declaration (looking at all the data members and member functions)
  - Definition (member function bodies)
  - So, within a class we can reference a class member that may not have been declared yet
    - See `isbn()` definition from Slide 4

# C7: Defining Member Functions

- **We can define member functions outside of the class**
  - In fact, most functions are defined this way to keep the implementation details from being present in the class declaration
  - We use the name of the **scope** (the class) and the **scope resolution operator** (**::**) to define member functions outside of the class.

```
// return type double
// function avg_price() in scope Sales_data (the class)
// the function is const (cannot change the data members of the class)
double Sales_data::avg_price() const
{
   if(units_sold)
   {
      return revenue / units_sold;
   }

   return 0;
}
```

# C7: Chaining Function Calls

- **We want our types to behave just like the built in types**

```
int x = 0, y = 9;
x = y = 10; // Chained assignment
```

- **We can accomplish this chaining by having functions return a reference to the current class object.**
  - Remeber we have an implicit parameter **this** available in member functions.

```
// Returns a reference to a Sales_data& object
// takes a const reference to a Sales_data object
// rhs -- an idiomatic name for “right hand side”
// this function adds the information from the “right hand side”
// to “the current item” - returns a reference to “the current item”
Sales_data& combine(const Sales_data& rhs)
{
   units_sold += rhs.units_sold;  // add up units sold
   revenue += rhs.revenue;        // add up revenue
   return *this;     // return reference to “this”
}
```

# C7: Non-Member Class Related Functions

- **When writing classes, prefer to keep the interface as small as possible**
  - Defining auxiliary functions that can be implemented using the public interface of the class can keep it small
  - For `Sales_data`, `read`, `add`, `print` functions all use the public interface, so they are not part of the class itself.
    - It’s good practice to keep the declaration of these auxiliary functions in the same header as the class

# C7: Non-Memeber Functions (contd)

- **The `read` function reads sales data objects from an input stream.**

```
istream& read(istream& is, Sales_data& item)
{
   double price = 0;
   is >> item.bookNo >> item.units_sold >> price;
   item.revenue = price * item.units_sold;
   return is;
}
```

- **The `print` function prints sales data objects to an output stream.**

```
ostream& print(ostream& is, Sales_data& item)
{
   os << item.isbn() << " " << item.units_sold << " "
      << item.revenue() << " " << item.avg_price();
   return os;
}
```

# C7: Non-Member Functions (contd)

- **The `add` function takes two Sales_data objects and returns a new Sales_data object representing their sum**

```
Sales_data add(const Sales_data& lhs,  // “left hand side”
               const Sales_data& rhs)  // “right hand side”
{
   Sales_data sum = lhs; // copy the left
   sum.combine(rhs);     // combine left and right
   return sum;           // return sum --
                         //    a new temporary object
}
```

# C7: Constructors

- **A constructor is a special member function that is run whenever an object of a class type is created**
  - The constructor is responsible for initializing the class members
  - As programmers, we have to decide what constructors are appropriate and necessary
- **A constructor should setup the invariant for the type**
  - An **invariant** is something that should always hold true – more on that later.

# C7: Default Constructors

- **A default constructor is a constructor that takes no arguments**
  - If the user doesn’t declare any constructors, the compiler will generate a default constructor for the class
  - The compiler generated constructor uses default initialization of all data members

# C7: Default Constructors (contd)

- **If the user declares any constructors the compiler will not generate a default**
  - If a default constructor (takes no arguments) is required, the user will have to explicitly provide it in this case

# C7: Constructors (contd)

- **The user can define any number of constructors**
  - It’s up to you to determine what is appropriate and what is not

# C7: `Sales_data` Constructors

- **The book defines 4 constructors for the `Sales_data` object, each with a different parameter list**
  - An `istream&` from which a transaction can be read
  - A `const string&` representing an ISBN, an `unsigned` representing the count of books sold and a `double` representing the price at which the books sold
  - A `const string&` representing an ISBN, where the other values (`units_sold` and `revenue`) use default values
  - An empty parameter list – the default constructor – needed because we want to be able to default construct `Sales_data` objects, and we've explicitly defined other constructors

# C7: Declaring Constructors

- **Like other member functions, constructors can be declared and defined inline, or declared and defined elsewhere**

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

   // .. other members are unchanged from Slide 4
};
```

# C7: Explaining Constructors

```
Sales_data() = default;
```

- **This defines the default constructor**
  - It takes no arguments
  - The `= default` syntax tells the compiler to generate the default constructor as it would have if we hadn't declared other constructors

# C7: Explaining Constructors (contd)

- **The next two constructors use a constructor initializer list**
  - This peculiar syntax allows us to initialize members of our class
  - You should prefer this to initializing members in the constructor body
    - See: http://stackoverflow.com/questions/1711990/what-is-this-weird-colon-member-syntax-in-the-constructor

# C7: Constructor Initializer List

- **After the constructor, if we use a : immediately before the constructor body, we can initialize one or more class data members.**
  - Members not included in the initializer list are default initialized

```
Sales_data(const std::string& s)
      : bookNo{s} // initialize bookNo with s
   {}
```

# C7: Constructor definition (contd)

- **We can choose to define the constructor outside of the class declaration.**
  - It has similar syntax to other member functions, but can use a constructor initializer list

```
Sales_data::Sales_data(std::istream& in)
   : bookNo{“”}, units_sold{0}, revenue{0}
{
   read(in, *this); // use the read() function
                    // to initialize this object
}
```

# C7: Copy/Assignment/Destruction

- **The compiler generates functions for copy, assignment and destruction**
  - The compiler generated versions aren’t always suitable, and in some cases we need to define our own.
  - BUT, we should try to **avoid that** as much as possible

# C7: Access Control and Encapsulation

- **Up to now the `Sales_data` type doesn't force users to interact with its interface**
  - Users of the class can freely access all member functions and data members
  - Allowing this type of access allows users to bypass the interface, and change things that may be simply incorrect.

```
// Assume I've sold 50 copies of my textbook for $100 each
Sales_data textbook{"11111", 50, 100.00};

cout << textbook.revenue << endl;  // prints 5000 awww yisss

textbook.revenue = 0;  // but you can change it...

cout << textbook.revenue << endl;  // prints 0… where'd my money go!
```

# C7: Access Control and Encapsulation (contd)

- **Let's take a simple example**

```
struct date
{
   int year = 0;
   int month = 1;
   int day = 1;
};
```

  - Here we have a simple data type that represents a date
  - It's pretty useful – it can represent any date in the past or the present!

```
date d{1984, 2, 14}; // Valentine's Day 1984
 d.month = 28;       // hmmm….???
```

# C7: Access Control and Encapsulation (contd)

- **Our simple `date` class, like our `Sales_data` class suffers from allowing users to modify our invariant by bypassing our interface.**
  - The invariant of our date class should be
    - The date held is always a valid day/month/year
  - The invariant of our `Sales_data` class should be
    - The revenue represents the total revenue of units sold multiplied by the sales price for a book with a given ISBN.

# C7: `struct` or `class`?

- **The keywords struct and class for a data type have the exact same meaning**
- **They can be substituted for one another**
- **The difference is the default access control**
  - `struct` members are default public
  - `class` members are default private

# C7: Access Control and Encapsulation (contd)

- **We need a way to limit users of our data types from manipulating state without using the interfaces we provide**
  - Fortunately the language allows us to do just that using the keywords `public` and `private`.
  - We specify these **access specifiers** in our class definition, then everything below that (to the end of the class declaration, or to the next access specifier) have that visibility.

# C7: A Better date

```
class date
{
public:  // things below are accessible to users
   date(int year, int month, int day);

   int get_year() const { return year; }
   int get_month() const { return month; }
   int get_day() const { return day; }

private: // things below are not accessible to users
   int year = 0;
   int month = 1;
   int day = 1;
};

date my_birthday{1978, 10, 20};
my_birthday.year = 1998; // wouldn't we all like to be young again?
                         // Too bad, this is a compile error
                         // cannot access private member year
```

# C7: Friends

- **Some functions are best implemented outside the class (like `read()`, `print()` for Sales_data) but still need access to private data members.**
  - We can do that too, by making them friends.

```
class Sales_data
{
public:
   // ...
private:  // things below are accessible to users
   std::string bookNo;
   unsigned units_sold = 0;
   double revenue = 0;

   // Grant “friendship” to read and print, they can now access
   // the private data mebers of the Sales_data class

   friend std::istream& read(std::istream&, Sales_data&);
   friend std::ostream& print(std::ostream&, const Sales_data&);
};
```

# C7: Defining Type Members

- **Sometimes we want to be more explicit in the types we support**
  - We can define types within our class, allowing us to be more explicit in our meaning

```
class date
{
public:  // things below are accessible to users
   using YEAR = int;
   using MONTH = int;
   using DAY = int;

   date(YEAR year, MONTH month, DAY day);

   YEAR get_year() const { return year; }
   MONTH get_month() const { return month; }
   DAY get_day() const { return day; }

private: // things below are not accessible to users
   YEAR year = 0;
   MONTH month = 1;
   DAY day = 1;
};
```

# Final Thoughts

- Writing our own data types makes our code easier to read, write and understand
- Data types can have data members and member functions
- Member functions get an implicit parameter `this` that is a pointer to the current instance
- `const` member functions cannot change the objects on which they are called
- Use the scope resolution operator to define members outside of the class

# Final Thoughts (contd)

- **Prefer non-member functions that use a classes interface to member functions**
- **Constructors are special functions that are used to initialize class members**
  - Defualt constructors take no arguments
  - Any number of constructors can be defined
- **Assignment, copy and destruction can be automatically generated by the compiler**
  - These versions aren’t always appropriate – we will learn about that later
  - But we should seek to write classes where the compiler generated versions ***are*** appropriate.

# Final Thoughts (contd)

- **Constructors have a strange initialization list syntax to allow you to initialize members**
  - Prefer this to initializing in the constructor body
- **Control user access to members via `public` and `private`**
- **Grant special access to specific functions as necessary by making them friends**
  - Only friends can see your privates
- **Define member types if it makes the code more reasonable**