Classes, Private and Public Members, Constructors, Helper and Utility Methods

Classes in C++

Introduction

- A class is a set of data and functions that define the attributes (ie., characteristics) and interface (i.e., behaviour/functionalities) of an object

- With classes, C++ allows you, e.g., to create a new type called `Result` that includes data about both mean and standard deviation

- These two quantities are connected logically and it may be useful, efficient, and/or convenient to have them “go together”

```
Result mean_and_stdev(const double* data, int nData) {
  Result result;
  // do your calculation
  return result;
}
```

Classes in C++

Based on `examples/03/Result.cpp`

Interface
or
Member Functions

Attributes
or
Data Members

```
class Result {
  public:

    // constructors
    Result() { };
    Result(const double& mean, const double& stdDev) {
      mean_ = mean;
      stdDev_ = stdDev;
    }

    // accessors
    double getMean() { return mean_; };
    double getStdDev() { return stdDev_; };

  private:
    double mean_;
    double stdDev_;
};
```

Don't forget this!

Classes in C++

Based on `examples/03/Result.cpp`

```
int main() {

  Result r1;
  cout << "r1, mean: " << r1.getMean()
       << ", stdDev: " << r1.getStdDev()
       << endl;

  Result r2(1.1, 0.234);
  cout << "r2, mean: " << r2.getMean()
       << ", stdDev: " << r2.getStdDev()
       << endl;

  cout << sizeof(r1) << endl;
  cout << sizeof(r2) << endl;

  return 0;
}
```

- No instructions on “filling up” `r1`
- Call for `r2` does “fill up” `r2`
- Regardless, the size of `Result` is twice the size of a `double`

```
$ g++ -o Result Result.cpp
$ ./Result
r1, mean: 0, stdDev: 0
r2, mean: 1.1, stdDev: 0.234
16
16
```

Attributes (or Data Members) of a Class

- Are data defined in the scope of that class

- Are defined in the class and can be used by all member functions

- Contain the actual data that characterise the content of the class

- Can be public or private
  - public data members are generally bad and a symptom of bad design
  - more on this topic later in the course

```
class Datum {
    double value_;
    double error_;
};
```

Interface (or Member Functions) of a Class

- Is the set of methods defined inside the scope of a class

- Member functions have access to all data members and can also be public or private

- All C++ rules discussed so far apply to arguments of member functions
  - Pass variables by value, pointer, or reference
  - Use the `const` qualifier to protect input data and restrict the capabilities of the methods
    - This has implications on declaration of methods using constants
    - We will discuss constant methods and data members later in the course
  - Member functions can return any type with the **exception** of Constructors and Destructor
    - These have no return type (more on this later)

Interface (or Member Functions) of a Class

Example

- `name_` is a data member; it is not declared in any member function!
- `name` is a local variable only within the member function `setName` (and the constructor `Student`)

```
#include <iostream>
#include <string>

class Student {
  using namespace std;

  public:
    // default constructor
    Student() { name_ = ""; }

    // another constructor
    Student(const string& name) { name_ = name; }

    // getter method: access to info from the class
    string name() { return name_; }

    // setter: set attribute of object
    void setName(const string& name) { name_ = name; }

    // utility method
    void print() {
      cout << "My name is: " << name_ << endl;
    }

  private:
    string name_; // data member
};
```

Access Specifiers: `public` and `private`

Based on `examples/03/Class1.cpp`

- `public` functions and data members are available to anyone
- `private` members and methods are available only to other member functions (and to `friend` classes which we will return to later in the course)
- Elements of a class are accessible via the member selection operator "`.`"

```
#include <iostream>
using std::cout;
using std::endl;

class Datum {
  public:
    Datum() { }
    Datum(double val, double error) {
      value_ = val;
      error_  = error;
    }

  double value() { return value_; }
  double error() { return error_; }

  void setValue(double value) { value_ = value; }
  void setError(double error) { error_ = error; }

  double value_; // public data member!!!

  private:
    double error_; // private data member
};
```

This provides access to a private member correctly

```
int main() {

  Datum d1(1.1223,0.23);

  cout << "d1.value(): " << d1.value()
       << " d1.error(): " << d1.error()
       << endl;


  cout << "d1.value_: " << d1.value_
       << " d1.error_: " << d1.error_
       << endl;

  return 0;
}
```

This combination yields an error at compilation time

Private Methods

Based on `examples/03/Class2.cpp` and `examples/03/Class3.cpp`

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

class Datum {
  public:
    Datum() { reset(); } // reset data members

    double value() { return value_; }
    double error() { return error_; }

    void setValue(double value) { value_ = value; }
    void setError(double error) { error_ = error; }

    void print() {
      cout << "datum: " << value_ << " +/- " << error_ << endl;
    }

  private:
    void reset() {
      value_ = 0.0;
      error_ = 0.0;
    }

    double value_;
    double error_;
};
```

Attempting access to private function outside the class!

```
int main() {

  Datum d1;
  d1.setValue( 8.563 );
  d1.print();


  return 0;
}
```

```
int main() {

  Datum d1;
  d1.setValue( 8.563 );
  d1.print();
  d1.reset();

  return 0;
}
```

Which one compiles successfully?

```
$ g++ -o Class2 Class2.cpp
$ ./Class2
datum: 8.563 +/- 0
```

Hiding Implementation from Users/Clients

- How do you decide what to make public or private?
  - **Principle of Least Privilege**: attributes or methods of a class must be private unless proven to be needed as public
- Rephrasing: users should be able to rely solely on the interface of a class and never need to use the internal details of the class
- This implies that **having public data members is a VERY bad idea**!
  - Name and characteristics of data members can change
  - Functionalities and methods remain the same
  - Must be able to change internal structure of the class without affecting users

Bad Example: Public Data Members

Based on `examples/03/Class4.cpp`

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

class Datum {
  public:
    Datum(double val, double error) {
      value_ = val;
      error_  = error;
    }

    double value() { return value_; }
    double error() { return error_; }

    void setValue(double value) { value_ = value; }
    void setError(double error) { error_ = error; }

    void print() {
      cout << "datum: " << value_ << " +/- " << error_ << endl;
    }

  //private:       // all data are public!
    double value_;
    double error_;
};
```

```
int main() {

  Datum d1(1.1223,0.23);
  double x = d1.value();
  double y = d1.error_;

  cout << "x: " << x << "\t y: " << y << endl;

  return 0;
}
```

```
$ g++ -o Class4 Class4.cpp
$ ./Class4
x: 1.1223    y: 0.23
```

- Works, but user accesses data members directly
- What if the class modifies them internally?
- What if the developer modifies the class?

Bad Example: Public Data Members

Based on `examples/03/Class5.cpp`

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

class Datum {
  public:
    Datum(double val, double error) {
      val_ = val;
      err_ = error;
    }

    double value() { return val_; }
    double error() { return err_; }

    void setValue(double value) { val_ = value; }
    void setError(double error) { err_ = error; }

    void print() {
      cout << "datum: " << val_ << " +/- " << err_ << endl;
    }

  //private:       // all data are public!
    double val_;
    double err_;
};
```

```
int main() {

  Datum d1(1.1223,0.23);
  double x = d1.value();
  double y = d1.error_;

  cout << "x: " << x << "\t y: " << y << endl;

  return 0;
}
```

- Suppose a developer simply relabels the class data members
- The code is now broken, despite the class not changing!
- Bad programming!
- Only use the interface of an object not its internal data!
- Private data members prevent this

Constructors

Based on `examples/03/Result.cpp`

```
class Result {
  public:

    // constructors
    Result() { };
    Result(const double& mean, const double& stdDev) {
      mean_ = mean;
      stdDev_ = stdDev;
    }

    // accessors
    double getMean() { return mean_; };
    double getStdDev() { return stdDev_; };

  private:
    double mean_;
    double stdDev_;
};
```

- Special member functions required by C++ to create a new object
- Their purpose is to initialize data members of an instance of the class
- There can be several constructors per class, i.e., different ways to declare an object of a given type
- **Must have same name as class**
- Have no return type
- Can accept any number of arguments (usual C++ rules for functions apply)

Types of Constructors

Based on `examples/03/Class6.cpp`

```
class Datum {
  public:
    Datum() { }

    Datum(double x, double y) {
      value_ = x;
      error_ = y;
    }

    Datum(const Datum& datum) {
      value_ = datum.value_;
      error_ = datum.error_;
    }

  private:
    double value_;
    double error_;
};
```

1. Default constructor
   - Has no argument
   - On most machines the default values for data members are assigned
2. Regular Constructor
   - Provides sufficient arguments to initialise all data members
3. Copy constructor
   - Makes a new object from a pre-existing one

Usage of Constructors

Based on `examples/03/Class6.cpp`

```
class Datum {
  public:
    Datum() { }

    Datum(double x, double y) {
      value_ = x;
      error_ = y;
    }

    Datum(const Datum& datum) {
      value_ = datum.value_;
      error_ = datum.error_;
    }

   void print() {
      cout << "datum: " << value_
           << " +/- " << error_
           << endl;
    }

  private:
    double value_;
    double error_;
};
```

Constructors

1. Default
2. Regular
3. Copy

```
int main() {

  Datum d1;
  d1.print();

  Datum d2(0.23,0.212);
  d2.print();

  Datum d3(d2);
  d3.print();

  return 0;
}
```

The behaviour of default constructors varies with architecture

```
$ g++ -o Class6 Class6.cpp
$ ./Class6
datum: 0 +/- 0
datum: 0.23 +/- 0.212
datum: 0.23 +/- 0.212
```

Advanced: Named Constructor Idiom

Based on `examples/03/2DCoordinates.cpp`

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

class Point {
    private:
        double x, y;

        // regular constructor
        Point(double x1, double x2) {
            x = x1;
            y = x2;
        };

    public:
        // polar(radius, angle)
        static Point Polar(double r, double theta) {
            return Point(r*cos(theta), r*sin(theta));
        }

        // cartesian(x, y)
        static Point Cartesian(double x, double y) {
            return Point(x,y);
        }

        // utility function to display coordinates
        void display() {
            cout << "x = " << x <<endl;
            cout << "y = " << y <<endl;
        }
};
```

- A 2D `Point` has 2 attributes: 2 `Polar` or `Cartesian` coordinates
- How can we handle both variants?
    - Make the regular `Point` constructor `private`
    - Provide 2 `public` methods that aptly invoke the regular constructor
- This is the **Named Constructor Idiom**: declare all constructors in `private` and create `public static` member functions to access objects of class (we will cover `static` later in the course)

Advanced: Named Constructor Idiom

Based on `examples/03/2DCoordinates.cpp`

```
int main() {
    // Two coordinates
    double x1 = 5.7, x2 = 1.2;

    // Polar coordinates
    Point pp1 = Point::Polar(x1, x2);
    cout << "polar coordinates \n";
    pp1.display();

    // Polar coordinates
    Point pp2 = Point::Polar(x2, x1);
    cout << "polar coordinates \n";
    pp2.display();

    // Cartesian coordinates
    Point pr1 = Point::Cartesian(x1, x2);
    cout << "rectangular coordinates \n";
    pr1.display();

    // Cartesian coordinates
    Point pr2 = Point::Cartesian(x2, x1);
    cout << "rectangular coordinates \n";
    pr2.display();

    return 0;
}
```

- Note the `::` syntax to invoke the (`static`) member functions `Polar` and `Cartesian`, in the absence of `public Point` constructors

```
$ g++ -o 2DCoordinates 2DCoordinates.cpp
$ ./2DCoordinates
polar coordinates
x = 2.06544
y = 5.31262
polar coordinates
x = 1.00166
y = -0.660823
rectangular coordinates
x = 5.7
y = 1.2
rectangular coordinates
x = 1.2
y = 5.7
```

Accessors and Helper/Utility Methods

- Allow read access to data members
- Can also provide functionalities commonly needed by users to elaborate information from the class, e.g., formatted printing of data
- Usually they do not modify the objects, i.e., they do not change the value of attributes

```
class Student {
    public:

    // getter method: access to data members
    string name() { return name_; }

    // utility method
    void print() {
        cout << "My name is: " << name_ << endl;
    }

    private:
        string name_; // private data member
};
```

```
class Datum {
    public:

        double value() { return value_; }
        double error() { return error_; }

        void print() {
            cout << "datum: " << value_
                 << " +/- " << error_
                 << endl;
        }

    private:
        double value_; // private data member
        double error_; // private data member
};
```

Getter Methods

- Helper methods with explicit names returning individual data members
- **Do not modify data members** and simply return them
- Good practice: for, e.g., a member `foo_`, label these methods `getFoo()` or `foo()`
- Return value of a getter method should be that of the data member being grabbed

```
class Student {
    public:
        // constructors
        Student() { name_ = ""; }
        Student(const string& name) { name_ = name; }

        // getter method: access to info from the class
        string name() { return name_; }

        // utility method
        void print() {
            cout << "My name is: " << name_ << endl;
        }

    private:
        string name_; // data member
};
```

```
class Datum {
    public:
        Datum(double val, double error) {
            val_ = val;
            err_ = error;
        }

        double value() { return val_; }
        double error() { return err_; }

        void print() {
            cout << "datum: " << val_ << " +/- " << err_ << endl;
        }

    private:
        double val_; // data member
        double err_; // data member
};
```

Setter Methods

Based on `examples/03/Class7.cpp`

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

class Datum {
  public:
    // constructor
    Datum(double val, double error) {
      value_ = val;
      error_ = error;
    }

    // getters
    double value() { return value_; }
    double error() { return error_; }

    // setters
    void setValue(double value) { value_ = value; }
    void setError(double error) { error_ = error; }

    void print() {
      cout << "datum: " << value_ << " +/- " << error_ << endl;
    }

  private:
    double value_; // data member
    double error_; // data member
};
```

- Member functions that **modify attributes** of an object after it is created
- Typically defined as void, but could return other values for error handling purposes
- Very useful to assign correct attributes to an object in algorithms
- Warning: abusing setter methods can cause unexpected problems

```
int main() {

  Datum d1(23.4, 7.5);
  d1.print();

  d1.setValue(8.563);
  d1.setError(0.45);
  d1.print();

  return 0;
}
```

```
$ g++ -o Class7 Class7.cpp
$ ./Class7
datum: 23.4 +/- 7.5
datum: 8.563 +/- 0.45
```

Pointers and References to Objects

Based on `examples/04/PointerToClass.cpp`

```
#include <iostream>
using std::cout;  // use using only for specific classes
using std::endl;  // not for entire namespace

class Counter {
  public:
    Counter() { count_ = 0; x_ = 0.0; };
    int value()  { return count_; }
    void reset() { count_ = 0; x_ = 0.0; }
    void increment() { count_++; }
    void increment(int step) { count_ = count_+step; }
    void print() {
      cout << "---- Counter::print() : begin ----" << endl;
      cout << "my count_: " << count_ << endl;
      cout << "my address: " << this << endl; // this is a special pointer
      cout << "&count_ : " << &count_ << "  sizeof(count_): " << sizeof(count_) << endl;
      cout << "&x_ : " << &x_ << "  sizeof(x_): " << sizeof(x_) << endl;
      cout << "---- Counter::print() : end ----" << endl;
    }

  private:
    int count_;
    double x_; // dummy variable
};
```

```
// print counter info by reference
void printByRef(Counter& counter)  {
  cout << "printByRef: counter value: " << counter.value() << endl;
}

// print counter info of pointer to counter
void printByPtr(Counter* counter)  {
  cout << "printByPtr: counter value: " << counter->value() << endl;
}

int main() {
  Counter counter;
  counter.increment(7);

  // ptr is a pointer to a Counter Object
  Counter* ptr = &counter;
  cout << "ptr = &counter: " << &counter << endl;

  // use . to access member of objects
  cout << "counter.value(): " << counter.value() << endl;

  // use -> with pointer to objects
  cout << "ptr->value(): " << ptr->value() << endl;

  printByRef( counter );
  printByPtr( ptr );

  ptr->print();

  cout << "sizeof(ptr): " << sizeof(ptr) << "\t"
       << "sizeof(counter): " << sizeof(counter)
       << endl;

  return 0;
}
```

Use -> instead of . with pointers to objects

Pointers and References to Objects

Based on `examples/04/PointerToClass.cpp`

```
ptr = &counter: 0x16b013718
counter.value(): 7
ptr->value(): 7
printByRef: counter value: 7
printByPtr: counter value: 7
---- Counter::print() : begin ----
my count_: 7
my address: 0x16b013718
&count_ : 0x16b013718  sizeof(count_): 4
&x_ : 0x16b013720  sizeof(x_): 8
---- Counter::print() : end ----
sizeof(ptr): 8            sizeof(counter): 16
```

0x16b013718 + sizeof(int) = 0x16b013720

```
// print counter info by reference
void printByRef(Counter& counter)  {
  cout << "printByRef: counter value: " << counter.value() << endl;
}

// print counter info of pointer to counter
void printByPtr(Counter* counter)  {
  cout << "printByPtr: counter value: " << counter->value() << endl;
}

int main() {
  Counter counter;
  counter.increment(7);

  // ptr is a pointer to a Counter Object
  Counter* ptr = &counter;
  cout << "ptr = &counter: " << &counter << endl;

  // use . to access member of objects
  cout << "counter.value(): " << counter.value() << endl;

  // use -> with pointer to objects
  cout << "ptr->value(): " << ptr->value() << endl;

  printByRef( counter );
  printByPtr( ptr );

  ptr->print();

  cout << "sizeof(ptr): " << sizeof(ptr) << "\t"
       << "sizeof(counter): " << sizeof(counter)
       << endl;

  return 0;
}
```

Pointers and References to Objects

Based on `examples/04/PointerToClass.cpp`

```
ptr = &counter: 0x16b013718
counter.value(): 7
ptr->value(): 7
printByRef: counter value: 7
printByPtr: counter value: 7
---- Counter::print() : begin ----
my count_: 7
my address: 0x16b013718
&count_ : 0x16b013718  sizeof(count_): 4
&x_ : 0x16b013720  sizeof(x_): 8
---- Counter::print() : end ----
sizeof(ptr): 8             sizeof(counter): 16
```

0x16b013718 + sizeof(int) = 0x16b013720

- Size of objects is platform-dependent (after all they contain data members the size of which is platform-dependent)
- Address of object is address of first data member in the object (just as arrays)