C++ Program Design
-- Object-oriented programming

Junjie Cao @ DLUT

Summer 2016

http://jjcao.github.io/cPlusPlus

Procedural programming

- Such as C

1. Split it up into a set of tasks and subtasks
2. Make functions for tasks
3. Instruct the computer to perform them in sequence

- Sequence is obvious but hierarchy is very ambiguity.
- It is hard to organize too much of functions

Procedural vs Object-Oriented programming

- People think of the world in terms of interacting **objects**
  - **Properties + behaviors** (they are **inseparable**)

- With procedural programming, the properties (data) and behaviors (functions) are **separated**
  - does not provide a very intuitive representation of reality
  - It’s up to the programmer to manage and connect the properties to the behaviors in an appropriate manner.

- OOP provides us with the ability to design an “object”: tool to manage complexity when needed
  - easier to write and understand
  - a higher degree of code-reusability

Once you've been properly familiarized with OOP, you'll likely never want to go back to pure procedural programming again.

Classes and class members

Struct

```
struct DateStruct
{
    int year;    int month;
    int day;};

void print(DateStruct &date){
    std::cout << date.year << "/" << date.month << "/" << date.day;
}

int main(){
    DateStruct today { 2020, 10, 14 }; // use uniform initialization

    today.day = 16; // use member selection operator to select a member of the struct
    print(today);

    return 0;}
```

Classes

```
struct DateStruct{
    int year;
    int month;
    int day;
};

class DateClass{
public:
    int m_year;
    int m_month;
    int m_day;
};
```

Pay attention to the difference

Instance and object

- `DateClass today { 2020, 10, 14 }; // declare a variable of class DateClass`

- when we define a variable of a class, we call it **instantiating** the class.
- The variable itself is called an **instance** of the class.
- A variable of a class type is also called an **object**.
- Instantiating a variable allocates memory for the object.

Member Functions

```
class DateClass
{
public:
    int m_year;
    int m_month;
    int m_day;

    void print() // defines a member function named print()
    {
        std::cout << m_year << "/" << m_month << "/" << m_day;
    }
};
```

the member selector operator (.)

```
int main()
{
    DateClass today { 2020, 10, 14 };

    today.m_day = 16; // use member selection operator to select a member variable of the class
    today.print(); // use member selection operator to select a member function of the class

    return 0;
}
```

```cpp
class Employee{
public:
    std::string m_name;
    int m_id;
    double m_wage;

    void print(){
      std::cout << "Name: " << m_name <<
            "  Id: " << m_id <<
            "  Wage: $" << m_wage << '\n';
    }
};
```

```cpp
int main()
{
    // Declare two employees
    Employee alex { "Alex", 1, 2
5.00 };
    Employee joe { "Joe", 2, 22.
25 };

    // What will be printed
    alex.print();
    joe.print();

    return 0;
}
```

Public vs private access specifiers

```
struct DateStruct{ // members are public by default
    int month; // public by default, can be accessed by anyone
    int day; // public by default, can be accessed by anyone
    int year; // public by default, can be accessed by anyone
};

int main(){
    DateStruct date;
    date.month = 10;    date.day = 14;
    date.year= 2020;

    return 0;}
```

```
class DateClass{ // members are private by default
    int m_month; // private by default, can only be accessed by other members
    int m_day; // private by default, can only be accessed by other members
};

int main(){
    DateClass date;
    date.m_month = 10; // error
    date.m_day = 14; // error

    return 0;}
```

```
class DateClass{
public:
    int m_year;
    int m_month;
    int m_day;
};
```

Mixing access specifiers

```
class DateClass // members are private by default
{
    int m_month; // private
    int m_day; // private
    int m_year; // private
public:
    void setDate(int month, int day, int year) {
        m_month = month;        m_day = day;        m_year = yea
r;
    }
…
};
```

```cpp
class DateClass // members are private by default
{
    ...
public:
    ...

    void print() // public, can be accessed by anyone
    {
        std::cout << m_month << "/" << m_day << "/" << m_year;
    }
};
```

Mixing access specifiers

```
int main()
{
    DateClass date;
    date.setDate(10, 14, 2020); // okay, because setDate() is public
    date.print(); // okay, because print() is public

    return 0;
}
```

public interface: setDate(), print()

Rule: Make member variables private, and member functions public, unless you have a good reason not to.

Quiz time

- 1a) What is a public member?
- 1b) What is a private member?
- 1c) What is an access specifier?
- 1d) How many access specifiers are there, and what are they?

Quiz time

- 2) Write a simple class named Point3d. The class should contain:
  * Three private member variables of type double named m_x, m_y, and m_z;
  * A public member function named setValues() that allows you to set values for m_x, m_y, and m_z.
  * A public member function named print() that prints the Point in the following format: <m_x, m_y, m_z>
- Make sure the following program executes correctly:

```
int main(){
    Point3d point;
    point.setValues(1.0, 2.0, 3.0);
    point.print();

    return 0;}
```

Assignment

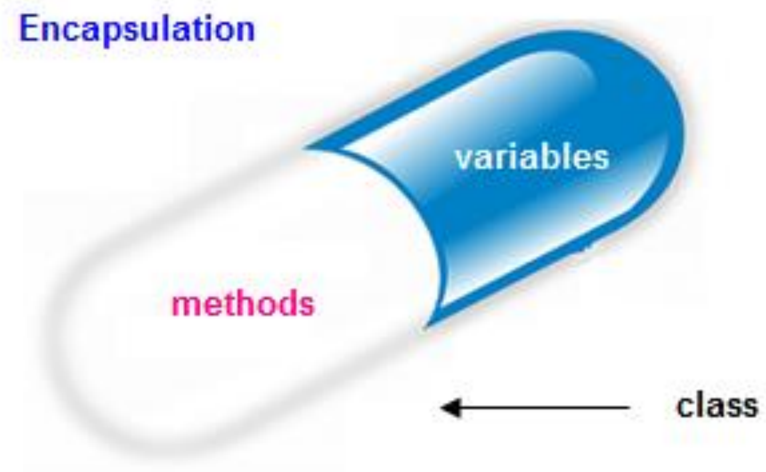
- write a class that implements a simple stack.

```
int main(){
Stack stack;
stack.reset();
stack.print();

stack.push(5);stack.push(3);stack.push(8);stack.print();

stack.pop();stack.print();

stack.pop();stack.pop();stack.print();

return 0;}
```

Why make member variables private?
Encapsulation

Encapsulation

- In OOP, **Encapsulation** (also called **information hiding**) is the process of keeping the details about how an object is implemented hidden away from users of the object.



- Instead, users of the object access the object through a **public interface**.

- In this way, users are able to use the object without having to understand how it is implemented.

Benefit: encapsulated classes are easier to use and reduce the complexity of your programs

- only need to know public members to use the class
- It doesn’t matter how the class was implemented internally
  - a class holding a list of names could have been implemented using a dynamic array of C-style strings, std::array, std::vector, std::map, std::list, or one of many other data structures.
-
- dramatically reduces the complexity of your programs, and also reduces mistakes
- Imagine how much more complicated C++ would be if you had to understand how std::string, std::vector, or std::cout were implemented in order to use them!

Benefit: encapsulated classes help protect your data and prevent misuse

- two variables have an intrinsic connection

```
class MyString{
    char *m_string; // we'll dynamically allocate our string here
    int m_length; // we need to keep track of the string length
};
```

- If m_length were public, anybody could change the length of the string without changing m_string (or vice-versa) => inconsistent state

- use public member functions can ensure that m_length and m_string are always set appropriately

Benefit: encapsulated classes help protect your data and prevent misuse

- two variables have an intrinsic connection

```
class IntArray{
public:
    int m_array[10];
};

IntArray array;
array.m_array[16] = 2; // invalid array index, now we overwrote memory that we don't own
```

- How to solve this?

```cpp
class IntArray
{
private:
    int m_array[10]; // user can not access this directly any more
public:
    void setValue(int index, int value){
        // If the index is invalid, do nothing
        if (index < 0 || index >= 10)
            return;

        m_array[index] = value;
    }
};
```

Benefit: encapsulated classes are easier to change

```cpp
class Something{
public:
    int m_value1;
    int m_value2;
    int m_value3;
};

int main(){
    Something something;
    something.m_value1 = 5;
    std::cout << something.m_value1 << '\n';
};
```

Nothing can be changed

```cpp
class Something{
private:
    int m_value1;    int m_value2;    int m_value3;

public:
    void setValue1(int value) { m_value1 = value; }
    int getValue1() { return m_value1; }
};

int main(){
    Something something;
    something.setValue1(5);
    std::cout << something.getValue1() << '\n';
```

Same printing result, but chance to change member data

Benefit: encapsulated classes are easier to change

```
class Something{
private:
    int m_value[3]; // note: we changed the implementation of this class!

public:
    // We have to update any member functions to reflect the new 
implementation
    void setValue1(int value) { m_value[0] = value; }
    int getValue1() { return m_value[0]; }
};
something.setValue1(5);
std::cout << something.getValue1() << '\n';
```

- Program using the code continues to work without any changes!
- They probably wouldn't even notice!

Benefit: encapsulated classes are easier to debug

- Often when a program does not work correctly, it is because one of our member variables has an incorrect value.

- If everyone is able to access the variable directly, tracking down which piece of code modified the variable can be difficult.

- However, if everybody has to call the same public function to modify a value, then you can simply breakpoint that function and watch as each caller changes the value until you see where it goes wrong.

Access functions

- Access functions typically come in two flavors: getters and setters.

```
class Date{
private:
    int m_month;    int m_day;
public:
    int getMonth() { return m_month; } // getter for month
    void setMonth(int month) { m_month = month; } // setter for month
    int getDay() { return m_day; } // getter for day
    void setDay(int day) { m_day = day; } // setter for day
};
```

Rule: Only provide access functions when it makes sense for th e user to be able to get or set a value directly

Constructors

Constructors

```
class Foo{
public:
    int m_x;
    int m_y;
};
```

```
int main(){
    Foo foo1 = { 4, 5 }; // initialization list
    Foo foo2 { 6, 7 }; // uniform initialization (C++11)
    return 0;
}
```

However, as soon as we make any member variables private, we're no lo nger able to initialize classes in this way.

It does make sense: if you can't directly access a variable (because it's p rivate), you shouldn't be able to directly initialize it. => constructor

Constructors

- A **constructor** is a special kind of class member function that is automatically called when an object of that class is instantiated.
  - Constructors should always have the **same name** as the class (with the same capitalization)
  - Constructors have **no return** type (not even void)

Default constructors

- A constructor that takes no parameters (or has parameters that all have default values)

```
class Fraction{
private:
     int m_numerator;    int m_denominator;
public:
    Fraction(){ // default constructor
        m_numerator = 0;
        m_denominator = 1;
    }
    int getNumerator() { return m_numerator; }
};
Fraction frac; // Since no arguments, calls Fraction() default constructor
std::cout << frac.getNumerator() << "/" << frac.getDenominator() << '\n';
```

Direct and uniform initialization using constructors with parameters

```
public:
    Fraction() { // default constructor
        m_numerator = 0;
        m_denominator = 1;
    }
  // Constructor with two parameters, one parameter having a default value
    Fraction(int numerator, int denominator=1) {
        assert(denominator != 0);
        m_numerator = numerator;
        m_denominator = denominator;
    }
```

```
int x(5);
Fraction fiveThirds(5, 3);
Fraction six(6);
```

Copy initialization using equals with classes

```
int x = 6; // Copy initialize an integer

Fraction six = Fraction(6); // Copy initialize a Fraction, will call Fraction(6, 1)

Fraction seven = 7; // Copy initialize a Fraction.  The compiler will try to find a way to convert 7 to a Fraction, which will invoke the Fraction(7, 1) constructor.
```

- less efficient
- *Rule: Do not copy initialize your classes*

Reducing your constructors

```
Fraction() { // default constructor
     m_numerator = 0;    m_denominator = 1;
}
Fraction(int numerator, int denominator=1) {
    assert(denominator != 0);
    m_numerator = numerator;
    m_denominator = denominator;
}
```

```
Fraction(int numerator=0, int denominator=1) {
    assert(denominator != 0);
    m_numerator = numerator;
    m_denominator = denominator;
}
```

Reducing your constructors

```
Fraction(int numerator=0, int denominator=1){
    assert(denominator != 0);
    m_numerator = numerator;
    m_denominator = denominator;
}

Fraction default; // will call Fraction(0, 1)
Fraction six(6); // will call Fraction(6, 1)
Fraction fiveThirds(5,3); // will call Fraction(5, 3)
```

Classes without default constructors

```
class Date{
private:
    int m_year;    int m_month;    int m_day;
// No default constructor provided, so C++ creates an empty one for u
s
// Because no other constructors exist, this provided constructor wil
l be public
};

Date date; // calls default constructor that does nothing
// date's member variables are uninitialized
// Who knows what date we'll get?
```

- if you do have other non-default constructors in your class, but no default constructor, C++ will not create an empty default constructor for you

```
class Date{
private:    int m_year;    int m_month;    int m_day;
public:
    Date(int year, int month, int day){ // not a default constructor
        m_year = year;    m_month = month;    m_day = day; }
    // No default constructor provided
};

Date date; // error: Can't instantiate object because default constructor doesn't exist
Date today(2020, 10, 14); // today is initialized to Oct 14th, 2020
```

Quiz time - Write a class named Ball.

- **Ball should have two private member variables with default values: m_color (“Black”) and m_radius (10.0).**
- **Ball should provide constructors to set only m_color, set only m_radius, set both, or set neither value.**
- **do not use default parameters for your constructors.**
- **Also write a function to print out the color and radius of the ball.**
- The following sample program should compile:

```
Ball def; def.print();
Ball blue("blue"); blue.print();
Ball twenty(20.0); twenty.print();
Ball blueTwenty("blue", 20.0); blueTwenty.print();
```

color: black, radius: 10
color: blue, radius: 10
color: black, radius: 20
color: blue, radius: 20

Quiz 2

- Update your answer to the previous question to use constructors with default parameters. Use as few constructors as possible.

Quiz 3

- What happens if you don’t declare a default constructor?

  - If you haven’t defined any other constructors, the compiler will create an empty public default constructor for you.
  - This means your objects will be instantiable with no parameters.

  - If you have defined other constructors (default or otherwise), the compiler will not create a default constructor for you.
  - Assuming you haven’t provided a default constructor yourself, your objects will not be instantiable with no parameters.

Constructor member initializer lists

```
class Something{
private:
    int m_value1;     char m_value3;

public:
    Something()
    {
        // These are all assignments, not initializations
        m_value1 = 1.0;          m_value3 = 'c';
    }
};
```

```
int m_value1;
double m_value2;
char m_value3;

m_value1 = 1.0;
m_value2 = 2.2;
m_value3 = 'c';
```

Constructor member initializer lists

```
class Something{
private:
    const int m_value;
public:
    Something(){
        m_value = 1; // error: const vars can not be assigned to
    }
};

const int m_value; // error: const vars must be initialized with a value
m_value = 5; //  error: const vars can not be assigned to
```

Member initializer lists

```
class Something{
private:
    int m_value1;
    double m_value2;
    char m_value3;
public:
    Something() : m_value1(1), m_value2(2.2), m_value3('c')
     // directly initialize our member variables
    {
    // No need for assignment here
    }
```

Overlapping and delegating constructors

```
class Foo
{
public:
    Foo() {
        // code to do A
    }

    Foo(int value) {
        // code to do A
        // code to do B
    }
};
```

Using a separate function

```
class Foo{
private:
    void DoA() { // code to do A }

public:
    Foo() {  DoA();   }

    Foo(int nValue) {
        DoA();
        // code to do B
    }
};
```

code duplication is kept to a minimum.

you may find yourself in the situation where you want to write a member function to re-initialize a class back to default values.

```
class Foo{
public:
    Foo(){ Init(); }

    Foo(int value){ Init();
        // do something with value
    }

    void Init() {   // code to init Foo }
};
```

Delegating constructors in C++11

```
class Employee{
private:
    int m_id;    std::string m_name;
public:
    Employee(int id, std::string name):
        m_id(id), m_name(name) {   }
    // All three of the following constructors use delegating constructors to minimize r
edundant code
    Employee() : Employee(0, "") { }
    Employee(int id) : Employee(id, "") { }
    Employee(std::string name) : Employee(0, name) { }
};
```

Destructors

Destructors

- A **destructor** is another special kind of class member function that is executed when an object of that class is destroyed.
- **Destructor naming**
  1) The destructor must have the same name as the class, preceded by a tilde (~).
  2) The destructor can not take arguments.
  3) The destructor has no return type.

```
class MyString
{
    ~MyString() { // destructor
      delete[] m_string;
    }
```

- only one destructor may exist per class
- like constructors, destructors should not be called explicitly
- destructors may safely call other member functions since the object isn’t destroyed until after the destructor executes.

Constructor and destructor timing

```
class Simple{
private:    int m_nID;
public:
    Simple(int nID) {
        std::cout << "Constructing Simple " << nID << '\n';
        m_nID = nID;
    }

    ~Simple(){std::cout << "Destructing Simple" << m_nID << '\n';}
    int getID() { return m_nID; }
};
```

Constructor and destructor timing

```cpp
int main(){
    // Allocate a Simple on the stack
    Simple simple(1);
    std::cout << simple.getID() << '\n';

    // Allocate a Simple dynamically
    Simple *pSimple = new Simple(2);
    std::cout << pSimple->getID() << '\n';
    delete pSimple;

    return 0;
} // simple goes out of scope here
```

Constructing Simple 1
1
Constructing Simple 2
2
Destructing Simple 2
Destructing Simple 1

A warning about the exit() function

- if you use the exit() function, your program will terminate and no destructors will be called.

- Be wary if you’re relying on your destructors to do necessary cleanup work (e.g. write something to a log file or database before exiting)

a hidden pointer named "this"

- "When a member function is called, how does C++ keep track of which object it was called on?"
- `simple.setID(2);`
- `setID(&simple, 2); // note that simple has been changed from an object prefix to a function argument!`
- `void setID(int id) { m_id = id; }`
- `void setID(Simple* const this, int id) { this->m_id = id; }`

Chaining objects

```
class Calc{
private:  int m_value;

public:
    Calc() { m_value = 0; }

    void add(int value) { m_value += value; }
    void sub(int value) { m_value -= value; }
    void mult(int value) { m_value *= value; }

    int getValue() { return m_value; }
};
```

Chaining objects

- `Calc calc;`
- `calc.add(5); // returns void`
- `calc.sub(3); // returns void`
- `calc.mult(4); // returns void`
- `std::cout << calc.getValue() << '\n';`

- `calc.add(5).sub(3).mult(4);`
- `Calc& add(int value) { m_value += value; return *this; }`
- `Calc& sub(int value) { m_value -= value; return *this; }`
- `Calc& mult(int value) { m_value *= value; return *this; }`

Class code and header files

- Defining member functions outside the class definition

```
#ifndef DATE_H
#define DATE_H

class Date{
private:    int m_year;    int m_month; …
public:
    Date(int year, int month, int day);

    void SetDate(int year, int month, int day);
    int getYear() { return m_year; }}; …
#endif
```

Class code and header files

- Date.cpp:

```
#include "Date.h"
Date::Date(int year, int month, int day){
    SetDate(year, month, day);
}

void Date::SetDate(int year, int month, int day){
    m_month = month;
    m_day = day;
    m_year = year;
}
```

a couple of downsides to expose implementation

- First, your class implementation code will be copied into every file that #includes it, and get recompiled there.
  - This can be slow, and will cause bloated file sizes.

- Second, if you change anything about the code in the header, then you'll need to recompile every file that includes that header.
  - This can have a ripple effect, where one minor change causes the entire program to need to recompile (which can be slow).
  - If you change the code in a .cpp file, only that .cpp file needs to be recompiled!

- **Default parameters**
  - Default parameters for member functions should be declared in the class declaration (in the **header** file), where they can be **seen** by whomever #includes the header.

Libraries

- Separating the class declarations and class implementation is very common for libraries that you can use to extend your program.

- #included iostream, string, ...
- No need to add iostream.cpp, string.cpp into your projects.
- the implementations for the classes that belong to the C++ standard library is contained in a precompiled file that is linked in at the link stage.

Libraries

- most 3rd party libraries provide only header files, along with a precompiled library file.

- reasons for this
  1) It’s faster to link a precompiled library than to recompile it every time

  2) a precompiled library can be distributed once, whereas compiled code gets compiled into every executable that uses it (inflating file sizes)

  3) intellectual property reasons (you don’t want people stealing your code).

Const class objects and member functions

Const objects

- `const int value1 = 5; // copy initialization`
- `const Date date2(2020, 10, 16); // initialize using parameterized constructor`
- `data2.setDay(5); // compiler error: violates const, even setDay() is public`

Const member functions

```
class Something{
public:
    int m_value ;
    int getValue() const;
    void resetValue() const { m_value = 0; } // compile error, c
onst functions can't change member variables.
};

int Something::getValue() const{
    return m_value;
}
```

- *rule: Make any member function that does not modify the state of the class object const*

Const references

```
class Date{
public:
    int getYear() { return m_year; }
};
```

- Compiling error:

```
void printDate(const Date &date){
    std::cout << date.getYear() << '\n';
}
```

- Solution:

```
int getYear() const { return m_year; }
```

Overloading const and non-const function

```
class Something
{
public:
    std::string m_value;

    const std::string& getValue() const { return m_value; } // getValue() for const objects
    std::string& getValue() { return m_value; } // getValue() for non-const objects
};
```

Static member variables

Static member variables

```
class Something{
public:     static int s_value;
};
int Something::s_value = 1;

int main(){
    Something first;    Something second;
    second.s_value = 2;
    std::cout << first.s_value << '\n';
    std::cout << second.s_value << '\n';
    return 0;
}
```

Static members are not associated with class objects

```
// note: we're not instantiating any objects of type Something
Something::s_value = 2;
std::cout << Something::s_value << '\n';
```

Defining and initializing static member variables

- If the class is defined in a .h file, the static member definition is usually placed in the associated code file for the class (e.g. Something.cpp).
- If the class is defined in a .cpp file, the static member definition is usually placed directly underneath the class.
- Do not put the static member definition in a header file (if that header file gets included more than once, you'll end up with multiple definitions, which will cause a compile error).
- exception: when the static member is of type const integer or const enum

```
class Whatever
{
public:
    static const int s_value = 4;
```

An example of static member variables

```
class Something{
private:
    static int s_idGenerator;    int m_id;

public:
    Something() { m_id = s_idGenerator++; }
};

int Something::s_idGenerator = 1;

int main(){
    Something first;    Something second;    Something third;
```

Static member functions

- static member variables are member variables that belong to the class rather than objects of the class.
- If the static member variables are public, we can access them directly using the class name and the scope resolution operator.

```
class Something{
private:     static int s_value;
};
int Something::s_value = 1; // initializer, this is okay even though s_value is private since it's a definition
int main(){
    // how do we access Something::s_value since it is private?
}  static int getValue() { return s_value; } // static member function
```

Static member functions have no *this pointer

- static member functions are not attached to an object

- static member functions can only access static member variables.
    - This is because non-static member variables must belong to a class object, and static member functions have no class object to work with!
-

Friend functions and classes

Why Friend functions?

- you may occasionally find situations where you will find you have classes and functions outside of those classes that need to work very closely together.

- For example, you might have a class that stores data, and a function (or another class) that displays the data on the screen.
- Although the storage class and display code have been separated for easier maintenance,
- the display code is really intimately tied to the details of the storage class.
- Consequently, there isn't much to gain by hiding the storage classes details from the display code.

Why Friend functions?

- In situations like this, there are two options:

1. using public functions
   - the storage class may have to expose functions for the display code that it doesn’t really want accessible to anybody else.
2. using friend classes and friend functions
   - lets the display code directly access all the private members and functions of the storage class,
   - while keeping everyone else out!

Friend functions

- a function that can access the private members of a class as though it were a member of that class.

```
class Accumulator{
private:
    int m_value;
...
    friend void reset(Accumulator &accumulator);
};
void reset(Accumulator &accumulator){    accumulator.m_value = 0;}
int main(){
    Accumulator acc;      reset(acc); // reset the accumulator to 0
```

- reset() is not a member function. It does not matter whether you declare the friend function in the private or public section of the class.

Multiple friends

- A function can be a friend of more than one class at the same time.

```
class Humidity;
class Temperature
{
    friend void printWeather(const Temperature &temperature, const Humidity &humidity);
};

class Humidity
{
    friend void printWeather(const Temperature &temperature, const Humidity &humidity);
};
```

Friend classes

```
//class Display;// this is not necessary.
class Storage{
private:    int m_nValue;    double m_dValue;
public:
     friend class Display;
};
class Display{
public:
    void displayItem(Storage &storage)    {
     std::cout << storage.m_nValue << '\n';
     }
};
```

Friend member functions

- Instead of making an entire class a friend, you can make a single member function a friend.

```
class Storage{
private:     int m_nValue;     double m_dValue;
public:
     friend void Display::displayItem(Storage& storage); // erro
r: Storage hasn’t seen the full declaration of class Display
};
```

Friend member functions

```
class Storage; // forward declaration for class Storage
class Display{
     void displayItem(Storage &storage); // forward declaration
above needed for this declaration line
};

class Storage{
private:    int m_nValue;    double m_dValue;
public:
     friend void Display::displayItem(Storage& storage); // OK
};
```

Quiz time

- http://www.learncpp.com/cpp-tutorial/813-friend-functions-and-classes/

Class - summary

- Encapsulation: properties and functions
- Access functions
- Constructors: default, non-default, system generated
  - member initializer lists
- Deconstructor
- Other
  - This
  - Const
  - Static
  - friend

Quiz time

1a) Write a class named Point2d. Point2d should contain two member variables of type double: m_x, and m_y, both defaulted to 0.0. Provide a constructor and a print function.

- The following program should run:

```
int main()
{
    Point2d first;
    Point2d second(3.0, 4.0);
    first.print();
    second.print();

    return 0;
}
```

1b) Now add a member function named distanceTo.

Given two points (x1, y1) and (x2, y2), the distance between them can be calculated as sqrt((x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2)).

The sqrt function lives in header cmath.

The following program should run:

```
Point2d first;
Point2d second(3.0, 4.0);
first.print();
second.print();
std::cout << "Distance between two points: " << first.distanceTo(second) << '\n';
```

- 1c) Change function distanceTo from a member function to a non-member friend function that takes two Points as parameters. Also rename it “distanceFrom”.
- The following program should run:

```
int main(){
    Point2d first;
    Point2d second(3.0, 4.0);
    first.print();
    second.print();
    std::cout << "Distance between two points: " << distanceFrom
(first, second) << '\n';

    return 0;
}
```

http://www.learncpp.com/cpp-tutorial/8-15-chapter-8-comprehensive-quiz/

- 3) Let’s create a random monster generator
- 4) rewrite the Blackjack games using classes!

Object-Based vs Object-Oriented programming

- Object-Based: **Encapsulation** (define composite datatypes using classes: fields + methods)

- Object-Oriented:
  - Encapsulation
  - **Inheritance**: reusing code between related types
  - **Polymorphism**: determining at runtime which functions to call on it based on its type