Program Design

-- Encapsulation

Junjie Cao @ DLUT

Summer 2022

https://github.com/jjcao-school/c

Access specifiers
访问说明符

Public vs private
公有 vs 私有

Public vs private access specifiers

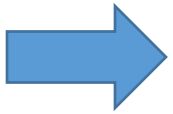
```
struct DateStruct{ // members are public by default
    int month; // public by default, can be accessed by anyone
    int day; // public by default, can be accessed by anyone
    int year; // public by default, can be accessed by anyone
};

int main(){
     DateStruct today { 2020, 10, 14 }; // use uniform initialization
    date.year= 2022;
    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;
};
```

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

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 = year;
   }
…
};
```

```
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共有接口: setDate(), print()

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

Quiz time

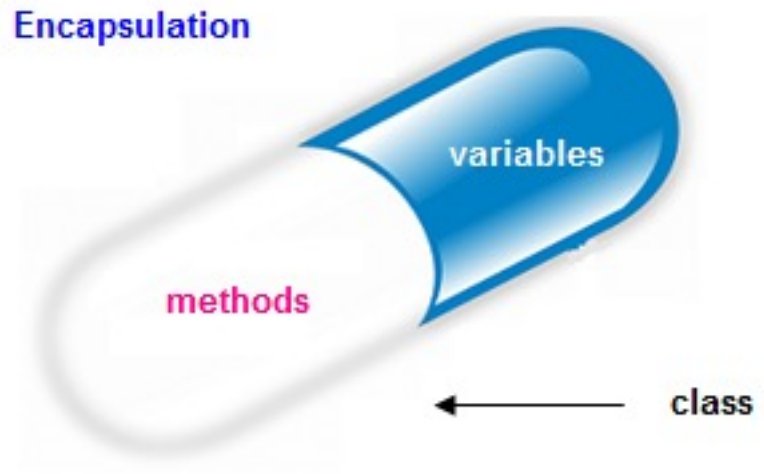
- 1a) What is a public member?
- 1b) What is a private member?
- 1c) What is an access specifier?
- 1d) How many access specifiers are there, and what are they?

Why make member variables private?

Encapsulation
封装

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?

```
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

```
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

```
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 implementa
tion
    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

Access functions in Python

```
class Dog(object):
    def __init__(self, age=0):
        self.humanAge = age

    @property
    def humanAge(self):
        return self._age

    @humanAge.setter
    def humanAge(self, value):
        self._age = value

    @property
    def dogAge(self):
        return self._age * 7
```

```
Cat cat;
cat.setHumanAge(5);
cout << cat.getAge();
```

```
d = Dog(age=4)
print(d.humanAge)
print(d.dogAge)
```