Lập trình hướng đối tượng (P.2)

Kỹ thuật Lập trình (CO1027)

Ngày 11 tháng 8 năm 2022

ThS. Trần Ngọc Bảo Duy

Khoa Khoa học và Kỹ thuật Máy tính

Trường Đại học Bách Khoa, ĐHQG-HCM

Tổng quan

ThS.
Trần Ngọc Bảo Duy

ĐẶC TÍNH ĐẶC BIỆT CỦA THÀNH VIÊN

ThS.
Trần Ngọc Bảo Duy

Thành viên có tính static

- Thuộc tính và phương thức đều có thể có tính `static` nếu ta đặt từ khóa `static` trước các khai báo của các thành viên.

```
class X {
public:
    static int a;
    static void foo() { }
};
```


ThS.
Trần Ngọc Bảo Duy

Thành viên có tính static

- Thuộc tính và phương thức đều có thể có tính `static` nếu ta đặt từ khóa `static` trước các khai báo của các thành viên.

```
class X {
public:
    static int a;
    static void foo() { }
};
```

- **Ý nghĩa chung**: thành viên được sử dụng chung cho toàn bộ tất cả các đối tượng của lớp.

ThS.
Trần Ngọc Bảo Duy

Thuộc tính có tính static

- Vùng nhớ cho nó không được tạo riêng cho mỗi đối tượng như trường hợp của thuộc tính non-static.
- Vùng nhớ cho nó được cấp 1 lần, chỉ một vùng, và dùng chung cho tất cả các đối tượng được tạo ra từ lớp đó.
- Vùng nhớ này được khởi tạo giá trị (bằng 0) trước khi đối tượng đầu tiên của lớp được tạo ra.
- Thời gian sống của vùng nhớ này là thời gian sống của toàn bộ chương trình.

ThS.
Trần Ngọc Bảo Duy

Thuộc tính non-static: Ví dụ

```
class Date {
private:
    int day, month, year;
public:
    Date(int day, int month,
        int year)
    {
        this->day = day;
        this->month = month;
        this->year = year;
    }

    void print() {}
};
int main() {
    Date d1(29, 3, 2021);
    Date d2(3, 12, 2021);
    Date d3(15, 12, 2021);

    d1.print();
    d2.print();
    d3.print();

    return 0;
}
```


ThS.
Trần Ngọc Bảo Duy

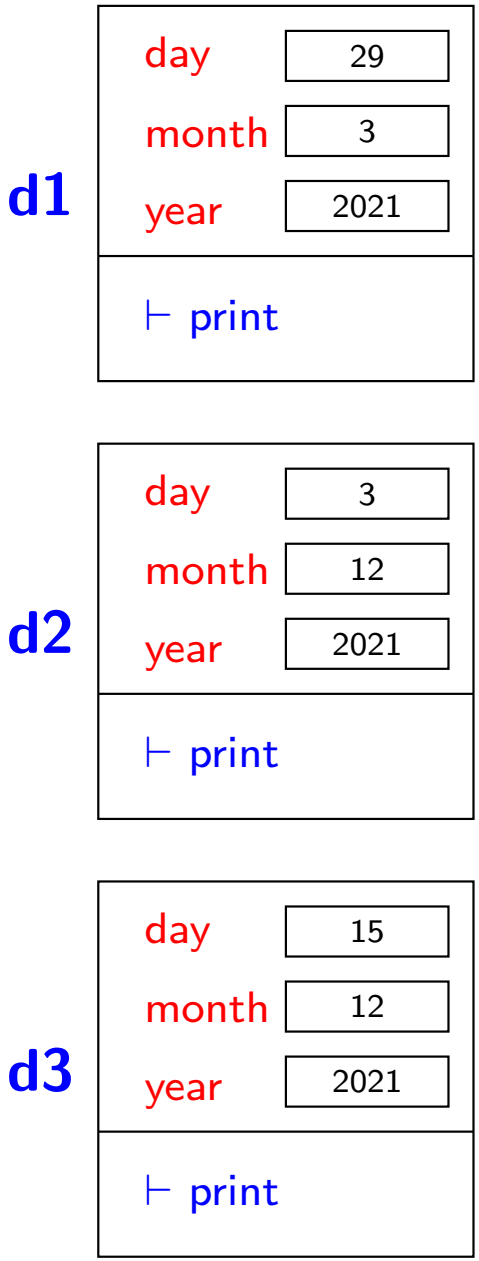
Thuộc tính non-static: Ví dụ

```
class Date {
private:
    int day, month, year;
public:
    Date(int day, int month,
        int year)
    {
        this->day = day;
        this->month = month;
        this->year = year;
    }

    void print() {}
};
int main() {
    Date d1(29, 3, 2021);
    Date d2(3, 12, 2021);
    Date d3(15, 12, 2021);

    d1.print();
    d2.print();
    d3.print();

    return 0;
}
```



ThS.
Trần Ngọc Bảo Duy

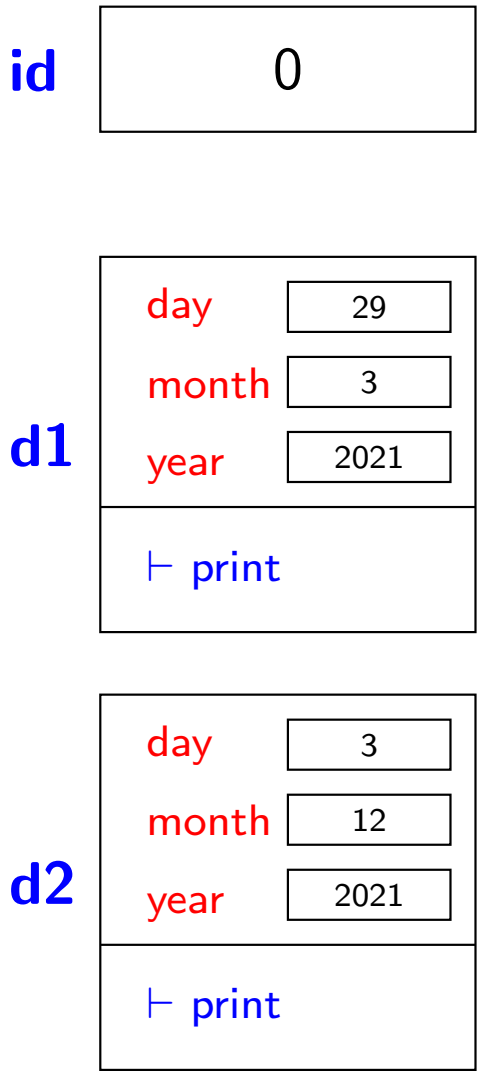
Thuộc tính static: Ví dụ

```
class Date {
private:
    int day, month, year;
public:
    static int id;
    Date(int day, int month,
        int year)
    {
        this->day = day;
        this->month = month;
        this->year = year;
    }

    void print() {}
    int getId() { return id++; }
};
int main() {
    Date d1(29, 3, 2021);
    Date d2(3, 12, 2021);

    cout << d1.getId();
    cout << d2.getId();

    return 0;
}
```



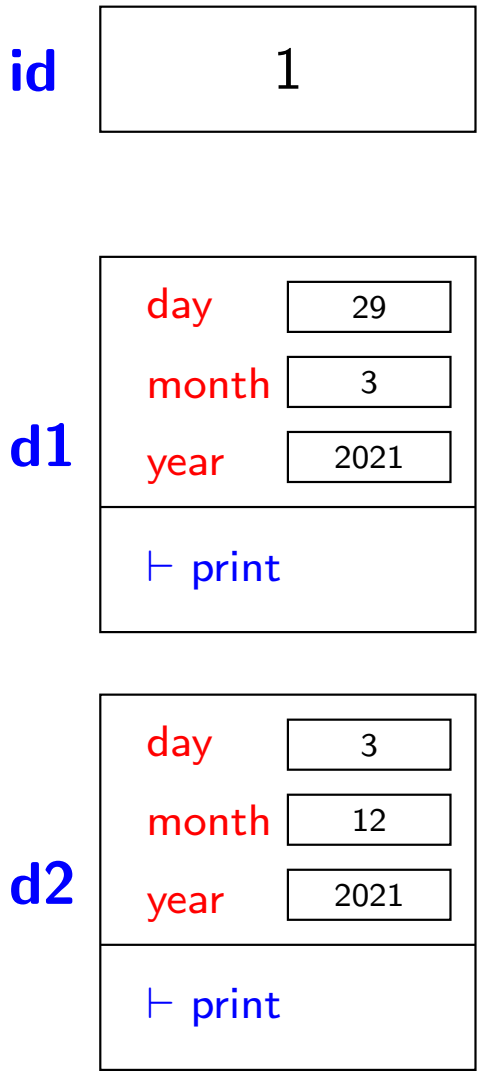
Thuộc tính static: Ví dụ

```
class Date {
private:
    int day, month, year;
public:
    static int id;
    Date(int day, int month,
        int year)
    {
        this->day = day;
        this->month = month;
        this->year = year;
    }

    void print() {}
    int getId() { return id++; }
};
int main() {
    Date d1(29, 3, 2021);
    Date d2(3, 12, 2021);

    cout << d1.getId();
    cout << d2.getId();

    return 0;
}
```



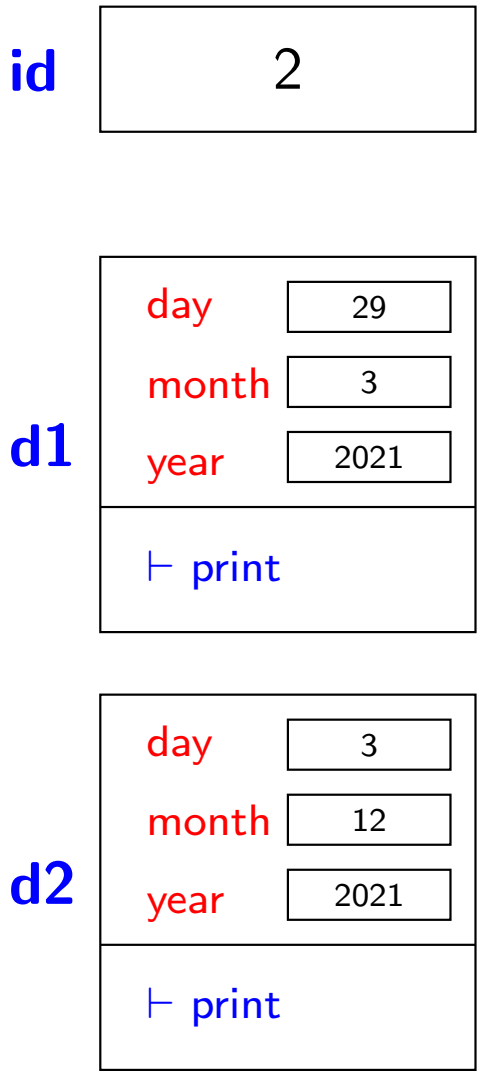
Thuộc tính static: Ví dụ

```
class Date {
private:
    int day, month, year;
public:
    static int id;
    Date(int day, int month,
        int year)
    {
        this->day = day;
        this->month = month;
        this->year = year;
    }

    void print() {}
    int getId() { return id++; }
};
int main() {
    Date d1(29, 3, 2021);
    Date d2(3, 12, 2021);

    cout << d1.getId();
    cout << d2.getId();

    return 0;
}
```



Phương thức có tính static

Định nghĩa

Phương thức có tính **static** là phương thức không của riêng bất kỳ đối tượng nào. Vì vậy:

1. Không thể gọi đến phương thức loại này bằng một đối tượng cụ thể.

ThS.
Trần Ngọc Bảo Duy

Phương thức có tính static

Định nghĩa

Phương thức có tính **static** là phương thức không của riêng bất kỳ đối tượng nào. Vì vậy:

1. Không thể gọi đến phương thức loại này bằng một đối tượng cụ thể.
2. Trong phương thức dạng này, con trỏ `this` không có giá trị, do đó việc dùng con trỏ `this` sẽ có lỗi.

ThS.
Trần Ngọc Bảo Duy

Phương thức có tính static

Định nghĩa

Phương thức có tính **static** là phương thức không của riêng bất kỳ đối tượng nào. Vì vậy:

1. Không thể gọi đến phương thức loại này bằng một đối tượng cụ thể.
2. Trong phương thức dạng này, con trỏ `this` không có giá trị, do đó việc dùng con trỏ `this` sẽ có lỗi.
3. Phương thức **static** không thể gọi trực tiếp các phương thức **non-static** khác.

ThS.
Trần Ngọc Bảo Duy

Phương thức có tính static

Định nghĩa

Phương thức có tính **static** là phương thức không của riêng bất kỳ đối tượng nào. Vì vậy:

1. Không thể gọi đến phương thức loại này bằng một đối tượng cụ thể.
2. Trong phương thức dạng này, con trỏ `this` không có giá trị, do đó việc dùng con trỏ `this` sẽ có lỗi.
3. Phương thức **static** không thể gọi trực tiếp các phương thức **non-static** khác.
4. Phương thức **static** có thể:
   - Truy xuất thuộc tính có tính **static**.
   - Gọi các phương thức có tính **static** khác.

ThS.
Trần Ngọc Bảo Duy

Thành viên có tính static: Minh họa

Với lớp Complex đã hiện thực:

- Người ta cần lưu trữ thêm thông tin về định dạng xuất ra màn hình khi in:
  1. Dạng đại số.
  2. Dạng lượng giác (số đo góc theo số đo độ).
  3. Dạng lượng giác (số đo góc theo radian).
- Giả định, định dạng này là thiết lập cho toàn bộ các đối tượng của lớp Complex kể từ thời điểm thiết lập định dạng.

ThS.
Trần Ngọc Bảo Duy

Thành viên có tính static: Minh họa

```
1 enum cformat { ALGEBRAIC, DEGREE, RADIAN};
2 class Complex {
3 public:
4     static cformat format;
5
6 // Other defined members
7 };
8
9 cformat Complex::format = ALGEBRAIC;
```


ThS.
Trần Ngọc Bảo Duy

Thành viên có tính static: Minh họa

```
1 enum cformat { ALGEBRAIC, DEGREE, RADIAN};
2 class Complex {
3 public:
4     static cformat format;
5
6 // Other defined members
7 };
8
9 cformat Complex::format = ALGEBRAIC;
```

- Dòng 4: Khai báo bên trong lớp, dùng từ khoá `static`.
- Dòng 9: Khởi động bên ngoài phạm vi lớp, dùng tên đầy đủ, sử dụng tiếp đầu ngữ: `Complex::`, không dùng từ khoá `static`.

ThS.
Trần Ngọc Bảo Duy

Thành viên có tính static: Minh họa

```
1  enum cformat { ALGEBRAIC, DEGREE, RADIAN};
2  class Complex {
3  public:
4      static cformat format;
5  // Other defined members
6  };
7
8  cformat Complex::format = ALGEBRAIC;
9
10 int main() {
11     Complex c1;
12     c1.format = DEGREE;
13
14     Complex c2;
15     cout << (c2.format == DEGREE);
16
17     return 0;
18 }
```


ThS.
Trần Ngọc Bảo Duy

Thành viên có tính static: Minh họa

```
1  enum cformat { ALGEBRAIC, DEGREE, RADIAN};
2  class Complex {
3  public:
4      static cformat format;
5  // Other defined members
6  };
7
8  cformat Complex::format = ALGEBRAIC;
9
10 int main() {
11     Complex c1;
12     c1.format = DEGREE;
13
14     Complex c2;
15     cout << (c2.format == DEGREE);
16
17     return 0;
18 }
```

Cả hai đối tượng `c1` và `c2` dùng chung vùng nhớ `format`. Do vậy, khi `c1` thay đổi thì `c2` cũng nhìn thấy sự thay đổi đó.

ThS.
Trần Ngọc Bảo Duy

Thành viên có tính static: Minh họa

```
1  enum cformat { ALGEBRAIC, DEGREE, RADIAN};
2  class Complex {
3  public:
4      static cformat format;
5  // Other defined members
6  };
7
8  cformat Complex::format = ALGEBRAIC;
9
10 int main() {
11     Complex c1;
12     Complex::format = DEGREE;
13
14     Complex c2;
15     cout << (Complex::format == DEGREE);
16
17     return 0;
18 }
```


ThS.
Trần Ngọc Bảo Duy

Thành viên có tính static: Minh họa

```
1  enum cformat { ALGEBRAIC, DEGREE, RADIAN};
2  class Complex {
3  public:
4      static cformat format;
5  // Other defined members
6  };
7
8  cformat Complex::format = ALGEBRAIC;
9
10 int main() {
11     Complex c1;
12     Complex::format = DEGREE;
13
14     Complex c2;
15     cout << (Complex::format == DEGREE);
16
17     return 0;
18 }
```

Một số trình biên dịch sẽ **warning** khi truy xuất thành viên có tính `static` bằng cách sử dụng như một thành viên của một đối tượng. Vì vậy, hãy sử dụng tiếp đầu ngữ `Complex::` (<Tên lớp>::).

ThS.
Trần Ngọc Bảo Duy

Thành viên có tính static: Minh họa

Giả sử, ta cần cung cấp phương thức để lấy số phức liên hợp. Có thể đặt ra các thiết kế như sau:

1. Gọi được phương thức `conj()` trên đối tượng `c`, tức là `c.conj()`. Phương thức `conj()` đó thực hiện tính liên hợp trên `c` và trả về chính `c`.

```
1 class Complex {
2 public:
3     Complex conj() {
4         this->img = -this->img;
5         return *this;
6     }
7 };
```

   Cách này không dùng `static`, nhưng làm thay đổi `c`.

ThS.
Trần Ngọc Bảo Duy

Thành viên có tính static: Minh họa

Giả sử, ta cần cung cấp phương thức để lấy số phức liên hợp. Có thể đặt ra các thiết kế như sau:

2. Gọi được phương thức `conj()` trên đối tượng `c`, tức là `c.conj()`. Phương thức `conj()` đó tạo ra một số phức trung gian, liên hợp số phức này và trả về số phức trung gian đó.

```
1 class Complex {
2 public:
3     Complex conj() {
4         Complex c2(this->real,
5                    -this->img);
6         return c2;
7     }
8 };
```

Cách này không dùng `static` và đồng thời cũng không thay đổi `c`.

ThS.
Trần Ngọc Bảo Duy

Thành viên có tính static: Minh họa

Giả sử, ta cần cung cấp phương thức để lấy số phức liên hợp. Có thể đặt ra các thiết kế như sau:

3. Nếu bản thiết kế yêu cầu phải cung cấp cả hai tình huống: `c` thay đổi và `c` không bị thay đổi. Lúc này phải dùng đến `static`.
   - Phương thức non-static như (1): `Complex conj()` là nhằm liên hợp và trả về chính số phức hiện tại.
   - Phương thức static: `Complex conj(const Complex&)`: Phương thức này tạo ra số phức trung gian, thực hiện liên hợp và trả về số phức trung gian đó.

ThS.
Trần Ngọc Bảo Duy

Thành viên có tính static: Minh họa

```
1  class Complex {
2  public:
3      Complex conj() {
4          this->img = -this->img;
5          return *this;
6      }
7
8      static Complex conj(const Complex& rhs) {
9          Complex c2(rhs.real,
10                     -rhs.img);
11         return c2;
12     }
13 };
14 int main() {
15     Complex c1(1, 3);
16     c1.conj();
17     c1.print();
18
19     Complex c2(1, 4);
20     Complex c3 = Complex::conj(c2);
21     c3.print();
22
23     return 0;
24 }
```


ThS.
Trần Ngọc Bảo Duy

Thuộc tính có tính const

Định nghĩa

Thuộc tính có tính **const** sẽ không cho phép sự thay đổi giá trị kể từ sau khi khởi động giá trị của nó, đòi hỏi khi khởi động giá trị của thuộc tính **const** bằng cách khởi động trên cùng hàm khởi tạo.

ThS.
Trần Ngọc Bảo Duy

Thuộc tính có tính const

Định nghĩa

Thuộc tính có tính **const** sẽ không cho phép sự thay đổi giá trị kể từ sau khi khởi động giá trị của nó, đòi hỏi khi khởi động giá trị của thuộc tính **const** bằng cách khởi động trên cùng hàm khởi tạo.

Ứng dụng: Giúp cho người lập trình duy trì một giá trị là hằng số suốt trong thời gian sống của đối tượng và gắn với mỗi đối tượng.

ThS.
Trần Ngọc Bảo Duy

Phương tính có tính const

Định nghĩa

Phương tính có tính **const** sẽ không thể thay đổi bất kỳ thuộc tính thành viên nào của lớp, nghĩa là, phương thức chỉ đọc giá trị của các thuộc tính thành viên. Với một ngoại lệ, phương thức dạng này có thể thay đổi thuộc tính thành viên nếu biến đó khai báo tường minh với từ khoá **mutable**.

ThS.
Trần Ngọc Bảo Duy

Phương tính có tính const

Định nghĩa

Phương tính có tính **const** sẽ không thể thay đổi bất kỳ thuộc tính thành viên nào của lớp, nghĩa là, phương thức chỉ đọc giá trị của các thuộc tính thành viên. Với một ngoại lệ, phương thức dạng này có thể thay đổi thuộc tính thành viên nếu biến đó khai báo tường minh với từ khoá **mutable**.

Ứng dụng: Trong một dự án có nhiều người tham gia, hạn chế truy xuất bằng cách dùng **const** như thế này sẽ tăng tính toàn vẹn dữ liệu, đảm bảo dữ liệu luôn luôn trong trạng thái đúng.

ThS.
Trần Ngọc Bảo Duy

Thành viên có tính hằng: Minh họa

```
1  class X {
2  private:
3      int nonconst_value;
4      const int const_value;
5      int mutable mutable_value;
6  public:
7      X(int value):
8          const_value(value),
9          nonconst_value(100),
10         mutable_value(200) {}
11
12     int process() const {
13         int rs;
14         rs = const_value + nonconst_value;
15         // const_value = 100;
16         // nonconst_value = 100;
17         mutable_value = 100;
18         rs += mutable_value;
19         return rs;
20     }
21 };
```

- Dòng 3: Khai báo một thuộc tính không có tính `const`, cũng không có tính `mutable`.
- Dòng 4: Khai báo một thuộc tính có tính `const`.
- Dòng 5: Khai báo một thuộc tính có tính `mutable`.

ThS.
Trần Ngọc Bảo Duy

Thành viên có tính hằng: Minh họa

```
1  class X {
2  private:
3      int nonconst_value;
4      const int const_value;
5      int mutable mutable_value;
6  public:
7      X(int value):
8          const_value(value),
9          nonconst_value(100),
10         mutable_value(200) {}
11
12     int process() const {
13         int rs;
14         rs = const_value + nonconst_value;
15         // const_value = 100;
16         // nonconst_value = 100;
17         mutable_value = 100;
18         rs += mutable_value;
19         return rs;
20     }
21 };
```

- Dòng 12: Khai báo một phương thức có tính `const`, từ khóa này đặt sau danh sách tham số và trước thân của định nghĩa hàm.

ThS.
Trần Ngọc Bảo Duy

Thành viên có tính hằng: Minh họa

```
1  class X {
2  private:
3      int nonconst_value;
4      const int const_value;
5      int mutable mutable_value;
6  public:
7      X(int value):
8          const_value(value),
9          nonconst_value(100),
10         mutable_value(200) {}
11
12     int process() const {
13         int rs;
14         rs = const_value + nonconst_value;
15         // const_value = 100;
16         // nonconst_value = 100;
17         mutable_value = 100;
18         rs += mutable_value;
19         return rs;
20     }
21 };
```

- Dòng 17: Dù phương thức có tính `const` nhưng vẫn có thể thay đổi giá trị của thuộc tính có tính `mutable`.

ThS.
Trần Ngọc Bảo Duy

Thành viên có tính hằng: Minh họa

```
1  class X {
2  private:
3      int nonconst_value;
4      const int const_value;
5      int mutable mutable_value;
6  public:
7      X(int value):
8          const_value(value),
9          nonconst_value(100),
10         mutable_value(200) {}
11
12     int process() const {
13         int rs;
14         rs = const_value + nonconst_value;
15         // const_value = 100;
16         // nonconst_value = 100;
17         mutable_value = 100;
18         rs += mutable_value;
19         return rs;
20     }
21 };
```

- Dòng 15: Nếu có dòng này thì chương trình sẽ bị lỗi, vì thuộc tính có tính `const` thì dù phương thức có tính `const` hay không thì cũng không thể thay đổi được giá trị khi đã khởi động trong hàm khởi tạo.

ThS.
Trần Ngọc Bảo Duy

Thành viên có tính hằng: Minh họa

```
1  class X {
2  private:
3      int nonconst_value;
4      const int const_value;
5      int mutable mutable_value;
6  public:
7      X(int value):
8          const_value(value),
9          nonconst_value(100),
10         mutable_value(200) {}
11
12     int process() const {
13         int rs;
14         rs = const_value + nonconst_value;
15         // const_value = 100;
16         // nonconst_value = 100;
17         mutable_value = 100;
18         rs += mutable_value;
19         return rs;
20     }
21 };
```

- Dòng 16: Nếu có dòng này thì chương trình sẽ bị lỗi, vì thuộc tính không `const`, không `mutable` sẽ không thể bị thay đổi giá trị trong phương thức có tính `const`.

ThS.
Trần Ngọc Bảo Duy

QUAN HỆ BẠN BÈ

ThS.
Trần Ngọc Bảo Duy

Quan hệ bạn bè (friendship)

- Giả sử lớp **ClassA** và **ClassB** là hai lớp độc lập, không có quan hệ thừa kế. Các phương thức thành viên của **ClassA** không thể truy cập các thành viên (thuộc tính & phương thức) có tính **private** và **protected** của **ClassB** và ngược lại.
- Trong dự án, có một số hàm hoặc lớp cần cho các lớp khác truy xuất các thành viên (thuộc tính & phương thức) có tính **private** và **protected** của nó. Vì lý do:
  1. Code cho đơn giản.
  2. Tránh gọi hàm nhiều lần.
  3. Các lớp này được thiết kế cẩn thận và nằm trong cùng dự án nên vẫn có thể kiểm soát sự toàn vẹn dữ liệu được.

$\Rightarrow$ Cơ chế này được gọi là **quan hệ bạn bè (friendship)**.

ThS.
Trần Ngọc Bảo Duy

Các dạng quan hệ bạn bè

1. **Hàm friend**: Cho phép một hàm không phải thành viên của lớp **ClassX** vẫn có thể truy cập các thành viên (thuộc tính & phương thức) có tính **private** và **protected** của **ClassX**.

2. **Lớp friend**: Cho phép tất cả các phương thức thành viên của **ClassY** có thể truy cập các thành viên (thuộc tính & phương thức) có tính **private** và **protected** của **ClassX**.

ThS.
Trần Ngọc Bảo Duy

Hàm bạn bè

Với lớp **Complex**, hãy bổ sung toán tử cộng/ trừ/ nhân với số nguyên/ số thực, sao cho đoạn mã sau đây thực thi thành công:

```
1 Complex a(1, 2);
2 Complex b = a + 1.2;
3 Complex c = a + 2;
4 Complex d = 1.2 + a;
5 Complex e = 2 + a;
```


ThS.
Trần Ngọc Bảo Duy

Hàm bạn bè

```
1 Complex a(1, 2);
2 Complex b = a + 1.2;
3 Complex c = a + 2;
4 Complex d = 1.2 + a;
5 Complex e = 2 + a;
```

Để dòng 2 - 3 thực thi thành công, ta chỉ cần định nghĩa thêm các phương thức sau đây:

```
1 Complex operator+(const int& rhs);
2 Complex operator+(const double& rhs);
3 Complex operator+(const float& rhs);
```

vì lúc này, biến bên trái toán tử là thuộc lớp **Complex**.

ThS.
Trần Ngọc Bảo Duy

Hàm bạn bè

```
1 Complex a(1, 2);
2 Complex b = a + 1.2;
3 Complex c = a + 2;
4 Complex d = 1.2 + a;
5 Complex e = 2 + a;
```

Tuy nhiên, để dòng 4 - 5 thì không thể dùng các định nghĩa này vì bên bên trái toán tử là thuộc kiểu `int` hoặc `float` hoặc `double`, ... Muốn vậy, ta phải thực hiện quá tải các hàm (function overloading) nằm ngoài lớp `Complex`:

```
1 Complex operator+(int lhs, const Complex& rhs);
2 Complex operator+(double lhs, const Complex& rhs);
3 Complex operator+(float lhs, const Complex& rhs);
```


ThS.
Trần Ngọc Bảo Duy

Hàm bạn bè: Minh họa

```
1  class Complex {
2  public:
3      Complex operator+(const double& rhs) {
4          return Complex(this->real + rhs, this->img);
5      }
6  };
7
8  Complex operator+(double lhs, const Complex& rhs) {
9      Complex res;
10     res.setReal(lhs + rhs.getReal());
11     res.setImg(rhs.getImg());
12     return res;
13 }
```

Việc truy cập bằng các getter/setter như dòng 10 - 11 khá phức tạp và dài dòng, nên ta có thể thiết lập hàm trên thành hàm bạn bè của lớp `Complex`.

ThS.
Trần Ngọc Bảo Duy

Hàm bạn bè: Minh họa

```
1  class Complex {
2  public:
3      Complex operator+(const double& rhs) {
4          return Complex(this->real + rhs, this->img);
5      }
6
7      friend Complex operator+(double lhs,
8                               const Complex& rhs);
9  };
10
11 Complex operator+(double lhs, const Complex& rhs) {
12     Complex res;
13     res.real = lhs + rhs.real;
14     res.img = rhs.img;
15     return res;
16 }
```

Nhờ thiết lập quan hệ bạn bè này, trong thân hàm (dòng 12 - 15) ta có thể truy xuất các thành viên có tính `private` và `protected`.

ThS.
Trần Ngọc Bảo Duy

Nhập xuất với toán tử » và «

Để toán tử >> và << của `cin` và `cout` hoạt động trên lớp định nghĩa, ta sử dụng hàm bạn bè với prototype sau:

```
ostream& operator<<(ostream &output, const Complex& c);
istream& operator>>(istream &output, const Complex& c);
```


ThS.
Trần Ngọc Bảo Duy

Lớp bạn bè

Tương tự với hàm bạn bè, ta có thể định nghĩa lớp bạn bè với công dụng cho phép lớp bạn truy xuất được các thành viên có tính `private` và `protected` của ta.

Giả sử, số phức có các ứng dụng lớn trong ngành Kỹ thuật điện, người ta có thể giải các bài toán điện xoay chiều bằng số phức. Ta có lớp điện áp **Voltage** để mô tả điện áp giữa hai điểm trên mạch điện xoay chiều:

```
1 class Voltage;
2
3 class Complex {
4 public:
5     friend class Voltage;
6 };
7 class Voltage {
8     Complex value;
9 };
```


ThS.
Trần Ngọc Bảo Duy

TỔ CHỨC MÃ NGUỒN CHO CÁC LỚP

ThS.
Trần Ngọc Bảo Duy

Tổ chức mã nguồn cho các lớp

- Một lớp có nhiều hàm thành viên, phần thân những hàm này có thể để chung với phần mô tả của lớp, như ở các ví dụ trước.
- Tuy vậy, thường các dự án lớn (đặc biệt là dạng thư viện phần mềm), mỗi lớp được tổ chức thành 2 tập tin riêng biệt:
  1. Tập tin header (*.h): Tập tin này chứa **phần mô tả** (description, declaration) cho lớp.
  2. Tập tin source (*.cpp): Tập tin này chứa **phần định nghĩa** (definition) cho lớp.

ThS.
Trần Ngọc Bảo Duy

Tổ chức mã nguồn cho các lớp: Minh họa

File **Date.h**: Chứa phần định nghĩa (hiện thực) cho lớp.

```
1  #ifndef DATE_H
2  #define DATE_H
3  class Date{
4  public:
5       int getDay(void);
6       void setDay(int newDay);
7       int getMonth(void);
8       void setMonth(int newMonth);
9       int getYear(void);
10      void setYear(int newYear);
11      int compareTo(const Date& aDate);
12      int distanceTo(const Date& aDate);
13      Date();
14      Date(const Date& oldDate);
15      void date(int day, int month, int year);
16 private:
17      int day;
18      int month;
19      int year;
20 };
21 #endif
```


Tổ chức mã nguồn cho các lớp: Minh họa

File **Date.cpp**: Chứa phần mô tả (khai báo) cho lớp.

```
1  #include "Date.h"
2  int Date::getDay(void){
3      return day;
4  }
5  void Date::setDay(int newDay){
6      day = newDay;
7  }
8  int Date::getMonth(void){
9      return month;
10 }
11 void Date::setMonth(int newMonth){
12     month = newMonth;
13 }
14 int Date::getYear(void){
15     return year;
16 }
17 void Date::setYear(int newYear){
18     year = newYear;
19 }
20
21 // Other methods
```

Dùng toán tử :: khi hàm định nghĩa bên ngoài phần mô tả của lớp. Các hàm có tên đầy đủ có dạng: <Tên lớp>::<Tên hàm>.

ThS.
Trần Ngọc Bảo Duy