Bài 8: **Lập trình hướng đối tượng**

Giới thiệu C++

Sơ lược về C++

- Bổ sung các tính năng mới so với C:
  - Hướng đối tượng (OOP)
  - Lập trình khái quát (template)
  - Nhiều thay đổi nhỏ khác
- Một số thay đổi nhỏ:
  - File mã nguồn thường dùng đuôi .cpp
  - Hàm main() có thể có kiểu trả về là void:
    - `void main() { … }`
  - Dùng `//` để chú thích đến hết dòng:
    - `dien_tich = PI*r*r;  // PI = 3.14`
  - Có sẵn kiểu bool và các giá trị false, true:
    - `bool b1 = true, b2 = false;`
  - Biến, hằng trong C++ có thể được khai báo ở bất kỳ đâu trong hàm (không giới hạn ở đầu hàm như C), kể cả trong vòng lặp for
  - Phép chuyển kiểu có thể viết như cú pháp gọi hàm: `int(5.32)`
  - Không cần thêm các từ khoá enum, struct, union khi khai báo biến

Vài khái niệm mới ít nhỏ hơn…

- Kiểu tham chiếu (reference) : có bản chất con trỏ
  - ```
    int a = 5;
    int& b = a;
    b = 10;    // → a = 10
    ```
  - ```
    int& foo(int& x)
      { x = 2; return x; }
    int y = 1;
    foo(y);            // → y = 2
    foo(y) = 3;        // → y = 3
    ```
- Namespace
  - ```
    namespace ABC {
       int x;
       int setX(int y)  { x = y; }
    }

    ABC::setX(20);
    int z = ABC::x;

    using namespace ABC;
    setX(40);
    ```
    ```

Vài khái niệm mới ít nhỏ hơn… *(tiếp)*

- Cấp phát bộ nhớ động
  - Dùng các toán tử new và new[] để cấp phát
    - ```
      int* a = new int;
      float* b = new float(5.23f);
      long* c = new long[5];
      ```
  - Toán tử delete và delete[] để huỷ
    - ```
      delete a;
      delete[] c;
      ```
  - Chú ý: không được dùng lẫn lộn malloc()/free() với new/delete:
    - Cấp phát bằng malloc() thì phải dùng free() để huỷ
    - Cấp phát bằng new thì phải dùng delete để huỷ
- Định nghĩa chồng hàm (hàm cùng tên, khác tham số):
  - ```
    int sum(int a, int b)                          {...}
    int sum(int a, int b, int c)                   {...}
    double sum(double a, double b)                 {...}
    double sum(double a, double b, double c)       {...}
    ```
- Xử lý ngoại lệ try ... catch: tự tìm hiểu thêm
    ```

Chương trình C++ đầu tiên

- Chương trình ví dụ:

```
#include <iostream>
using namespace std;
void main() {
   int n;
   cout << "Nhap n: ";
   cin >> n;
   cout << "n = " << n << endl;
}
```

- Xuất/nhập với C++:
  - Dùng thư viện iostream
  - “cout <<” dùng để in ra stdout (màn hình)
  - “cin >>” dùng để đọc dữ liệu từ stdin (bàn phím)
  - Các đối tượng của thư viện C++ nằm trong một namespace có tên std. Nếu không khai báo “using …” thì phải viết đầy đủ std::cout, std::cin và std::endl

Lớp và đối tượng (*classes and objects*)

Khái niệm

- Từ thực tiễn:
  - Đối tượng (object) là những vật, sự việc, thực thể,… bao gồm các thuộc tính (property) đặc trưng cho nó và có thể thực hiện các tác vụ (operation) nhất định
    - Mỗi sinh viên là một đối tượng với các thuộc tính: tên, tuổi, khoa, lớp, khoá,… và có thể có các tác vụ: học bài, làm bài tập, nghe giảng, làm bài kiểm tra,…
    - Mỗi chiếc điện thoại là một đối tượng với các thuộc tính: số SIM, model, kích thước,… và có các tác vụ: gọi số, nhắn tin, nghe cuộc gọi tới, từ chối cuộc gọi,…
  - Lớp (class) là phần mô tả các thuộc tính và các tác vụ tương ứng của đối tượng
    - Có thể hiểu một cách đơn giản: mỗi sinh viên là một đối tượng trong khi khái niệm sinh viên là một lớp, tương tự với mỗi chiếc điện thoại và khái niệm điện thoại

Khái niệm *(tiếp)*

- … đến lập trình:
  - Lớp là kiểu dữ liệu mở rộng của kiểu cấu trúc (struct). Ngoài các trường (field) tương ứng cho thuộc tính của đối tượng, các phương thức (method) tương tự như các hàm được bổ sung thêm tương ứng với các tác vụ có thể thực hiện của đối tượng
  - Đối tượng là một biến được khai báo với kiểu là lớp đã được định nghĩa
- Từ lập trình cấu trúc: lấy hàm làm trung tâm
  - ```
    struct SinhVien {
       char ten[20];
       int khoa;
    };
    void len_lop(SinhVien& sv, int phong_hoc)   { ... }
    void kiem_tra(SinhVien& sv)                 { ... }

    SinhVien sv = { ... };
    len_lop(sv, 103);
    kiem_tra(sv);
    ```
    ```

Khái niệm *(tiếp)*

- … đến lập trình hướng đối tượng:

```
struct SinhVien {
    char ten[20];
    int khoa;

    void len_lop(int phong_hoc)        { ... }
    void kiem_tra()                    { ... }
};

SinhVien sv = { ... };
sv.len_lop(103);
sv.kiem_tra();
```

  - Các hàm (function) trở thành phương thức (method) của lớp và có thể truy cập trực tiếp các thuộc tính (biến thành phần) của đối tượng gọi
  - Đối tượng (biến) trở thành chủ thể của phương thức (hàm) được gọi chứ không còn được truyền như tham số → lấy đối tượng làm trung tâm của việc lập trình

Phạm vi của các thành phần

- Các thuộc tính hoặc phương thức public có thể truy cập được từ ngoài, private thì chỉ giới hạn gọi trong lớp đó
  - ```
    struct SinhVien {
    public:
       char ten[20];
       void len_lop(int lop)      { ... }
    private:
       int khoa;
       void kiem_tra()            { ... }
    };

    strcpy(sv.ten, "Nguyen Hung Long"); // OK
    sv.len_lop(103);          // OK
    sv.khoa = 50;             // lỗi
    sv.kiem_tra();            // lỗi
    ```
    ```

class và struct

- Trong C++, để tránh nhầm lẫn với các kiểu cấu trúc thông thường, dùng từ khoá class để khai báo lớp:
  - `class SinhVien { ... };`
- class và struct chỉ khác nhau về phạm vi mặc định của các thuộc tính: struct là public còn class là private
  - ```
    struct A {
       int a; // public
    };
    class B {
       int b; // private
    };
    ```
- Thông thường, nên khai báo các biến thành phần là private, truy xuất thông qua các phương thức của hàm → ẩn dữ liệu, khả năng đóng gói cao hơn
    ```

Ví dụ khai báo và sử dụng lớp

```
#include <iostream>
class Circle {
private:
    double r;
public:
    void setR(double rr)        { r = rr; }
    double dien_tich()          { return 3.14*r*r; }
    double chu_vi()             { return 3.14*2.*r; }
};

void main() {
    Circle c;
    c.setR(1.23);
    std::cout << "Dien tich: " << c.dien_tich() << std::endl
              << "Chu vi: " << c.chu_vi() << std::endl;
}
```

Constructor

- Constructor: phương thức khởi tạo cho đối tượng, không trả về giá trị, được tự động gọi khi đối tượng được tạo ra
- Có thể khai báo nhiều constructor với tham số khác nhau, trình dịch sẽ tự động xác định gọi constructor tuỳ trường hợp dựa vào tham số khởi tạo

```
class Circle {
public:
    Circle()  { r = 0.; }  // constructor mặc định (không tham số)
    Circle(double rr)  { r = rr; }
    ...
};

Circle c1, c2(), c3[3];     // constructor 1
Circle c4(1.23);            // constructor 2
c1 = Circle;                // lỗi
c1 = Circle();              // constructor 1
c1 = Circle(2.33);          // constructor 2

Circle* c5 = new Circle;           // constructor 1
Circle* c6 = new Circle();         // constructor 1
Circle* c7 = new Circle(3.22);     // constructor 2
Circle* c8 = new Circle[3];        // constructor 1
```

Constructor sao chép (copy constructor)

- Dùng cho việc khởi tạo một đối tượng mới từ một đối tượng đã có thuộc lớp đó. Có hai cú pháp khởi tạo sử dụng CSC: cú pháp chuẩn và cú pháp gán:

```
class Circle {
    ...
    Circle(const Circle& c) { r = c.r; }
};

Circle c1(2.5);              // không dùng CSC
Circle c2 = c1, c3(c1);      // có dùng CSC
c2 = c1;                     // phép gán, không dùng CSC
```

- Còn được dùng khi tạo các đối tượng tạm (vd: tham số khi gọi hàm)

```
void func1(Circle c) { ... }
void func2(const Circle& c) { ... }
func1(c1); //tạo ra một đối tượng c tạm, copy từ c1 bằng CSC
func2(c1); //không tạo đối tượng mới, không gọi CSC
```

→ nên dùng "`const Circle& c`" ở tham số hàm thay "`Circle c`" để tăng hiệu quả

- Nếu không khai báo CSC, trình biên dịch sẽ tự động tạo một CSC mặc định. CSC mặc định copy toàn bộ biến thành phần tử đối tượng cũ sang đối tượng mới

Constructor chuyển kiểu (cast constructor)

- Dùng cho việc khởi tạo một đối tượng từ một biến/đối tượng đã có nhưng khác kiểu. Cũng có hai cú pháp khởi tạo sử dụng CCK:

```
class Ellipse {
    ...
    Ellipse(const Circle& c) { rx = ry = c.r; }
};

Ellipse e1(c1), e2 = c2; // có dùng CCK Circle -> Ellipse
e1 = c1;  // chuyển kiểu ngầm định tạo một đối tượng Circle -> Ellipse
          // bằng CCK rồi thực hiện tiếp phép gán Ellipse -> Ellipse
```

- Còn được dùng trong các trường hợp chuyển kiểu:

```
void func3(const Ellipse& e) { ... }
func3(c1);  // có dùng CCK (ngầm định)
cout  << ((Ellipse)c1).area();  // có dùng CCK (tường minh)
```

  - Có thể thêm từ khoá "explicit" trước khai báo CCK nếu hạn chế không cho dùng nó trong các phép chuyển kiểu ngầm định
  - Sẽ trở lại với phép gán và chuyển kiểu trong bài học về định nghĩa chồng toán tử

- Trở lại lớp Circle:

```
class Circle {
    ...
    Circle(double rr) {r = rr; }  // chính là CCK int -> Circle
};
```

Danh sách khởi tạo

- Sau phần tên constructor có thể có danh sách khởi tạo các biến thành phần của lớp cần được khởi tạo

```
class Person {
private:
    string name;
    int age;
public:
    Person(const char* n): name(n) { ... }
    Person(const char* n, int a): name(n), age(a) { ...}
    ...
};
```

- Các constructor trong ví dụ trên có thể được hiểu là:

```
Person(const char* n) {
    name = n;
    ...
}
Person(const char* n, int a) {
    name = n;
    age = a;
    ...
}
```

- Bắt buộc phải dùng với các biến thành phần const hoặc tham chiếu

Destructor

- Destructor: phương thức huỷ đối tượng, không trả về giá trị, được gọi tự động khi đối tượng bị huỷ
- Không dùng malloc()/free() để làm việc với class
- Chỉ có thể khai báo tối đa một destructor cho mỗi class

```
class String {
private:
   char* str;
public:
   String()          { str = NULL; }
   String(const char* s)
            { str = new char[strlen(s)+1];  strcpy(str, s); }
   ~String()         { if (str) delete str; }
   ...
};
String* abc() {
   String s1, s2("Hello!"), s3[3];
   String *s4 = new String("xyz"), *s5 = new String[3];
   String *s6 = new String("abc");
   delete s4;        // huỷ 1 đối tượng
   delete[] s5;      // huỷ 3 đối tượng, không được dùng: delete s5
   return s6;        // huỷ s1, s2, s3[3] nhưng s6 vẫn còn
}
```

Tách phần khai báo và nội dung các phương thức

string.h

```
class String {
private:
  char* str;
public:
  String();
  String(const char* s);
  ~String();
  void set(const char* s);
  const char* get();
};
```

string.cpp

```
String::String()
  { str = NULL; }
String::String(const char* s)
  { str = NULL; set(s); }
String::~String()
  { if (str) delete str; }
void String::set(const char* s) {
  if (str) delete str;
  if (!s) { str = NULL; return; }
  str = new char[strlen(s)+1];
  strcpy(str, s);
}
const char* String::get()
  { return str; }
```

Hàm và lớp bạn (friend function/class)

- Hàm và lớp được khai báo là bạn của lớp nào thì có thể truy xuất tới các biến và phương thức private của một lớp đó
  - ```
    class Circle {
    private:
        double r;
    public:
        ...
        friend void printCircle(Circle c);
        friend class Ellipse;
    };

    void printCircle(Circle c)
        { cout << "Ban kinh: " << c.r; }

    class Ellipse {
    private:
        double rx, ry;
    public:
        void convert(Circle c) {
            rx = ry = c.r;
        }
    };
    ```
    ```

Con trỏ "this"

- Là con trỏ chỉ có phạm vi trong các phương thức của một lớp, trỏ tới chính đối tượng đang được gọi

```
class Buffer;
void do_smth(Buffer* buf);

class Buffer {
private:
   char* b;    int n;
public:
   Buffer(int n)
     { this->n = n; this->b = new char[n]; }
   ~Buffer()
     { delete this->b; }
   void some_method()
     { do_smth(this); }
};

Buffer buf(4096);
buf.some_method();
```

- Chữ **màu đỏ** có thể lược bớt (được ngầm hiểu), **màu xanh** phải giữ nguyên

Biến và phương thức static của lớp

- Biến static là biến chỉ tồn tại duy nhất đối với tất cả các đối tượng của một lớp (ngay cả khi chưa có đối tượng nào được tạo)
- Phương thức static là phương thức chỉ dùng được các biến static của lớp

```
class C {
public:
  static int count;
  C()    { count++; }
  ~C()   { count--; }
  static int getCount()
    { return count;   }
  ...
};
int C::count = 0;
```

```
C c1, c2, c3[10],
  *c4 = new C(),
  *c5 = new C[20];
delete c4;

cout << "So doi tuong kieu C hien co: "
     << C::getCount() << endl;
// cout << C::count << endl;
// cout << c1.count << endl;
// cout << c2.getCount() << endl;
```

- Có thể truy xuất đến biến/phương thức static thông qua tên lớp và toán tử "::" mà không cần đối tượng gọi, hoặc coi như thành phần của các đối tượng như bình thường
- Không tồn tại con trỏ "this" trong các phương thức static

Phương thức hằng

- Một phương thức được khai báo là hằng thì trong phương thức đó, các biến của lớp sẽ là hằng. Hệ quả:
  - Không thể gán hay thay đổi giá trị các biến thành phần trong một phương thức hằng
  - Không thể gọi được các phương thức không hằng ở trong một phương thức hằng
  - Nếu một đối tượng được khai báo là hằng, thì chỉ dùng được các phương thức hằng của nó

→ khai báo toàn bộ các phương thức không thay đổi các biến thành phần là hằng

```
class Circle {
  ...
  void setR(double r) { this->r = r; }
  double getR() const { return r; }
  double area() const { return PI*r*r; }
  void cf(double r) const {
    area();       // OK
    setR(r);      // lỗi
    this->r = r; // lỗi
  }
};
```

```
Circle c1;
const Circle c2(2.333);

c1.setR(1.22); // OK
c2.setR(1.22); // lỗi

c1.area();     // OK
c2.area();     // OK

cout << c1.getR()   // OK
     << c2.getR(); // OK
```

Bài tập

1. Viết một lớp String để đóng gói kiểu chuỗi char* và các phương thức cần thiết
2. Viết một lớp File để đóng gói kiểu FILE* và các phương thức cần thiết
3. Viết một lớp LList để thao tác với DSLK, sử dụng lại thư viện đã viết bằng C
4. Viết lớp Fraction (phân số) với những phương thức cần thiết
5. Viết hai lớp Circle và Ellipse hoàn chỉnh, có sử dụng với các khái niệm đã học trong bài: biến private, constructor thường, sao chép, chuyển kiểu, destructor, biến static để đếm số đối tượng, lớp bạn, phương thức hằng,…
6. Viết hai lớp Complex (số phức) và Vector với những phương thức cần thiết