به نام خدا

برنامه‌سازی پیشرفته

آرش شفیعی

برنامه‌سازی شیءگرا

مقدمه

- یک کلاس یک نوع دادهٔ تعریف‌شده توسط کاربر [1] است که یک مفهوم را پیاده‌سازی می‌کند. معمولا در پیاده‌سازی یک سیستم، نیاز به تعدادی مفهوم انتزاعی داریم که هر کدام از این مفاهیم ویژگی‌هایی دارند. این مفاهیم را توسط یک کلاس در زبان سی++ نشان می‌دهیم و پیاده‌سازی می‌کنیم.
- برنامه‌ای که از تعدادی مفهوم تشکیل شده باشد که با هم در ارتباط هستند، قابل فهم‌تر است و راحت‌تر می‌توان آن را عیب‌یابی کرد. همچنین در چنین برنامه‌ای منطق برنامه را می‌توان بهتر دنبال کرد و بهره‌وری بهتری خواهد داشت.

[1] user-defined type

کلاس‌ها

- یک نمونه از یک کلاس را شیء می‌نامیم. پس برای استفاده از یک کلاس باید یک شیء از آن بسازیم و عملیات تعریف شده برای کلاس را بر روی شیء مورد نظر انجام دهیم.
- یک کلاس تعدادی عضو داده [1] و تعدادی تابع عضو [2] دارد. اعضای داده‌ای می‌توانند از انواع اصلی، تعریف‌شده توسط کاربر (شامل کلاس‌ها)، و انواع مشتق شده باشند.
- به داده‌های یک کلاس ویژگی‌ها [3] و به توابع کلاس رفتارها [4] نیز می‌گوییم.
- هر عضو کلاس یک سطح دسترسی دارد. این سطح دسترسی می‌تواند عمومی (`public`)، خصوصی (`private`)، یا محافظت‌شده (`protected`) باشد.
- سطوح دسترسی عمومی و خصوصی را توضیح می‌دهیم و سطح دسترسی محافظت‌شده را در آینده بررسی خواهیم کرد.

[1] data member
[2] member function
[3] attributes
[4] behaviours

کلاس‌ها

- سطح دسترسی عمومی بدین معناست که دادهٔ تعریف شده توسط این سطح دسترسی برای همه قابل استفاده است. پس اگر یک شیء از کلاسی تعریف شود، می‌توان به اعضای عمومی آن دسترسی پیدا کرد.

```
۱ class student {
۲ public:
۳     string name;
۴ };
۵ student st; //st is an object of student class
۶ st.name = "ali";
```

کلاس‌ها

- سطح دسترسی خصوصی بدین معناست که دادهٔ تعریف شده توسط این سطح دسترسی فقط برای توابع عضو قابل استفاده است. پس اگر یک شیء از کلاسی تعریف شود، نمی‌توان به اعضای خصوصی آن دسترسی پیدا کرد. اگر سطح دسترسی برای یک عضو تعریف نشود، سطح دسترسی پیش‌فرض خصوصی است.

```
۱  class student {
۲  private:
۳      int average;
۴  public:
۵      string name;
۶      int getAverage() { return average ; }
۷  };
۸  student st; //st is an object of student class
۹  st.name = "ali";
۱۰ st.average = 17; // error : average is a private member
```

کلاس‌ها

- معمولا بهتر است همهٔ داده‌ها با سطح دسترسی خصوصی تعریف شوند. سپس توابعی با سطح دسترسی عمومی برای تغییر داده‌های عضو تعریف کنیم. بدین ترتیب از دستکاری شدن و تغییرات غیرمنتظره داده‌های عضو جلوگیری می‌کنیم.

- تنها توابعی را عمومی تعریف می‌کنیم که شیء نیاز به دسترسی به آنها را دارد. باقی توابع نیز با سطح دسترسی خصوصی تعریف می‌شوند.

```
۱  class student {
۲  private:
۳      string name;
۴      int average;
۵  public:
۶      void setName(string n) { name = n; }
۷      int getAverage() { return average ; }
۸  };
۹  student st;
۱۰ st.setName("ali");
```

کلاس‌ها

- همانطور که برای انواع دادهٔ اصلی می‌توانستیم اشاره‌گر و آرایه تعریف کنیم، برای کلاس‌ها نیز می‌توانیم اشاره‌گرهایی به اشیاء و یا آرایه‌هایی از اشیا بسازیم.

```
۱ student st;
۲ student * stptr;
۳ stptr = &st;
۴ int s = stptr->getAverage();
۵ student stgroup[40];
۶ stgroup[0].setName("ali");
```

کلاس‌ها

- هر کلاس می‌تواند یک سازنده [1] داشته باشد. وقتی شیئی از یک کلاس ساخته می‌شود، به طور خودکار سازنده فراخوانی می‌شود. نام سازنده با نام کلاس یکسان است و سازنده هیچ مقداری باز نمی‌گرداند. سازنده‌ای که مقدار ورودی ندارد، سازندهٔ پیش‌فرض [2] نامیده می‌شود.

```
۱  class student {
۲  private:
۳      string name;
۴      int age;
۵      int average;
۶  public:
۷      // default constructor
۸      student() { name = " "; age = 0; }
۹      // constructor with name and age as arguments
۱۰     student(string n, int a) : name(n), age(a) {}
۱۱ };
```

[1] constructor
[2] default constructor

کلاس‌ها

- بنابراین در مثال قبل تابع سازنده در کلاس `student` سربارگذاری [1] شده است.
- با استفاده از سربارگذاری تابع [2]، توابعی که از نظر منطقی کار یکسانی را انجام می‌دهند و تنها تفاوت آنها در نوع ورودی‌های آنهاست را توسط یک نام واحد نامگذاری می‌کنیم.
- دو تابع که ورودی‌های آنها یکسان است و نوع خروجی آنها متفاوت است نمی‌توانند سربارگذاری شوند.

```
۱ bool print(int i);
۲ bool print(float f); // print is overloaded
۳ int print(int i); //error : functions that differ
۴       // only in their return type cannot be overloaded.
```

[1] overloaded
[2] function overloading

کلاس‌ها

- سازندهٔ پیش‌فرض را می‌توانیم همچنین با مقداردهی ورودی‌های یک سازنده غیرپیش‌فرض بسازیم.

```
۱  class student {
۲  private:
۳      string name;
۴      int age;
۵      int average;
۶  public:
۷      // this is both a default constructor and
۸      // a constructor with name and age as arguments
۹      student(string n=" ", int a=0) : name(n), age(a) {}
۱۰ };
```

کلاس‌ها

- برای تخصیص حافظه به یک اشاره‌گر می‌توانیم از عملگر new استفاده کنیم. وقتی از عملگر new استفاده می‌کنیم، سازنده نیز فراخوانی می‌شود.
- همچنین از عملگر delete برای آزادسازی فضای حافظه استفاده می‌کنیم.
- در صورتی که یک اشاره‌گر به یک آرایه اشاره کند از []delete برای آزادسازی حافظه استفاده می‌کنیم.

```
۱  // constructor is not called when malloc is used
۲  student *st1 = (student*)malloc(sizeof(student));
۳  // constructor is called when new is used
۴  student *st2 = new student;
۵  student *st3 = new student("ali", 21);
۶  student *st4 = new student[40];
۷  free(st1);
۸  delete st2;
۹  delete st3;
۱۰ delete[] st4;
```

کلاس‌ها

- هر کلاس همچنین یک مخرب [1] دارد. وقتی شیئی از کلاس تخریب می‌شود، تابع مخرب فراخوانی می‌شود.
- ممکن است در سازندهٔ یک کلاس حافظه‌ای را به طور پویا تخصیص دهیم. وقتی که شیئی از کلاس ساخته شود حافظه تخصیص داده می‌شود، ولی وقتی شیء از بین می‌رود، حافظه آزاد نمی‌شود. همچنین ممکن است در یکی از توابع کلاس حافظه‌ای به طور پویا تخصیص داده شود، که در آن صورت نیز، در صورتی که آن تابع فراخوانی شده باشد، باید حافظه را در مخرب آزاد کرد.
- بنابراین باید در مخرب یک کلاس همهٔ حافظه‌هایی که تخصیص داده‌ایم را آزاد کنیم.

```
۱ class student {
۲ private:
۳     int * courses;
۴ public:
۵     student() { courses = new int[100]; }
۶     ~student() { delete[] courses; }
۷ };
```

[1] destructor

کلاس‌ها

- در ابتدای برنامه همهٔ متغیرهای عمومی در حافظه ساخته می‌شوند، بنابراین اگر شیئی از یک کلاس به طور عمومی تعریف شده باشد، در ابتدای برنامه سازندهٔ آن فراخوانی می‌شود و در پایان برنامه مخرب آن فراخوانی می‌شود.
- همچنین هر گاه وارد تابعی می‌شویم، سازندهٔ اشیای درون تابع فراخوانی می‌شوند و در پایان اجرای تابع مخرب آن اشیا (از آخر به اول) فراخوانی می‌شوند.
- برای اشیایی که با کلیدواژهٔ `static` در یک تابع تعریف شده‌اند، در اولین فراخوانی تابع سازندهٔ آنها فراخوانی می‌شود و مخرب آنها در پایان برنامه فراخوانی می‌شود.

کلاس‌ها

- می‌توانیم یک شیء را با استفاده از کلیدواژهٔ const به صورت ثابت تعریف کنیم.
- از آنجایی که یک شیء ثابت نمی‌تواند هیچ‌یک از ویژگی‌هایش را تغییر دهد، لذا هیچ تابعی هم بر روی آن قابل فراخوانی نیست، چرا که یک تابع ممکن است یکی از ویژگی‌های آن را تغییر دهد.
- برای اینکه یک شیء ثابت بتواند یک تابع را فراخوانی کند، در تعریف آن تابع باید از کلیدواژهٔ const استفاده کرد.
- یک تابع ثابت نمی‌تواند ویژگی‌های کلاس را تغییر دهد.

کلاس‌ها

```
۱  class student {
۲  private:
۳      string name;
۴      int birth_year;
۵  public:
۶      student(string n, int b) { name = n; bith_year = b; }
۷      string getName() const { return name; }
۸      int getBirthYear() { return birth_year; }
۹  };
۱۰ const student st("ali", 2000);
۱۱ cout << st.getName();
۱۲ cout << st.birthYear(); // error: constant object
۱۳                         // cannot call non-constant function
```

کلاس‌ها

- داده‌های عضو یک کلاس نیز می‌توانند ثابت باشند. در اینصورت امکان تغییر آنها بعد از ساخته شدن شیء وجود نخواهد داشت.
- داده‌های ثابت یک کلاس را باید قبل از ورود به بدنهٔ سازنده در لیست مقدارهی به صورت `:member1(var1), member2(var2), ...` مقداردهی اولیه کرد.

```
۱ class student {
۲ private:
۳     const int national_id;
۴ public:
۵     student(int id) : national_id(id) { }
۶ };
```

کلاس‌ها

- لیست مقداردهی در موارد دیگری نیز استفاده می‌شود. برای مثال وقتی یک دادهٔ عضو یک متغیر مرجع باشد که مقداردهی اولیه نیاز دارد و یا وقتی یکی از اعضای کلاس شیئی از کلاسی باشد که آن کلاس سازندهٔ پیش‌فرض ندارد، در چنین مواردی نیز از لیست مقداردهی استفاده می‌کنیم.
- به طور کلی هر متغیر یا شیئی که مقدار دهی اولیه نیاز داشته باشد، باید در لیست مقداردهی قبل از ورود به بدنهٔ سازنده مقدار اولیه آن تعیین شود.

کلاس‌ها

- اعضای یک کلاس می‌توانند همچنین ایستا (`static`) باشند.
- ویژگی یک متغیر ایستا این است که یک بار مقداردهی اولیه می‌شود و حتی اگر از حوزهٔ تعریف آن متغیر خارج شویم، متغیر مقدار خود را نگه می‌دارد.
- هر متغیر عضو یک کلاس برای یک شیء معین از آن کلاس در حافظه در فضای پشته [1] ساخته می‌شود. اما اگر متغیری ایستا باشد، آن متغیر در فضای داده‌ای [1] ساخته می‌شود و بنابراین مقدار آن برای همهٔ اشیای ساخته شده از کلاس یکسان است.

[1] stack
[1] data segment

کلاس‌ها

- فرض کنید می‌خواهیم متغیری در یک کلاس تعریف کنیم که تعداد اشیای ساخته شده از آن کلاس را بشمارد.
- اگر این متغیر به طور عادی یک عضو داده‌ای باشد، به ازای ساخته شدن هر شیء، متغیر برای آن شیء ساخته شده و مقداردهی می‌شود. برای حل این مشکل از متغیر ایستا استفاده می‌کنیم.

```
// student.h
class student {
private:
    static int counter;
public:
    student() { counter++; }
    static int getCounter() { return counter; }
    ~student() { counter--; }
};
// student.cpp
int student::counter = 0;
```

کلاس‌ها

- در اینجا متغیر ایستای counter تنها یک بار در یک فضای حافظه برای کلاس student ساخته می‌شود و حتی اگر هیچ شیئی از این کلاس وجود نداشته باشد، این متغیر ساخته شده است.
- حال می‌توانیم از این متغیر ایستا به صورت زیر استفاده کنیم.

```
۱ student::getCounter(); // counter = 0
۲ student st1;
۳ st1.getCounter(); // counter = 1
۴ student st2;
۵ st2.getCounter(); // counter = 2
۶ student st3;
۷ student::getCounter(); // counter = 3
```

کلاس‌ها

- به هر یک از توابع کلاس می‌توانیم شیئی از همان کلاس به عنوان ورودی بدهیم و یا از یک تابع یک کلاس شیئی از همان کلاس را بازگردانیم.

```
۱  class student {
۲  private:
۳      string name;
۴      int age;
۵      int average;
۶  public:
۷      student(string n, int a) : name(n), age(a) {}
۸      bool compareAge(student s) { return (age >= s.age); }
۹      student * copy() { return new student(name, age); }
۱۰ };
```

کلاس‌ها

- به هر یک از توابع کلاس می‌توانیم شیئی از همان کلاس به عنوان ورودی بدهیم و یا از یک تابع یک کلاس شیئی از همان کلاس را بازگردانیم.

```
۱ student st1, st2, *s3;
۲ if (st1.compareAge(st2)) {
۳     cout << "st1 is older than st2\n";
۴ }
۵ s3 = s1.copy();
```

کلاس‌ها

- وقتی یک تابع از یک کلاس فراخوانی می‌شود، در واقع شیئی از آن کلاس ساخته شده، و شیء مورد نظر تابع کلاس را فراخوانی کرده است. در درون تعریف تابع نمی‌دانیم چه شیئی تابع را فراخوانی کرده است، اما زبان سی++ اشاره‌گری تعریف کرده و در اختیار برنامه‌نویس قرار داده است که با استفاده از آن اشاره‌گر به شیئی که تابع برای آن فراخوانی شده، دسترسی پیدا می‌کنیم. این اشاره‌گر this نام دارد.

```
۱ class student {
۲ public:
۳     student * compareAge(const student * st) {
۴         if (this->age >= st->age)
۵             return this;
۶         else
۷             return st;
۸     }
۹ };
```

- استفاده از this->age به جای age در اینجا به جهت خوانایی بهتر برنامه است.

کلاس‌ها

- کلاس وکتور را در نظر بگیرید.

```
۱  class Vector {
۲  public:
۳      Vector(int s) :elem{new double[s]}, sz{s} {
۴          for (int i=0; i!=s; ++i)
۵              elem[i]=0;
۶      }
۷      void setElement(int i, double d) { elem[i] = d; }
۸      ~Vector() { delete[] elem; }
۹  private:
۱۰     double* elem;
۱۱     int sz;
۱۲ };
```

کلاس‌ها

- حال فرض کنید می‌خواهیم یک شیء از این کلاس بسازیم. پس از مقداردهی تعدادی از عناصر این وکتور می‌خواهیم وکتور دیگری بسازیم که کپی وکتور اولیه است.

```
۱
۲ Vector v1(10);
۳ v1.setElement(0,1.6);
۴ v1.setElement(1,3.14);
۵ Vector v2 = v1;
```

- در اینجا چه اتفاقی می‌افتد؟ آیا کامپایلر به طور خودکار اعضای v1 را در v2 کپی می‌کند؟ در اینصورت کامپایلر اشاره‌گر elem را چگونه کپی می‌کند؟

کلاس‌ها

- وقتی از عملگر تساوی برای کپی کردن یک شیء در شیء دیگر استفاده می‌کنیم، کامپایلر تابعی به نام سازندهٔ کپی [1] را فراخوانی می‌کند.
- در صورتی که تابع سازندهٔ کپی توسط کاربر تعریف نشده باشد، مانند توابع سازنده و مخرب، یک سازندهٔ کپی پیش‌فرض [2] توسط کامپایلر تعریف می‌شود.
- در تابع سازندهٔ کپی پیش‌فرض، کامپایلر اعضای کلاس را یک به یک کپی می‌کند. پس سازندهٔ کپی پیش‌فرض برای کلاس وکتور به صورت زیر خواهد بود.

```
۱ class Vector {
۲ public:
۳     Vector(Vector & v) { elem = v.elem; sz = v.sz; }
۴ private:
۵     double* elem;  int sz;
۶ };
```

[1] copy constructor
[2] default copy constructor

- اما منظور استفاده‌کننده کلاس وکتور کپی کردن تمام عناصر وکتور است و نه کپی کردن مقدار اشاره‌گر.

```
۱ Vector v2 = v1;
۲ // v1 and v2 both use the same pointer elem
۳ // if v2 changes its elements, the elements of v1 also changes.
```

- در کد بالا با استفاده از سازندهٔ کپی پیش‌فرض v1 و v2 هر دو اشاره‌گری به نام elem دارند که به یک مکان واحد در حافظه اشاره می‌کند. همچنین با تخریب v1 حافظهٔ تخصیص داده شده برای elem آزاد می‌شود و در هنگام تخریب v2 مکانی در حافظه که قبلا آزاد شده باید دوباره آزاد شود که به یک خطای حین اجرا [1] برمی‌خوریم.

[1] run-time fault

کلاس‌ها

- بنابراین برای استفاده از عملگر تساوی و کپی اشیا، سازندهٔ کپی باید توسط برنامه‌نویس پیاده‌سازی شود.

```
۱  class Vector {
۲  public:
۳      Vector(Vector & v) {
۴          sz = v.sz;
۵          elem = new double[sz];
۶          for (int i=0; i<v.sz; i++)
۷              elem[i] = v.elem[i];
۸      }
۹  private:
۱۰     double* elem;  int sz;
۱۱ };
۱۲ Vector v2 = v1;
۱۳ // v1 and v2 have two different locations on memory
۱۴ // allocated for their elements
```

کلاس‌ها

- به طور کلی سازندهٔ کپی در سه موقعیت فراخوانی می‌شود.

۱. وقتی از عملگر تساوی برای کپی یک شیء در یک شیء دیگر استفاده می‌کنیم.

```
۱ Vector v1;
۲ Vector v2 = v1;
```

۲. وقتی یک تابع فراخوانی با مقدار می‌شود و مقادیر ورودی تابع اشیایی از یک کلاس هستند.

```
۱ void print(Vector v) {
۲    for (int i=0; i<v.size(); i++)
۳        cout <<v.getElement(i);
۴ }
۵ Vector v1;
۶ print(v1); // v = v1
```

۳. وقتی یک تابع شیئی از یک کلاس را بازمی‌گرداند.

```
۱ Vector larger(Vector &v1, Vector &v2) {
۲     if (v1.size() > v2.size()) return v1;
۳     else return v2;
۴ }
۵ Vector v1 {1,2}, v2 {3};
۶ Vector v3 = larger(v1,v2); // tmp = v1; v3 = tmp;
```

کلاس‌ها

- فرض کنید می‌خواهیم یک نوع دادهٔ جدید برای ذخیره و محاسبات بر روی اعداد مختلط تعریف کنیم.

```
class complex {
    double re, im; // representation: two doubles
public:
    // construct complex from two scalars
    complex(double r, double i) :re{r}, im{i} {}
    // construct complex from another complex
    complex(complex& c) :re{c.real()}, im{c.imag()} {}
    // construct complex from one scalar
    complex(double r) :re{r}, im{0} {}
    // default complex: {0,0}
    complex() :re{0}, im{0} {}

    ...
```

```
۱      ...
۲      double real() const { return re; }
۳      void real(double d) { re=d; }
۴      double imag() const { return im; }
۵      void imag(double d) { im=d; }
۶      ...
```

کلاس‌ها

- در اینجا سه سازنده [1] برای کلاس `complex` تعریف کردیم.
- وقتی یک شیء از یک کلاس ساخته می‌شود، سازندهٔ آن فراخوانی می‌شود.
- اگر در هنگام ساختن شیء سازندهٔ آن مشخص نباشد، سازندهٔ پیش‌فرض فراخوانی می‌شود. سازندهٔ پیش‌فرض، سازنده‌ای است که هیچ مقدار ورودی ندارد.

```
۱ complex c1;
۲ complex c2(3);
۳ complex c3(2,4);
۴ complex c4 {3};
۵ complex c5 {2,4};
۶ complex c6(c5);
```

[1] constructor

سربارگذاری عملگرها

- حال می‌خواهیم با استفاده از عملگرهای رایج در زبان سی++ دو عدد مختلط را با هم جمع یا از هم تفریق کنیم.
- برای این کار نیاز داریم عملگرها را برای اعمال بر روی اشیای کلاس تعریف کنیم.
- به تعریف یک عملگر برای یک کلاس سربارگذاری عملگر می‌گوییم. در سربارگذاری یک عملگر (تعریف مجدد یک عملگر برای یک کلاس) در واقع تابعی برای کلاس تعریف می‌کنیم که نام آن یک عملگر است.
- عملگرها می‌توانند یگانی [1] یا دوتایی [2] باشند.
- عملگرهای یگانی مانند `!`, `--`, `++` و عملگرهای دوتایی مانند `&`, `||`, `&&`, `*`, `/`, `-`, `+`, `!=`, `==`, `|` هستند.

[1] unary operator
[2] binary operator

سربارگذاری عملگرها

- می‌توانیم عملگر + را نیز به صورت زیر تعریف کنیم.

```
۱       ...
۲       complex operator+(const complex& b) {
۳           return complex(re+b.re, im+b.im);
۴       }
۵   };
```

- سپس می‌توانیم از این عملگر به صورت زیر استفاده کنیم.

```
۱   complex c1 {1,2};
۲   complex c2(3);
۳   complex c3 = c1 + c2;
۴   complex c4 {c2+complex{1,2.3}};
```

سربارگذاری عملگرها

- در تعریف عملگر جمع به صورت زیر، اگر عملگر بر روی یک متغیر ثابت فراخوانی شود، کامپایلر خطای کامپایل صادر خواهد کرد، زیرا یک تابع غیر ثابت بر روی یک شیء ثابت فراخوانی شده است.

```
۱       ...
۲       complex operator+(const complex& b) {
۳           return complex(re+b.re, im+b.im);
۴       }
۵   };
۶   const complex c1; complex c2;
۷   complex c3 = c1 + c2; // error : non-constant function is called
۸       // on constant object.
```

- در صورتی که می‌دانیم تابع operator+ مقدار اعضای کلاس را تغییر نمی‌دهد. پس می‌توانیم آن را به صورت ثابت تعریف کنیم.

```
۱   complex operator+(const complex& b) const;
```

سربارگذاری عملگرها

- وقتی یک عملگر بر روی یک شیء اعمال می‌شود، در واقع تابع تعریف شده برای آن عملگر فراخوانی می‌شود.
- پس کامپایلر به طور خودکار در هنگام کامپایل صورت نحوی یک عبارت حاوی عملگر را تشخیص داده، آن را ترجمه می‌کند.
- در مثال فوق `c1+c2;` را کامپایلر به صورت `c1.operator+=(c2);` ترجمه می‌کند.

```
۱ c3 = c1+c2; // is equivalent to:
۲ c3= c1.operator+=(c2);
```

سربارگذاری عملگرها

- عملگرهای == و != را نیز برای این کلاس تعریف می‌کنیم.

```
۱       ...
۲       bool operator==(const complex& z) {
۳           return (re == z.re && im == z.im);
۴       }
۵       bool operator!=(const complex& z) {
۶           return (re != z.re || im != z.im);
۷       }
۸   };
```

سربارگذاری عملگرها

- می‌توانیم عملگر + و == را به گونه‌ای تعریف کنیم که یک عدد مختلط را با یک عدد صحیح جمع کند یا با یک عدد صحیح مقایسه کند.

```
complex operator+(const int& i) {
    return complex(re+i, im);
}
bool operator==(const int& i) {
    return (re == i && im == 0);
}
```

- سپس می‌توانیم از این عملگر به صورت زیر استفاده کنیم.

```
complex c2 = c1 + 5;
if (c3==4) { ... }
```

سربارگذاری عملگرها

- همچنین می‌توانیم یک عملگر را خارج از کلاس تعریف کنیم.

```
۱ complex operator-(complex a, complex b) { return a-=b; }
۲ bool operator==(complex a, complex b) { // equal
۳     return a.real()==b.real() && a.imag()==b.imag();
۴ }
۵ bool operator!=(complex a, complex b) {
۶     return !(a==b);
۷ }
```

سربارگذاری عملگرها

- برای جمع یک عدد صحیح و یک عدد مختلط وقتی که عدد صحیح اولین عملوند و عدد مختلط دومین عملوند باشد، راهی جز تعریف تابع خارج از کلاس نداریم.

```
۱  complex operator+(const int& a, const complex& b) {
۲      return complex(a+b.real(), b.image());
۳  }
۴
۵  c3 = 1 + c2 // c3 = operator+(1, c2);
```

سربارگذاری عملگرها

- برای اینکه بتوانیم در تابعی که خارج از کلاس تعریف می‌شود، به اعضای خصوصی کلاس دسترسی پیدا کنیم، آن تابع را باید به صورت `friend` تعریف کنیم.

```
۱  class complex {
۲      ...
۳  public:
۴      friend complex operator+(const int& a, const complex& b);
۵  };
۶
۷  complex operator+(const int& a, const complex& b) {
۸      // we can access b.re and b.im, because
۹      // operator+(int, complex) is a friend function.
۱۰     return complex(a+b.re, b.im);
۱۱ }
۱۲
۱۳ c3 = 1 + c2 // c3 = operator+(1, c2);
```

سربارگذاری عملگرها

- توصیه می‌شود، توابع حتی الامکان به صورت `friend` تعریف نشوند و تنها در مواقع ضروری مانند مثال قبل توابع را به صورت `friend` تعریف کنیم.
- همچنین می‌توان یک کلاس را دوست یک کلاس تعریف کرد. بدین ترتیب وقتی کلاس F دوست کلاس A باشد همهٔ توابع کلاس F به اعضای خصوصی کلاس A دسترسی خواهند داشت.

```
۱ class A {
۲     ...
۳ public:
۴     friend class F;
۵ };
۶ // now all functions of F can access
۷ // private members of A.
```

- توصیه می‌شود کلاس‌ها نیز دوست تعریف نشوند، مگر اینکه واقعا نیازی به تعریف آن باشد و دلیلی برای آن وجود داشته باشد.

سربارگذاری عملگرها

- می‌توانیم عملگرهای دیگری مانند += و -= را نیز برای این کلاس تعریف می‌کنیم.

```
۱       ...
۲       complex& operator+=(complex z) {
۳           re+=z.re; im+=z.im;
۴           return *this; // and return the result
۵       }
۶       complex& operator-=(complex z) {
۷           re-=z.re; im-=z.im;
۸           return *this;
۹       }
۱۰  };
```

سربارگذاری عملگرها

- دلیل استفاده از `complex &` در مقدار خروجی تابع این است که اگر بخواهیم مقدار `(c1 += c2) += c3` را محاسبه کنیم، باید پس از محاسبهٔ مقدار `c1 += c2` متغیر `c1` را بازگردانیم تا با مقدار `c3` جمع شود.
- اگر نوع خروجی تابع به جای `complex &` نوع `complex` باشد، آنگاه پس از محاسبهٔ مقدار `c1 += c2` یک متغیر موقت بازگردانده می‌شود و مقدار آن متغیر موقت با `c3` جمع می‌شود.

```
complex operator+=(complex z) {
    re+=z.re; im+=z.im;
    return *this;
}
(c1 += c2) += c3 // this is equal to:
// (tmp=c1.operator+=(c2)).operator+=(c3), so tmp is updated.
```

سربارگذاری عملگرها

- اما اگر نوع خروجی تابع `&complex` باشد، آنگاه پس از محاسبهٔ مقدار `c1 += c2` متغیر `c1` بازگردانده می‌شود و مقدار آن با `c3` جمع می‌شود.

```
complex& operator+=(complex z) {
    re+=z.re; im+=z.im;
    return *this;
}
(c1 += c2) += c3 // this is equal to:
// (c1=c1.operator+=(c2)).operator+=(c3), so c1 is updated.

c1 += c2 += c3 // this is equal to:
// (c1.operator+=(c2.operator+=(c3))
// because += associativity is rigt to left
```

سربارگذاری عملگرها

- در سربارگذاری عملگرها، تنها می‌توان عملگرهای موجود در زبان سی++ را سربارگذاری کرد، و نمی‌توان اولویت این عملگرها را تغییر داد.
- همچنین عملگرهای زیر را نمی‌توان سربارگذاری کرد:
- عملگر . که برای دسترسی به اعضای کلاس به کار می‌رود.
- عملگر : ? که برای انشعاب شرطی به کار می‌رود.
- عملگر : : که برای تفکیک حوزهٔ اشیا و کلاس‌ها به کار می‌رود.
- و عملگر *. که برای دسترسی به اعضای کلاس به کار می‌رود.

```
۱ class Vector { public : int size; ... };
۲ int Vector::* s = &Vector::size;
۳ Vector v;
۴ cout << (v.*s);
```

سربارگذاری عملگرها

- عملگر یگانی ! را برای کلاس complex اینگونه تعریف می‌کنیم.

```
۱ class complex {
۲ public :
۳     complex operator!() {
۴         return complex(-re, -im);
۵     }
۶ };
```

سربارگذاری عملگرها

- عملگرهای یگانی ++ و -- می‌توانند به دو صورت پیشوند [1] و پسوند [2] استفاده شود.
- برای سربارگذاری چنین عملگرهایی که هم به صورت پیشوند و هم به صورت پسوند مورد استفاده قرار می‌گیرند، کامپایلر قرارداد کرده است که در حالت پیشوند تابع سربارگذاری عملگر بدون ورودی است: `operator++()` , `operator--()`
- و در حالت پسوند تابع سربارگذاری عملگر یک ورودی عدد صحیح می‌گیرد: `operator++(int)` , `operator--()`

[1] prefix
[2] postfix

سربارگذاری عملگرها

```
۱  class complex {
۲  public :
۳      // prefix ++
۴      complex& operator++() {
۵          re++; // first increment and then return
۶          return *this;
۷      }
۸      // postfix ++
۹      complex operator++(int) {
۱۰         complex res(*this);
۱۱         re++;
۱۲         return res; // return the current result and increment
۱۳     }
۱۴ };
```

سربارگذاری عملگرها

- فرض کنید می‌خواهیم عملگر >> را برای کلاس اعداد مختلط سربارگذاری کنیم تا اعداد مختلط را توسط این عملگر بر روی خروجی استاندارد چاپ کنیم.
- این عملگر یک عملگر دوتایی است که ورودی اول آن یک شیء از کلاس `ostream` و ورودی دوم آن یک شیء است که در اینجا می خواهیم آن را از کلاس `complex` تعریف کنیم.
- از آنجایی که ورودی اول این عملگر از کلاس `ostream` است و ما به این کلاس دسترسی نداریم، بنابراین این تابع را باید خارج از کلاس `complex` تعریف کنیم، زیرا اگر آن را در کلاس `complex` تعریف کنیم، ورودی آن از نوع اعداد مختلط خواهد بود.

سربارگذاری عملگرها

```
۱ ostream& operator<<(ostream &out, const complex& c) {
۲     // we are modifying out, so it cannot be constant
۳     out << c.re << showpos << c.im << "i";
۴     return out;
۵ }
```

سربارگذاری عملگرها

- همچنین برای دسترسی به اعضای خصوصی کلاس complex در تابع operator>>، آن را به عنوان یک تابع دوست در کلاس complex تعریف می‌کنیم.

```
۱ class complex {
۲ public:
۳     ...
۴     friend ostream& operator<<(ostream &, complex&);
۵ }
```

سربارگذاری عملگرها

- به طور مشابه عملگر << را نیز برای کلاس اعداد مختلط سربارگذاری می‌کنیم تا اعداد مختلط را توسط این عملگر از روی ورودی استاندارد دریافت کنیم.

- این عملگر یک عملگر دوتایی است که ورودی اول آن یک شیء از کلاس `istream` و ورودی دوم آن یک شیء است که در اینجا می خواهیم آن را از کلاس `complex` تعریف کنیم.

```
۱  class complex {
۲  public:
۳      ...
۴      friend istream& operator>>(istream &, complex&);
۵  }
۶  istream& operator>>(istream &in, complex& c) {
۷      in >> c.re;
۸      in >> c.im
۹      return in;
۱۰ }
```

سربارگذاری عملگرها

- عملگرهای تبدیل نوع [1] برای تبدیل یک نوع به نوع دیگر استفاده می‌شوند.
- حال فرض کنید می‌خواهیم نوع عدد مختلط را با استفاده از عملگر تبدیل نوع `double()` به یک عدد اعشاری تبدیل کنیم و منظور ما از این تبدیل استخراج قسمت حقیقی عدد مختلط است.

[1] type casting

سربارگذاری عملگرها

- عملگر `()double` را باید برای کلاس اعداد مختلط به صورت زیر تعریف کنیم.

```
۱  class complex {
۲  public:
۳      ...
۴      operator double() {
۵          return re;
۶      }
۷  }
۸  complex c(1,2);
۹  double d1 = double(c);
۱۰ double d2 = (double)c;
۱۱ double d3 = c;
```

سربارگذاری عملگرها

- در صورتی که بخواهیم کاربر را مجبور کنیم که از عملگر تبدیل نوع استفاده کند و کاربر این تبدیل را به کامپایلر واگذار نکند، از کلیدواژهٔ `explicit` استفاده می‌کنیم.

```
۱  class complex {
۲  public:
۳      ...
۴      explicit operator double() {
۵          return re;
۶      }
۷  }
۸  complex c(1,2);
۹  double d1 = double(c);
۱۰ double d2 = (double)c;
۱۱ double d3 = c; // error
```

سربارگذاری عملگرها

- به طریق مشابه می‌توانیم عملگر تبدیل نوع string را برای نوع مختلط تعریف کنیم.

```
۱  class complex {
۲  public:
۳      ...
۴      operator string() {
۵          string res = to_string(re);
۶          if (im > 0) res+= "+";
۷          res += (to_string(im) + "i");
۸          return res;
۹      }
۱۰ }
۱۱ complex c(1,2);
۱۲ string s = (string)c;
```

سربارگذاری عملگرها

- در صورتی که بخواهیم با استفاده از عملگر تبدیل نوع، یک نوع را به نوع کلاس خود تبدیل کنیم، باید از سازنده استفاده کنیم.
- برای مثال فرض کنید می‌خواهیم یک عدد اعشاری را با استفاده از عملگر `complex()` به یک عدد مختلط تبدیل کنیم.
- در اینصورت می‌توانیم بنویسیم `complex(5.6)` و یا `5.6(complex)`.

سربارگذاری عملگرها

- در این موارد سازندهٔ کلاس complex اگر با ورودی عدد اعشاری تعریف شده باشد، فراخوانی می‌شود.

```
۱  class complex {
۲  public:
۳      ...
۴      complex(double r) :re{r}, im{0} {}
۵  };
۶  complex c1(1,2);
۷  // using the constructor,
۸  // 0 is implicitly type-casted to complex
۹  if (c1 == 0) { ... }
۱۰ complex c2;
۱۱ // using the constructor,
۱۲ // 5.6 is explicitly type-casted to complex
۱۳ c2 = c1 + (complex)5.6;
```

سربارگذاری عملگرها

- اگر بخواهیم تبدیل یک نوع به نوع کلاس مورد نظر ما به طور خودکار توسط کامپایلر انجام نشود، از کلیدواژهٔ `explicit` استفاده می‌کنیم.

```
۱ class complex {
۲ public:
۳     ...
۴     explicit complex(double r) :re{r}, im{0} {}
۵ };
۶ complex c1 = 10; // error
```

- پس برای جمع دو عدد اعشاری و مختلط اکنون دو راه وجود دارد. اول اینکه از عملگر جمع برای جمع دو عدد اعشاری و مختلط استفاده کنیم و دوم اینکه عدد اعشاری را با استفاده از سازنده به مختلط تبدیل کنیم و با استفاده از عملگر جمع برای دو عدد مختلط آن دو عدد را با هم جمع کنیم.

```
// 1. use operator+(double, complex)
// complex = double + complex
c2 = 1.6 + c1;
// 2. use constructor to cast double to complex
// complex = (complex)double + complex
c2 = (complex)1.6 + c1;
```

سربارگذاری عملگرها

- همچنین برای برای جمع دو عدد اعشاری و مختلط و بازگرداندن یک عدد اعشاری سه راه وجود دارد. اول اینکه از عملگر جمع برای جمع دو عدد اعشاری و مختلط استفاده کنیم و دوم اینکه عدد اعشاری را با استفاده از سازنده به مختلط تبدیل کنیم و با استفاده از عملگر جمع برای دو عدد مختلط آن دو عدد را با هم جمع کنیم.

```
// 1. use operator+(double, complex) and operator double()
// double = (double)(double + complex)
d = (double)(1.6 + c1);
// 2. use operator double()
// double = (double + (double)complex)
d = 1.6 + (double)c1;
// 3. use constructor to cast double to complex and operator double()
// double = (double)((complex)double + complex)
d = (double)((complex)1.6 + c1);
```

سربارگذاری عملگرها

- حال فرض کنید می‌خواهیم یک عدد اعشاری بزرگ (long double) را با یک عدد مختلط جمع کنیم. در این صورت کامپایلر دو راه پیش رو دارد. می‌تواند عدد اعشاری بزرگ را به اعشاری تبدیل کند و سپس آن را با عدد مختلط جمع کند. و یا می‌تواند عدد مختلط را به اعشاری تبدیل کند و سپس آن را با عدد اعشاری بزرگ جمع کند. کامپایلر در این مورد نمی‌تواند تصمیم بگیرد و بنابراین پیام خطا ارسال می‌کند.

```
۱ d = 1.6L + c1; // error : use of operator + is ambigious
۲ d = (double)1.6L + c1; // operator+(double,complex) is defined
۳ d = 1.6L + (double)c1; // operator+(long double, double) is defined
```

سربارگذاری عملگرها

- برای چاپ یک عدد مختلط نیز دو راه حل وجود دارد. یا از عملگر درج برای چاپ یک عدد مختلط استفاده کنیم و یا یک عدد مختلط را به یک رشته تبدیل و سپس رشته را چاپ کنیم.

```
// 1. use operator<<(ostream, complex)
cout << c1;
// 2. use operator string()
cout << (string)c1;
```

سربارگذاری عملگرها

- حال فرض کنید می‌خواهیم عملگر زیرنویس [1] [] را برای کلاس وکتور سربارگذاری کنیم، به گونه‌ای که با فراخوانی این عملگر بر روی شیء یک کلاس، یکی از اعضای وکتور متناسب با مقدار درون عملگر بازگردانده شود.
- بنابراین می‌خواهیم از این عملگر به صورت زیر استفاده کنیم.

```
۱ class Vector {
۲     double* elem;
۳     int sz;
۴ public:
۵     Vector(int s) :elem{new double[s]}, sz{s} { }
۶ };
۷ Vector v(10);
۸ v[0] = 6;
۹ cout << v[0];
```

[1] subscript

سربارگذاری عملگرها

- عملگر زیرنویس [] را به صورت زیر سربارگذاری می‌کنیم.

```
۱ class Vector {
۲ public:
۳     double& operator[](int i) { return elem[i]; }
۴     ...
۵ };
۶ Vector v(10);
۷ v[0] = 6;
۸ cout << v[0];
```

سربارگذاری عملگرها

- حال فرض کنید می‌خواهیم از عملگر زیرنویس برای یک شیء ثابت استفاده کنیم.

```
۱ class Vector {
۲ public:
۳     double& operator[](int i) { return elem[i]; }
۴     ...
۵ };
۶ const Vector v(10);
۷ cout << v[0]; // error
```

- از آنجایی که شیء v یک شیء ثابت است، نمی‌توانیم یک تابع غیرثابت (در اینجا عملگر زیرنویس) را برای آن فراخوانی کنیم.
- از طرفی اگر تابع سربارگذاری عملگر زیرنویس را ثابت تعریف کنیم، امکان مقدار دهی عناصر وکتور به صورت `v[0] = 6` را نخواهیم داشت.

سربارگذاری عملگرها

- بنابراین باید تابع عملگر زیرنویس را به صورت ثابت و غیرثابت سربارگذاری کنیم و کامپایلر نیز این اجازه را به ما می‌دهد، گرچه ورودی هر دو تابع یکسان است.

```
۱  class Vector {
۲  public:
۳      double& operator[](int i) { return elem[i]; }
۴      double operator[](int i) const { return elem[i]; }
۵      ...
۶  };
۷  const Vector v1(10);
۸  cout << v1[0];
۹  Vector v2(10);
۱۰ v2[0] = 6;
۱۱ cout << v2[0]
```

سربارگذاری عملگرها

- می‌توانیم عملگر <- را نیز برای این کلاس تعریف می‌کنیم.

```
۱  class Element {
۲  public :
۳      double e[100];
۴  };
۵  class Vector {
۶  private:
۷      Element * element;
۸  public:
۹      Vector() : element{new Element()} {}
۱۰     Element * operator->() { return element; }
۱۱ };
۱۲ Vector v;
۱۳ cout << v->e[2];
```

سربارگذاری عملگرها

- عملگر = مانند توابع سازنده و سازندهٔ کپی به طور پیش‌فرض برای همهٔ کلاس‌ها تعریف شده است.
- در تعریف پیش‌فرض این تابع همهٔ مقادیر اعضای کلاس از شیء مبدأ به شیء مقصد کپی می‌شوند.
- در مواردی که اعضای کلاس شامل اشاره‌گر هستند و نمی‌خواهیم اعضای کلاس عینا کپی شوند، بلکه می‌خواهیم برای اشاره‌گرها حافظه تخصیص بدهیم، عملگر مساوی را تعریف می‌کنیم.

سربارگذاری عملگرها

```
۱  class Vector {
۲  public:
۳      Vector& operator=(const Vector & v) {
۴          sz = v.sz;
۵          elem = new double[sz];
۶          for (int i=0; i<v.sz; i++)
۷              elem[i] = v.elem[i];
۸          return *this;
۹      }
۱۰ private:
۱۱     double* elem;  int sz;
۱۲ };
۱۳ Vector v1;
۱۴ Vector v2 = v1; // the copy constructor is called.
۱۵ Vector v3;
۱۶ v3 = v2; // v3.operator=(v2) is called.
```

سربارگذاری عملگرها

- عملگر فراخوانی تابع () را نیز می‌توانیم سربارگذاری کنیم.
- با شیئی که از این کلاس ساخته می‌شود، می‌توان مانند یک تابع رفتار کرده و آن را فراخوانی کرد یا آن را مانند اشاره‌گر به تابع به عنوان ورودی به توابع دیگر وارد کرد.
- به چنین اشیایی اشیای تابعی [1] یا فانکتور [2] نیز می‌گوییم.

[1] function object
[2] functor

سربارگذاری عملگرها

```
۱  class Linear {
۲  private:
۳      double a, b;
۴  public:
۵      Linear(double _a, double _b) : a(_a), b(_b) {}
۶      double operator()(double x) const {
۷          return a * x + b;
۸      }
۹  };
۱۰ Linear f{2, 1};  // Represents function 2x + 1.
۱۱ Linear g{-1, 0}; // Represents function -x.
۱۲ // f and g are objects that can be used like a function.
۱۳ double f_0 = f(0);
۱۴ double f_1 = f(1);
۱۵ double g_0 = g(0);
```

سربارگذاری عملگرها

- در کلاس صف، برای کار راحت‌تر با صف می‌توانیم عملگرهای زیر را سربارگذاری کنیم.

```
۱ class Queue {
۲ public:
۳   Queue& operator,(const int& data) { push(data); return *this; }
۴   int operator()() { return pop(); }
۵   bool operator!() { return !empty(); }
۶ }
۷ Queue q1(100);
۸ q1,2,3,7,8,9; // push 2,3,7,8,9 into the queue
۹ while(!q1) { cout << q1() << " "; } // pop from the queue
```