به نام خدا

برنامه‌سازی پیشرفته

آرش شفیعی

مقدمه

تاریخچه

- در سال ۱۹۷۹ یعنی حدود ۷ سال پس از تکمیل زبان سی، بیارنه استراستروپ [1] کار بر روی زبانی جدید به نام زبان سی با کلاس‌ها [2] را در آزمایشگاه‌های بل [3] آغاز کرد. در آن زمان زبان Simula به عنوان زبانی که در آن مفاهیم شیءگرایی به کار می‌رفت استفاده میشد. برنامه‌سازان با استفاده از این زبان شیءگرا می‌توانستند برنامه‌های بسیار بزرگ را نظم دهند و خوانایی برنامه را افزایش دهند. از طرفی زبان C زبان بسیار پرکاربرد و کارامدی برای سیستم‌عامل یونیکس به حساب می‌آمد. بنابراین انگیزهٔ اصلی از طراحی زبان سی با کلاس‌ها ایجاد زبانی شیءگرا مانند سیمولا بود که به اندازهٔ سی کارامد باشد. در سال ۱۹۸۴ این زبان به C++ تغییر نام یافت. دلیل این نامگذاری این بود که سی++ همهٔ ویژگی‌های سی را دارا بود بنابراین زبان جدیدی نبود، پس به جای تغییر نام آن به زبان دی، این زبان سی++ نامیده شد. در سال ۱۹۸۵ اولین نسخه از کتاب زبان برنامه‌سازی سی++ [5] منتشر شد.

---

[1] Bjarne Stroustrup
[2] C with Classes
[3] AT&T Bell Labs
[5] The C++ Programming Language

مقدمه

- در زبان‌های شیءگرا [1] مانند سی++ برخلاف زبان‌های رویه‌ای [2] یک برنامه از تعدادی شیء ساخته شده است که با یکدیگر در ارتباط هستند. پس زبان مدلسازی این زبان‌ها به زبان انسان و جهانی که انسان در آن زندگی می‌کند و توسط آن می‌اندیشد نزدیک‌تر است. جهان تشکیل شده است از اجسام و مفاهیم که با یکدیگر در ارتباط اند. همینطور یک برنامه در زبان‌های شیءگرا تشکیل شده است از تعدادی شیء که با یکدیگر در ارتباط اند. یک شیء که متعلق به یک کلاس یا یک خانواده است در واقع اجسام و مفاهیم را مدلسازی و پیاده‌سازی می‌کند. هر شیء تعدادی ویژگی و تعدادی رفتار دارد. ویژگی‌ها خصوصیات و ماهیت یک شیء را تعیین می‌کنند و رفتارهای عملیاتی که آن شیء می‌تواند انجام دهد.

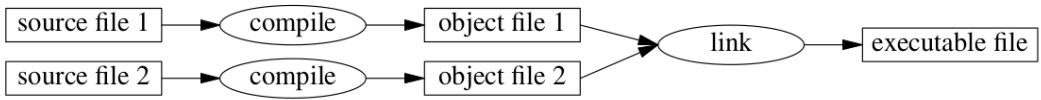
- به طور مثال یک دانشجوی معین در یک برنامهٔ سامانهٔ دانشگاهی نوشته شده توسط یک زبان شیء گرا در واقع شیئی است از کلاس یا خانوادهٔ دانشجو. این دانشجو ویژگی‌هایی دارد مانند نام و شمارهٔ دانشجویی و رفتارهایی دارد مانند ورود به سامانه و یا اخذ یا حذف درس.

- در زبان‌های رویه‌ای مانند سی، یک برنامه تشکیل شده است از تعدادی تابع که هر کدام عملیات معینی را انجام می‌دهند. پس زبان‌های شیءگرا به زبان‌های مدل‌سازی ما در جهان واقعی نزدیک‌تر اند.

[1] object-oriented programming languages

[2] procedural programming languages

- کامپایلر سی++ متن سورس برنامه را که در یک فایل سورس [1] نگهداری می‌شود به زبان ماشین ترجمه می‌کند و فایل‌هایی به نام آبجکت [2] می‌سازد که حاوی برنامه به زبان ماشین مقصد برای اجرا است.
- سپس فایل‌های آبجکت باید توسط لینکر [3] (یا پیونددهنده) به یکدیگر پیوند داده‌شوند و یک فایل اجرایی برای اجرا تهیه شود. معمولا یک برنامه از تعداد زیادی فایل سورس تشکیل شده است.



[1] source file
[2] object file
[3] linker

مقدمه

- در نهایت یک برنامه اجرایی [1] در قالب یک فایل اجرایی [2] برای یک سخت‌افزار مقصد تهیه می‌شود. این فایل قابل انتقال [3] نیست، بدین معنی که نمی‌توان آن را از یک سیستم‌عامل به یک سیستم‌عامل دیگر یا از یک سخت‌افزار به سخت‌افزار دیگر انتقال داد و اجرا کرد.
- زبان C++ تشکیل شده است از یک هستهٔ زبان [4] و یک کتابخانهٔ استاندارد [5] که در آن بسیاری از ابزارهای مورد نیاز برنامه‌نویسان (توسط خود زبان سی++) پیاده‌سازی شده‌اند.

[1] executable program
[2] executable file
[3] portable
[4] core language
[5] standard library

مقدمه

- یک برنامهٔ کوتاه برای چاپ یک عبارت بر روی خروجی استاندارد در زبان سی++ به صورت زیر نوشته می‌شود.

```
۱ #include <iostream>
۲ int main(){
۳     std::cout << "Hello, World!\n";
۴ }
```

- در اینجا `cout` یک شیء است (مانند یک متغیر از یک نوع معین) که عملگر >> برای آن تعریف شده است. با اعمال عملگر >> بر روی شیء `cout` رشته‌ای که در طرف دیگر عملگر نوشته شده است بر روی خروجی استاندارد چاپ می‌شود. در واقع جملهٔ `Hello, World!` بر روی استریم خروجی استاندارد [1] `std::cout` نوشته می‌شود.
- `#include <iostream>` امکانات استاندارد ورودی خروجی زبان را به متن برنامه اضافه می‌کند.

[1] standard output stream

مثال

```
۱  #include <iostream>
۲  // include (import) the declarations for the I/O stream library
۳  using namespace std;
۴  // make names from std visible without std::
۵
۶  double square(double x) {
۷      return x*x;
۸  } // square a double precision floating-point number
۹
۱۰ void print_square(double x) {
۱۱     cout << "the square of " << x << " is " << square(x) << "\n";
۱۲ }
۱۳
۱۴ int main() {
۱۵     print_square(1.234);
۱۶ } // print: the square of 1.234 is 1.52276
```

توابع

- معمولا برای انجام محاسبات طولانی، برنامه را به تعداد زیادی تابع تقسیم می‌کنیم. هر تابع وظیفهٔ انجام قسمتی از محاسبات را دارد.
- برای تعریف توابع، نوع دادهٔ خروجی، نام تابع و نوع داده‌های ورودی را مشخص می‌کنیم. در فراخوانی توابع نوع داده‌های ورودی و خروجی باید با نوع تعریف شده مطابقت داشته باشد.

```
۱  Elem * next_elem();
۲  // no argument; return a pointer to Elem (an Elem*)
۳  void exit(int);
۴  // int argument; return nothing
۵  double sqrt(double);
۶  // double argument; return a double
۷
۸  double s2 = sqrt(2);
۹  // call sqrt() with the argument double{2}
۱۰ double s3 = sqrt("three");
۱۱ // error: sqrt() requires an argument of type double
```

توابع

- وقتی چند تابع با نام یکسان تعریف شده باشند، ولی ورودی‌ها و خروجی‌های آنها از نوع‌های متفاوت تعریف شده باشد، کامپایلر در هنگام فراخوانی، از تعریف تابعی استفاده می‌کند که ورودی و خروجی‌های مناسب داشته باشد.

```
۱ void print(int); // takes an integer argument
۲ void print(double); // takes a floating-point argument
۳ void print(string); // takes a string argument
۴ void user() {
۵     print(42); // calls print(int)
۶     print(9.65); // calls print(double)
۷     print("Hello"); // calls print(string)
۸ }
```

- تعریف چند تابع با یک نام را سربارگذاری تابع [1] می‌نامیم.

[1] function overloading

توابع

- اگر در هنگام فراخوانی دو تابع با نام یکسان ابهامی وجود داشته باشد، کامپایلر پیام خطا صادر می‌کند.

```
۱ void print(int,double);
۲ void print(double,int);
۳ void user2() {
۴     print(0,0); // error : ambiguous
۵     print(0.0,0); //calls print(double,int)
۶ }
```

متغیرها و عملگرها

- همانند سی، در زبان C++ انواع دادهٔ اصلی، تعریف‌شده توسط کاربر، و مشتق شده وجود دارد.
- یکی از انواع داده که در زبان سی وجود ندارد، نوع `bool` است که یک مقدار منطقی درست یا نادرست را نگهداری می‌کند. متغیرهایی از این نوع یک بایت را در حافظه اشغال می‌کنند.
- عملگرهای C++ مانند عملگرهای زبان سی هستند.

مقداردهی اولیه

- متغیرها را به سه شکل می‌توان مقداردهی اولیه [1] کرد:

```
۱ double d1 = 2.3;
۲ double d2 {2.3};
۳ double d3 = {2.3};
```

- برای اطمینان از صحت مقداردهی اولیه معمولا از شکل دوم یا سوم استفاده می‌کنیم:

```
۱ int i1 = 7.8; // it becomes 7
۲ int i2 {7.8}; // error: floating-point to integer conversion
۳ int i3 = {7.8}; // error: floating-point to integer conversion
```

- مقادیر ثابت باید همیشه مقداردهی اولیه شوند.

```
۱ const int i4 {7};
```

[1] initialize

مقداردهی اولیه

- می‌توانیم از کلیدواژهٔ auto برای تعریف یک متغیر استفاده کنیم. نوع چنین متغیری با توجه به محتوای کد تعیین می‌شود. با استفاده از این کلیدواژه می‌توان برنامه‌های کوتاه‌تری نوشت. متغیری که با auto تعریف می‌شود، باید حتما مقداردهی اولیه شود.

```
1 auto b = true; // a bool
2 auto ch = 'x'; // a char
3 auto i = 123; // an int
4 auto d = 1.2; // a double
5 auto z = sqrt(y); // z has the type of whatever sqr t(y) returns
6 auto bb {true}; // bb is a bool
```

محدودهٔ تعریف

- یک نام (نام متغیر، نام تابع، ...) در یک محدوده [1] تعریف می‌شود.
- نام سراسری [2] نامی است که در همه جای برنامه تعریف شده است.
- نام محلی [3] نامی است که در یک بلوک از کد تعریف شده و قابل دسترسی است. یک بلوک از کد بین دو علامت آکولاد { } قرار دارد.
- نام اعضای کلاس [4] نامی است که در یک کلاس تعریف شده است (در مورد کلاس در آینده بیشتر صحبت خواهیم کرد).
- نام اعضای فضای نام [5] نامی است که در یک فضای نام تعریف شده است (در مورد فضای نام در آینده بیشتر صحبت خواهیم کرد)

[1] scope
[2] global name
[3] local name
[4] class member name
[5] namespace member name

محدودهٔ تعریف

```
۱  vector<int> vec;
۲  // vec is global (a global vector of integers)
۳  struct Record { string name; // ... };
۴  // name is a member of Record (a string member)
۵  void fct(int arg) // fct is global (a global function)
۶  // arg is a local variable for fct (an integer argument)
۷  {
۸      string motto {"Truth shall set you free"};
۹      // motto is local for fct
۱۰     auto p = new Record{"Ali"};
۱۱     // p points to an unnamed Record (created by new)
۱۲ }
```

محدودهٔ تعریف

- به جای استفاده از `malloc` و `free` در سی++ از دو کلیدواژهٔ `new` و `delete` استفاده می‌کنیم.

```
۱     Record * tp = new Record;
۲     int * p = new int;
۳     int * array = new int[20];
۴     delete p;
۵     delete tp;
۶     delete[] array;
```

- تا زمانی که برای یک مکان حافظه که توسط `new` تخصیص داده شده است، `delete` فراخوانی نشده، آن مکان در حافظه باقی می‌ماند، حتی اگر از حوزهٔ تعریف آن خارج شویم.

ثابت‌ها

- در سی++ دو نوع ثابت وجود دارد.
- const به معنی مقدار ثابتی است که می‌تواند در زمان اجرا مقداردهی اولیه شود. به طور مثال برای اشاره‌گرها در توابع وقتی می‌خواهیم مقدار آنها تغییر نکند از const استفاده می‌کنیم.
- constexpr ثابتی است که در زمان کامپایل باید مقدار اولیهٔ آن تعریف شده باشد.

```
constexpr int dmv = 17; // dmv is a named constant
int var = 17; // var is not a constant
const double sqv = sqrt(var);
// sqv is a named constant, possibly computed at run time
double sum(const vector<double> * v);
// sum will not modify its argument v, since it is constant
const double s1 = sum(v);
// OK: sum(v) is evaluated at run time
constexpr double s2 = sum(v);
// error : sum(v) is not a constant expression
```

ثابت‌ها

- اگر یک تابع توسط constexpr تعریف شود، مقدار آن در زمان کامپایل محاسبه می‌شود.

```
constexpr double square(double x) { return x*x; }
constexpr double max1 = 1.4*square(17);
// OK 1.4*square(17) is a constant expression
constexpr double max2 = 1.4*square(var);
// error : var is not a constant expression
const double max3 = 1.4*square(var);
// OK, may be evaluated at run time
```

- برای بهبود سرعت برنامه می‌توان توابع ساده را به صورت constexpr تعریف کرد.

آرایه، اشاره‌گر، و مرجع

- دنباله‌ای از داده‌ها در حافظه که همگی از یک نوع هستند را آرایه می‌نامیم.
- اشاره‌گر متغیری است که به یک خانه از حافظه اشاره می‌کند.
- با استفاده از عملگر ارجاع، می‌توان آدرس یک خانه از حافظه را استخراج و اشاره‌گری به یکی از اعضای آرایه تعریف کرد: `char * p = &v[3];`

آرایه، اشاره‌گر، و مرجع

- علاوه بر تعریف یک اشاره‌گر، در سی++ می‌توانیم یک متغیر مرجع نیز تعریف کنیم. یک متغیر مرجع را به صورت `type & var;` تعریف می‌کنیم. امکان تعریف آرایه‌های از متغیرهای مرجع وجود ندارد.
- پس از اینکه یک متغیر مرجع مقداردهی اولیه شد و به یک خانه از حافظه اشاره کرد، نمی‌توان مکانی در حافظه که آن متغیر به آن اشاره می‌کند را تغییر داد. پس یک مرجع برخلاف اشاره‌گر یک خانه در حافظه نیست که یک آدرس را نگهداری کند، بلکه نامی مستعار [1] است برای یک متغیر دیگر.
- همچنین برای دسترسی به مقدار یک متغیر مرجع به عملگر * نیازی نداریم.

```
1  int v[] = {0,1,2,3,4,5,6,7,8,9};
2  int & ref1;
3  // error : declaration of reference ref1 required initilizer
4  int & ref2 = v[2];
5  ref2 = 5; // here we change the value of v[2]
6  cout << "ref2: " << ref2 << " v[2]: " << v[2] << endl;
```

[1] alias

آرایه، اشاره‌گر، و مرجع

- استفاده اصلی متغیر مرجع برای فراخوانی با ارجاع است. بدین صورت دیگر نیازی به اشاره‌گر و اشغال فضای حافظه برای ذخیره‌سازی آدرس‌ها نخواهیم داشت.

```
۱ void swap(int & x, int & y) {
۲     int tmp = x; x = y; y = tmp;
۳ }
```

- هنگامی که می‌خواهیم از مرجع جهت کاهش سربار کپی استفاده کنیم ولی نمی‌خواهیم مقادیری که مرجع به آن اشاره می‌کند تغییر کنند، از کلیدواژهٔ `const` استفاده می‌کنیم.

```
۱ double sum(const vector<double>&)
```

آرایه، اشاره‌گر، و مرجع

- همچنین مقدار بازگشت یک تابع می‌تواند یک متغیر مرجع باشد.

```
int vals[] = { 2, 6, 1, 3, 5 , 4};

int & value(int i) {
    if (i >= 0 && i <= 5)
        return vals[i];
}

int a = value(3);
value(1) = 7; // we set vals[1] = 7
cout << value(4);
cin >> value(0);
```

آرایه، اشاره‌گر، و مرجع

- در سی++ می‌توانیم با استفاده از یک حلقه بر روی دامنه [1] توسط کلیدواژهٔ for به هر یک از اعضای یک آرایه به ترتیب از اولین عضو تا آخرین عضو دسترسی پیدا کنیم.

```
۱ void print()
۲ {
۳     int v[] = {0,1,2,3,4,5,6,7,8,9};
۴     for (auto x : v)
۵         cout << x << '\n'; // for each x in v
۶
۷     for (auto x : {10,21,32,43,54,65})
۸         cout << x << '\n';
۹ }
```

[1] range for statement

آرایه، اشاره‌گر، و مرجع

- توجه کنید که در عبارت `auto x : v` هر یک از اعضای v در متغیر x کپی می‌شوند. اگر بخواهیم از سربار این کپی بکاهیم، می‌توانیم از مرجع استفاده کنیم: `auto & x : v`
- همچنین با استفاده از مرجع می‌توانیم متغیر x را تغییر داده و در نتیجه اعضای آرایهٔ v را تغییر دهیم.

```
۱  void print() {
۲      int v[] = {0,1,2,3,4,5,6,7,8,9};
۳      for (auto & x : v) cout << x++ << " ";
۴      cout << endl;
۵      for (auto x : v)
۶          cout << x << " ";
۷      cout << endl;
۸      for (const int & x : v)
۹          cout << x << '\n';
۱۰ }
```

آرایه، اشاره‌گر، و مرجع

- وقتی یک اشاره‌گر به هیچ مکانی در حافظه اشاره نمی‌کند از کلیدواژهٔ `nullptr` برای مقداردهی اولیهٔ آن استفاده می‌کنیم.

```
۱  double * pd = nullptr;
۲  int x = nullptr; // error: nullptr is a pointer not an integer
۳
۴  int count_x(const char * p, char x) {
۵  // count the number of occurrences of x in p[]
۶  // p is assumed to point to
۷  //   a zero-terminated array of char (or to nothing)
۸      if (p==nullptr) return 0;
۹      int count = 0;
۱۰     for (;  *p != 0; ++p)
۱۱         if (*p == x)
۱۲             ++count;
۱۳     return count;
۱۴ }
```

انواع دادهٔ تعریف‌شده توسط کاربر

- یک ساختمان یا struct یک نوع دادهٔ تعریف شده توسط کاربر است که برای تعریف یک نوع داده ترکیب شده از داده‌هایی از انواع مختلف در حافظه تحت یک نام واحد به کار می‌رود.

```
۱  struct Vector {
۲      int sz; // number of elements
۳      double* elem; // pointer to elements
۴  };
۵  void vector_init(Vector& v, int s) {
۶      v.elem = new double[s]; // allocate an array of s doubles
۷      v.sz = s;
۸  }
۹  Vector v;
۱۰ vector_init(v,10);
```

- بر خلاف زبان سی در تعریف یک متغیر از یک ساختمان نیازی به واژهٔ struct نیست.

انواع دادهٔ تعریف‌شده توسط کاربر

- برای متغیرهایی از نوع ساختمان، اگر به صورت مرجع تعریف شده باشند توسط عملگر نقطه، و اگر به صورت اشاره‌گر تعریف شده باشند، توسط عملگر -> می‌توانیم به اعضای آنها دسترسی پیدا کنیم.

```
1 void f(Vector v, Vector& rv, Vector* pv) {
2     int i1 = v.sz; // access through name
3     int i2 = rv.sz; // access through reference
4     int i3 = pv->sz; // access through pointer
5 }
```

انواع دادهٔ تعریف‌شده توسط کاربر

- یک اجتماع یا union شبیه یک ساختمان است با این تفاوت که union تنها به اندازهٔ بزرگترین عضو خود در حافظه فضا اشغال می‌کند.
- از یک اجتماع زمانی استفاده می‌کنیم که از بین چندین متغیر در هر زمان فقط به یکی از آنها نیاز داشته باشیم.
- برای مثال فرض کنید یک ورودی Entry همیشه یا یک شماره دارد و یا یک اسم. پس این ورودی را بدین صورت تعریف می‌کنیم.

```
1 enum Type { str, num }; // a Type can hold values str and num
2 struct Entry {
3     Type t;
4     string s; // use s if t==str
5     int i; // use i if t==num
6 };
```

- در اینجا همیشه یکی از متغیرهای s یا i بلا استفاده می‌ماند.

انواع دادهٔ تعریف‌شده توسط کاربر

- توسط یک اجتماع می‌توانیم ورودی Entry را چنین تعریف کنیم.

```
union Value {
    string s;
    int i;
};
struct Entry {
    Type t;
    Value v; // use v.s if t==str; use v.i if t==num
    // there is no extra space for v.i in the memory
};
```

انواع دادهٔ تعریف‌شده توسط کاربر

- یک کلاس `class` اساسی‌ترین مفهوم در زبان‌های شیءگرا است.
- یک کلاس یک نوع داده را تعریف می‌کند. این نوع داده تعدادی متغیر به همراه تعدادی تابع که بر روی آن متغیرها تغییر اعمال می‌کنند را کپسوله‌سازی [1] یا لفافه‌بندی می‌کند.
- یک کلاس تعدادی اعضا [2] دارد که این اعضا می‌توانند داده، یا تابع باشند.
- یک نمونه از یک کلاس را یک شیء می‌نامیم.

[1] encapsulate
[2] member

انواع دادهٔ تعریف‌شده توسط کاربر

- یک کلاس دارای یک سازنده و یک مخرب است. یک سازنده تابعی است که در هنگام ساخته شدن یک شیء به صورت خودکار فراخوانی می‌شود و یک مخرب تابعی است که در هنگام تخریب یک شیء به طور خودکار فراخوانی می‌شود.
- در یک ساختمان، دسترسی به همهٔ اعضای ساختمان ممکن است. در یک کلاس برای داده‌ها سطح دسترسی تعریف می‌شود.
- اگر سطح دسترسی یک عضو کلاس `public` باشد، می‌توان به آن عضو از طریق شیء ساخته شده از کلاس دسترسی پیدا کرد. در صورتی که سطح دسترسی یک عضو `private` باشد، آن عضو توسط شیء ساخته شده از آن قابل دسترسی نیست.

انواع دادهٔ تعریف‌شده توسط کاربر

- کلاس Vector را می‌توان به صورت زیر به صورت یک کلاس تعریف کرد.

```
۱  class Vector {
۲  public:
۳      Vector(int s) { elem = new double[s]; sz = s; }
۴      double& value(int i) { return elem[i]; }
۵      int size() { return sz; }
۶      ~Vector() { delete[] elem; }
۷  private:
۸      double* elem; // pointer to the elements
۹      int sz; // the number of elements
۱۰ };
۱۱ Vector v(6); // a Vector with 6 elements
۱۲ cout << v.value(2);
```

- تابع سازندهٔ Vector همیشه به محض ساختن یک شیء از کلاس Vector فراخوانی می‌شود. همچنین تابع مخرب ~Vector همیشه به محض تخریب یک شیء کلاس فراخوانی می‌شود.

انواع دادهٔ تعریف‌شده توسط کاربر

- حال می‌توانیم بدین صورت از Vector استفاده کنیم.

```
double read_and_sum(int s) {
    Vector v(s); // make a vector of s elements
    // we cannot access v.elem and v.sz
    // and we cannot change their values directly
    for (int i=0; i!=v.size(); ++i)
        cin>>v.value(i); // read into elements
    double sum = 0;
    for (int i=0; i!=v.size(); ++i)
        sum+=v.value(i); // take the sum of the elements
    return sum;
}
```

انواع دادهٔ تعریف‌شده توسط کاربر

- علاوه بر enum که در سی برای نامگذاری تعدادی مقدار صحیح به کار می‌رود، در سی++ نوع دادهٔ enum class نیز وجود دارد.

- با استفاده از enum class مقدار دو enum متفاوت را نمی‌توان به متغیرهایی از نوع متفاوت انتساب کرد.

```
۱  enum class Color { red, blue, green };
۲  enum class Traffic_light { green, yellow, red };
۳  Color col = Color::red;
۴  Traffic_light light = Traffic_light::red;
۵
۶  Color x = red; // error : which red?
۷  Color y = Traffic_light::red; // error: that red is not a Color
۸  Color z = Color::red; // OK
۹
۱۰ int i = Color::red; // error: Color::red is not an int
۱۱ Color c = 2; // initialization error: 2 is not a Color
```

انواع دادهٔ تعریف‌شده توسط کاربر

- عملگرهای مقایسه برای `enum class` تعریف شده‌اند.

```
1 if (x > Color::blue) {
2     // do something
3 }
```

انواع دادهٔ تعریف‌شده توسط کاربر

- همچنین می‌توان برای یک `enum class` عملگرهای جدید تعریف کرد.

```
// prefix increment: ++
Traffic_light& operator++(Traffic_light& t) {
    switch (t) {
        case Traffic_light::green:
            return t=Traffic_light::yellow;
        case Traffic_light::yellow:
            return t=Traffic_light::red;
        case Traffic_light::red:
            return t=Traffic_light::green;
    }
}
Traffic_light light = Traffic_light::red;
Traffic_light next = ++light;
// next becomes Traffic_light::green
```

رشته‌ها

- در زبان سی++ از کتابخانهٔ استاندارد <string> برای عملیات بر روی رشته‌ها استفاده می‌کنیم. در این کتابخانه کلاس string تعریف شده است.
- برای این کلاس عملگرها و توابع مورد نیاز برای کار بر روی رشته‌ها تعریف شده‌اند.
- برای مثال، عملگر + برای الحاق رشته‌ها به یکدیگر تعریف شده است.

```
۱ string compose(const string& name, const string& domain) {
۲     return name + '@' + domain;
۳ }
۴ auto addr = compose("user","computer");
```

رشته‌ها

- عملگر =+ دو رشتهٔ سمت چپ و راست عملگر را به یکدیگر الحاق و در رشتهٔ سمت چپ عملگر ذخیره می‌کند.

```
string s1, s2;
s1 = s1 + '\n'; // append newline
s2 += '\n';
```

- تابع substr برای استخراج یک زیررشته از یک رشته، و تابع replace برای جایگزین کردن یک زیررشته استفاده می‌شود.

```
name = "C++ language"
string s = name.substr(4,8); // s = "language"
name.replace(4,8,"programming"); // name becomes "C++ programming"
```

رشته‌ها

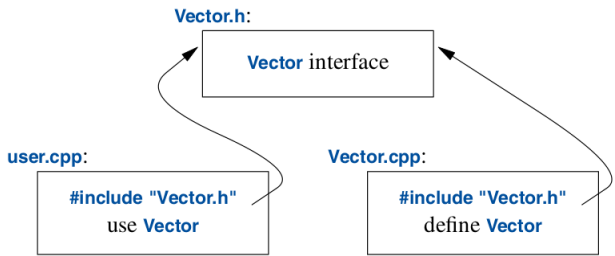
- عملگرهای دیگر از جمله = برای انتساب رشته‌ها، [] برای استخراج یک حرف از رشته، ==، =! برای مقایسه تساوی رشته‌ها، >، =>، <، =< برای مقایسه رشته‌ها بر اساس ترتیب الفبایی برای کلاس رشته سربارگذاری شده‌اند.
- برای تبدیل یک رشته به یک * char جهت استفاده از توابعی که در کتابخانه‌های سی پیاده‌سازی شده‌اند، می‌توان از تابع c_str() استفاده کرد.

```
۱ string s = "hello";
۲ // s.c_str() returns a pointer to 's characters
۳ printf("For people who like printf: %s\n",s.c_str());
۴ cout << "For people who like streams: " << s << '\n';
```

تقسیم‌بندی برنامه

- در یک برنامهٔ سی++ معمولا تعاریف توابع و کلاس‌ها در یک فایل به نام فایل سرتیتر یا هدر [1] و پیاده‌سازی تعاریف در یک فایل جداگانه به نام فایل کدمنبع یا فایل سورس [2] قرارگرفته می‌شوند.

- هر فایل سورس به طور جداگانه کامپایل می‌شود و در نهایت لینکر فایل‌های آبجکت تولید شده پس از کامپایل فایل‌های سورس را به هم پیوند می‌دهد و فایل اجرایی می‌سازد.



[1] header file
[2] source file

تقسیم‌بندی برنامه

- در یک برنامهٔ سی++ معمولا تعاریف توابع و کلاس‌ها در یک فایل به نام فایل سرتیتر یا هدر و پیاده‌سازی تعاریف در یک فایل جداگانه به نام فایل کدمنبع یا فایل سورس قرارگرفته می‌شوند.

```
// Vector.h:
class Vector {
public:
    Vector(int s);
    double& value(int i);
    int size();
private:
    double* elem;
    // elem points to an array of sz doubles
    int sz;
};
```

تقسیم‌بندی برنامه

- در یک برنامهٔ سی++ معمولا تعاریف توابع و کلاس‌ها در یک فایل به نام فایل سرتیتر یا هدر و پیاده‌سازی تعاریف در یک فایل جداگانه به نام فایل کدمنبع یا فایل سورس قرار گرفته می‌شوند.

```
// Vector.cpp:
#include "Vector.h" // get 'Vectors interface

// initialize members
Vector::Vector(int s) { elem = new double[s]; sz = s;}

double& Vector::value(int i) { return elem[i]; }

int Vector::size() { return sz; }
```

تقسیم‌بندی برنامه

- در یک برنامهٔ سی++ معمولا تعاریف توابع و کلاس‌ها در یک فایل به نام فایل سرتیتر یا هدر و پیاده‌سازی تعاریف در یک فایل جداگانه به نام فایل کدمنبع یا فایل سورس قرارگرفته می‌شوند.

```
// user.cpp:
#include "Vector.h"
#include <cmath>
// get 'Vectors interface
// get the standard-library math function interface
//     including sqrt()
double sqrt_sum(Vector& v) {
    double sum = 0;
    for (int i=0; i!=v.size(); ++i)
        sum+=std::sqrt(v.value(i));
    return sum;
}
```

فضای نام

- برای اینکه در یک برنامهٔ بسیار بزرگ تعداد نام‌ها زیاد است و نام‌ها ممکن است با یکدیگر مشابه باشند، در سی++ می‌توان فضای نام [1] تعریف کرد. بدین صورت نام‌ها با یکدیگر تداخل پیدا نمی‌کنند. در دو فضای نام می‌توانند فضای نام‌های مشابه وجود داشته باشند ولی در یک فضای نام، نام‌ها نمی‌توانند مشابه باشند.

```
۱  namespace List {
۲      class Vector { };
۳  };
۴
۵  namespace Euclidean {
۶      class Vector { };
۷  };
۸
۹  List::Vector lv;
۱۰ Euclidean::Vector ev;
```

[1] namespace

فضای نام

- همچنین می‌توان یک فضای نام را با استفاده از کلیدواژهٔ using به برنامه افزود. در اینصورت همهٔ نام‌ها در آن فضای نام را می‌توان بدون استفاده از نام فضای نام استفاده کرد.

```
std::string str1;

using namespace std;

string str;
cout << str;
```

- در صورتی که بخواهیم تنها از یکی از نام‌ها در یک فضای نام استفاده کنیم، می‌توانیم با استفاده از کلیدواژهٔ using تنها آن نام را به برنامه بیفزاییم.

```
using std::string;
string str; // OK
cout << str; // error : cout is undeclared
```

فضای نام

- فضاهای نام می‌توانند همچنین تودرتو باشند. بدین ترتیب می‌توانیم در یک فضای نام یک فضای نام دیگر تعریف کنیم.

```
۱  namespace N1 {
۲      int i;
۳      namespace N2 {
۴          int i;
۵          int j;
۶      }
۷  }
۸
۹  N1::i = 2;
۱۰ N1::N2::i = 3
۱۱ N1::N2::j = 4;
```

فضای نام

- همچنین فضاهای نام می‌توانند به صورت گسسته در فایل‌ها مختلف تعریف شوند.

```
۱  // file1.h
۲  namespace N1 {
۳      int i;
۴  }
۵
۶  // file2.h
۷  namespace N1 {
۸      int j;
۹  }
۱۰
۱۱ // main.cpp
۱۲ N1::i = 2;
۱۳ N1::j = 3;
```