به نام خدا

برنامه‌سازی پیشرفته

آرش شفیعی

برنامه‌سازی عمومی

برنامه‌سازی عمومی و قالب‌ها

- یکی از قابلیت‌هایی که در زبان سی++ وجود دارد، برنامه‌سازی عمومی [2] است که توسط قالب‌ها [3] فراهم شده است.
- با استفاده از این قابلیت، یک تابع یا یک کلاس می‌تواند به جای اینکه برای یک نوع خاص داده تعریف شود، برای همهٔ انواع داده تعریف شود.
- در صورتی که یک تابع به صورت عمومی با استفاده از یک قالب تعریف شود، یک یا چند ورودی مورد نظر و/یا خروجی تابع می‌توانند از هر نوع داده‌ای باشند.
- همچنین در صورتی که یک کلاس به صورت عمومی تعریف شود، یک یا چند عضو کلاس یا ورودی و خروجی برخی از توابع آن می‌توانند از هر نوع داده‌ای باشند.
- وقتی از برنامه‌سازی عمومی استفاده می‌کنیم، یک متغیر مانند `T` یا `Y` به عنوان یک متغیر عمومی تعریف می‌شود و این متغیر در زمان کامپایل می‌تواند با هر نوع داده‌ای مانند `int`، `double`، `string` جایگزین شود.

[2] generic programming
[3] templates

برنامه‌سازی عمومی و قالب‌ها

- برای مثال فرض کنید می‌خواهیم تابعی داشته باشیم که یک عنصر را در یک آرایه جستجو کند. تابعی به صورت زیر برای آرایه‌هایی از نوع عدد صحیح می‌نویسیم.

```
۱ int search(int * list, int length, int key) {
۲     for (int i=0; i< length; i++) {
۳         if (list[i] == key)
۴             return i;
۵     }
۶     return -1;
۷ }
```

برنامه‌سازی عمومی و قالب‌ها

- حال فرض کنید می‌خواهیم تابعی مشابه برای اعداد اعشاری داشته باشیم. باید این تابع را برای اعداد اعشاری نیز تعریف کنیم. با این که هر دوی این توابع کاری مشابه انجام می‌دهند، اما نیاز به تعریف مجدد آن وجود دارد.
- همچنین اگر بخواهیم این تابع را برای نوع‌های دیگری که توسط کاربر تعریف شده‌اند استفاده کنیم، باید توابعی جدید برای آنها نیز تعریف کنیم. فرض کنید نویسندهٔ این تابع در یک کتابخانه تابع خود را تعریف کرده و در اختیار استفاده کنندگان قرار داده است. در اینصورت الزاما نویسندهٔ تابع از نوع داده‌هایی که توسط آنها تابع فراخوانی خواهد شد اطلاع نخواهد داشت.

برنامه‌سازی عمومی و قالب‌ها

- پس نیاز داریم تابع جستجو را برای یک نوع عمومی تعریف کنیم و نمی‌دانیم در زمان فراخوانی این تابع توسط چه نوعی فراخوانی خواهد شد.
- برای برنامه‌سازی عمومی در چنین مواردی زبان سی++ قالب‌ها را عرضه کرده است.
- با استفاده از یک قالب می‌توانیم تابع جستجو را به صورت زیر تعریف کنیم.

```
۱ template<typename T>
۲ int search(T * list, int size, T key) {
۳     for (int i=0; i< size; i++) {
۴         if (list[i] == key)
۵             return i;
۶     }
۷     return -1;
۸ }
```

- سپس تابع را با استفاده از نوع مورد نظر فراخوانی می‌کنیم.

```
۱ int iarr[100]; double darr[100];
۲ search<int>(iarr, 100, 5);
۳ search<double>(darr, 100, 5.0);
۴ search(iarr, 100, 5); //ok
۵ search(darr, 100, 5.0); //ok
```

- در زمان کامپایل، کامپایلر دو تابع با ورودی `int` و `double` از تابع عمومی می‌سازد.

برنامه‌سازی عمومی و قالب‌ها

- در حالت کلی قبل از تابع از کلیدواژهٔ `template` استفاده می‌کنیم و نوع‌های عمومی که در تابع مورد استفاده قرار می‌گیرند را معرفی می‌کنیم.

```
template<typename T1, typname T2, typename T3 ...>
T1 function(T2 x, T3 y, ...) { ... }
```

- سپس در استفاده از تابع آن را با استفاده از نوع‌های داده‌ای مورد نیاز فراخوانی می‌کنیم.

```
int res; double x; float y;
res = function<int, double, float, ...>(x, y, ...);
res = function(x, y, ...); // also ok
```

- به ازای هر فراخوانی از یک تابع عمومی، کامپایلر تابعی با نوع‌های داده‌ای مورد نظر را می‌سازد. تعداد توابعی که کامپایلر از یک تابع عمومی می‌سازد وابسته به نوع داده‌ها در فراخوانی‌های آن تابع است.

برنامه‌سازی عمومی و قالب‌ها

- یک تابع که به صورت عمومی تعریف شده است، می‌تواند برای یک حالت خاص نیز تعریف شود.
- برای مثال تابع جستجو را می‌توان در حالت خاص برای نوع رشته به گونه‌ای دیگر تعریف کرد.

```
template<typename T>
int search(T * list, int size, T key) {
    ...
}
int search(string * list, int size, string key) {
    ...
}
```

برنامه‌سازی عمومی و قالب‌ها

- مشابه توابع، یک کلاس را نیز می‌توان به صورت عمومی با استفاده از قالب تعریف کرد.
- برای تعریف یک کلاس از نوع عمومی، قبل از تعریف تابع، از کلیدواژهٔ `template` استفاده می‌کنیم.

```
template<typename T1, typname T2, typename T3 ...>
class className {
    ...
    T1 variable;
    T2 function(T3 x, ...) { ... }
};
```

برنامه‌سازی عمومی و قالب‌ها

- برای مثال فرض کنید کلاسی برای پیاده‌سازی یک وکتور تعریف کرده‌ایم که تنها می‌تواند شامل اعداد صحیح باشد.

```
۱ class Vector {
۲ private:
۳     int list[100];
۴ public:
۵     push_back(int x) { ... }
۶ };
```

برنامه‌سازی عمومی و قالب‌ها

- برای اینکه این وکتور بتواند همهٔ نوع‌های داده‌ای را شامل شود، از قالب استفاده می‌کنیم.

```
template <typename T>
class Vector {
private:
    T list[100];
public:
    void push_back(T x) { ... }
};
```

- در صورتی که بخواهیم تابع push_back را خارج از تعریف کلاس تعریف کنیم، باید مجددا قالب را تعریف کنیم.

```
template <typename T>
void Vector<T>::push_back(T x) { ... }
```

برنامه‌سازی عمومی و قالب‌ها

- حال می‌توانیم شیئی از کلاس مورد نظر بسازیم.

```
۱ Vector<int> v1;
۲ v1.push_back(4);
۳ Vector<double> v2;
۴ v2.push_back(5.5);
```

برنامه‌سازی عمومی و قالب‌ها

- همچنین می‌توانیم یک کلاس عمومی را برای یک نوع خاص پیاده سازی کنیم. برای چنین کاری باید نام تابع را با آن نوع خاص تعیین و کلاس را مجددا جداگانه پیاده‌سازی کنیم.

```
template < >
class Vector<string> { ...
};
```

- وقتی شیئی از کلاس Vector<string> ساخته می‌شود، کامپایلر پیاده‌سازی رشته‌ای وکتور را که توسط کاربر تعیین شده به کار می‌برد.

برنامه‌سازی عمومی و قالب‌ها

- برای یک نوع عمومی می‌توان یک مقدار پیش فرض نیز تعیین کرد. بدین منظور، در تعریف قالب مقداری برای متغیر عمومی قرار می‌دهیم.

```
۱ template <typename T=int>
۲ class Vector { ...
۳ };
```

- حال در استفاده از کلاس وکتور می‌توانیم متغیر عمومی را مقدار دهی نکنیم.

```
۱ Vector< > v; // this is a vector of default type "int"
```

برنامه‌سازی عمومی و قالب‌ها

- علاوه بر نوع‌های داده‌ای، می‌توان یک مقدار را نیز به صورت عمومی تعریف کرد. پس ورودی یک قالب علاوه بر اینکه می‌تواند یک نوع عمومی باشد، می‌تواند یک متغیر نیز باشد.
- در مثال زیر، یک وکتور از یک نوع عمومی تعریف شده است و اندازهٔ وکتور هم به عنوان ورودی به قالب باید تعیین شود.

```
۱ template <typename T, int SIZE>
۲ class Vector { ...
۳     T data[SIZE];
۴ };
```

برنامه‌سازی عمومی و قالب‌ها

- حال در استفاده از کلاس برای ساختن شیء باید علاوه بر نوع داده‌های وکتور، اندازهٔ آن را نیز تعیین کنیم.

```
Vector<int, 100> v;
// this is a vector of data type int and of size 100
```

- دقت کنید برای مقادیر ورودی قالب نیز همانند نوع‌های داده‌ای، کامپایلر به ازای هر مقدار جدید یک نسخهٔ جدید از کلاس را در زمان کامپایل می‌سازد که سربارِی در زمان کامپایل ایجاد می‌کند.

برنامه‌سازی عمومی و قالب‌ها

- دقت کنید وقتی یک تابع از یک کلاس به صورت عمومی تعریف می‌شود، در زمان کامپایل، کامپایلر هیچ اطلاعی از نوع داده‌ای که در برنامه استفاده خواهد شد ندارد، بنابراین نمی‌تواند آن تابع را با نوع مورد نظر کامپایل کند و فایل آبجکت بسازد.
- بنابراین همهٔ توابع عمومی باید در فایل سرتیتر یا هدر تعریف شوند.

برنامه‌سازی عمومی و قالب‌ها

- پس به طور کلی از قالب برای دریافت یک نوع داده به عنوان پارامتر استفاده می‌کنیم.

```
template<typename T>
class Vector {
private:
    T* elem; // elem points to an array of sz elements of type T
    int sz;
public:
    explicit Vector(int s); // constructor: acquire resources
    ~Vector() { delete[] elem; } // destructor: release resources
    T& operator[](int i); // for non-const Vectors
    const T& operator[](int i) const; // for const Vectors
    int size() const { return sz; }
};
```

برنامه‌سازی عمومی و قالب‌ها

- عبارت `template<typename T>` بدین معنی است که برای همهٔ نوع‌های داده‌ای T تابع یا کلاس را تعریف کن.
- سپس وکتور را به صورت‌های زیر می‌توانیم تعریف کنیم.

```
۱ Vector<char> vc(200); // vector of 200 characters
۲ Vector<string> vs(17); // vector of 17 strings
۳ Vector<list<int>> vli(45); // vector of 45 lists of integers
```

برنامه‌سازی عمومی و قالب‌ها

- می‌توانیم از این وکتور به صورت زیر استفاده کنیم.

```
۱ void write(const Vector<string>& vs) {
۲     for (int i = 0; i!=vs.size(); ++i)
۳     cout << vs[i] << '\n';
۴ }
```

برنامه‌سازی عمومی و قالب‌ها

- اگر دو تابع begin و end برای کلاس وکتور تعریف شده باشند، می‌توانیم از ساختار حلقه بر روی دامنه استفاده کنیم.

```
template<typename T>
T* begin(Vector<T>& x) {
    // pointer to first element or nullptr
    return x.size() ? &x[0] : nullptr;
}
template<typename T>
T* end(Vector<T>& x) {
    // pointer to one-past-last element
    return x.size() ? &x[0]+x.size() : nullptr;
}
void write2(Vector<string>& vs) {
    for (auto& s : vs)
        cout << s << '\n';
}
```

فانکتور

- یکی از مفاهیمی که در سی++ استفاده می‌شود، شیءتابع [1] یا فانکتور [2] است با کلمهٔ تابعگون نیز ترجمه شده است. با استفاده از فانکتور می‌توانیم شیئی بسازیم و از آن شیء مانند یک تابع استفاده کنیم.
- کلاس زیر را در نظر بگیرید. عملگر () برای این کلاس تعریف شده است، بنابراین اگر شیئی از این کلاس ساخته شود، می‌توان آن را با استفاده از این عملگر فراخوانی کرد.

```
template<typename T>
class Less_than {
    const T val; // value to compare against
public:
    Less_than(const T& v) :val{v} { }
    // call operator
    bool operator()(const T& x) const { return x<val; }
};
```

[1] function object
[2] functor

فانکتور

- حال می‌توانیم شیئی از این کلاس به صورت زیر بسازیم.

```
// lti(i) will compare i to 42 using < (i<42)
Less_than lti {42};
// lts(s) will compare s to "Hello" using < (s<"Hello")
Less_than lts {"Hello"s};
// "Hello" is a C-style string,
// so we need <string> to get the right <
Less_than<string> lts2 {"Hello"};
```

- از این اشیاء می‌توانیم به صورت زیر استفاده کنیم.

```
void fct(int n, const string& s) {
    bool b1 = lti(n); // true if n<42
    bool b2 = lts(s); // true if s<"Backus"
    // ...
}
```

فانکتور

- از فانکتور می‌توانیم به صورت زیر استفاده کنیم. می‌خواهیم بر روی اعضای یک لیست دلخواه (که به صورت پارامتری تعیین می‌شود و می‌تواند هر نوع لیستی باشد)، یک تابع دلخواه (که آن نیز به صورت پارامتری تعیین می‌شود و می‌تواند هر نوع تابعی باشد) را فراخوانی کنیم.

```
template<typename C, typename P>
int count(const C& c, P func) {
    int cnt = 0;
    for (const auto& x : c)
        if (func(x)) cnt++;
    return cnt;
}
```

- حال می‌توانیم از این تابع به صورت زیر استفاده کنیم.

```
Vector<int> vec {12, 24, 43};
Less_than lti {42};
int c = count(vec, lti);
```

تابع لامبدا

- به جای استفاده از فانکتور، می‌توانیم از توابع لامبدا [1] استفاده کنیم.
- یک عبارت لامبدا [2] که به صورت زیر تعریف می‌شود، در واقع یک شیءتابع یا فانکتور بازمی‌گرداند. به عبارت دیگر عبارت لامبدا شیئی بازمی‌گرداند که می‌توان از آن به عنوان یک تابع استفاده کرد و آن تابع، تابع لامبدا نامیده می‌شود.

```
[<variables to capture>](<input variables>){ <function body> }
```

- در قسمت [] مشخص می‌کنیم چه متغیرهایی که در بیرون عبارت لامبدا تعریف شده‌اند را می‌خواهیم استفاده کنیم. در قسمت () لیست ورودی‌های تابع را ذکر می‌کنیم. و در قسمت { } بدنهٔ تابع را می‌نویسیم.

[1] lambda function
[2] lambda expression

تابع لامبدا

- برای مثال می‌خواهیم یک تابع لامبدا تعریف کنیم که عدد ورودی به تابع را با یک عدد معین مقایسه کند.
- این تابع را به صورت زیر می‌نویسیم.

```
1  auto lti = [&](int a){ return a<x; }
```

- عبارت [&] بدین معنی است که می‌خواهیم به همهٔ متغیرهای بیرون عبارت لامبدا دسترسی با ارجاع داشته باشیم. بنابراین متغیر x بیرون از عبارت تعریف شده است.
- همچنین می‌توانیم بنویسیم [=] بدین معنا که می‌خواهیم به متغیرهای بیرون از عبارت لامبدا دسترسی با کپی داشته باشیم.
- حال می‌توانیم تابع لامبدا به صورت `lti(n)` استفاده کنیم.
- این تابع را قبلا به صورت فانکتور با استفاده از کلاس `Less_than` تعریف کرده بودیم.

تابع لامبدا

- از آنجایی که در این تابع فقط به متغیر x نیاز داریم، بنابراین عبارت لامبدا را می‌توانیم به صورت زیر نیز تعریف کنیم.

```
1 auto lti = [&x](int a){ return a<x; }
```

تابع لامبدا

- فرض کنید می‌خواهیم تابعی بنویسیم که یک لیست دلخواه (که می‌تواند هر نوع لیستی باشد) دریافت کند و یک تابع دلخواه را (که می‌تواند هر نوع تابعی باشد) را دریافت کرده و تابع را بر روی همهٔ اعضای لیست فراخوانی کند.

```
// C is a container, and O is an operation
template<typename C, typename O>
void for_all(C& c, O op) {
    for (auto& x : c)
        op(x);
}
```

- حال این تابع را به صورت زیر استفاده می‌کنیم.

```
std::vector<int> v {20, 30, 40, 50};
for_all(v, [](int& n){ n *= 2; });
```