فصل
چهاردهم

الگوها

اهداف

- استفاده از الگوهای تابع در ایجاد راحتر توابع مرتبط.
- تمایز قائل شدن مابین الگوهای تابع و الگوهای تابع تخصصی شده.
- استفاده از الگوهای کلاس برای ایجاد گروهی از نوع‌های مرتبط.
- تمایز قائل شدن مابین الگوهای کلاس و الگوهای کلاس تخصصی شده.
- سربارگذاری الگوهای تابع.
- درک رابطه موجود مابین الگوها، دوستان، توارث و اعضای استاتیک.

رئوس مطالب

۱ -۱٤ مقدمه

۱٤-۲	**الگوهای تابع**
۱٤-۳	**سربارگذاری الگوهای تابع**
۱٤-٤	**الگوهای کلاس**
۱٤-٥	**پارامترهای بدون نوع و نوع‌های پیش‌فرض در الگوهای کلاس**
۱٤-٦	**الگوها و توارث**
۱٤-۷	**الگوها و دوستان**
۱٤-۸	**الگوها و اعضای استاتیک**

۱-۱٤ مقدمه

در این فصل، در ارتباط با یکی از ویژگیهای قدرتمند C++ که در استفاده مجدد از نرم‌افزار نقش دارد و بنام *الگوها* یا *قالب‌ها* شناخته می‌شود صحبت خواهیم کرد. الگوهای تابع و الگوهای کلاس به برنامه‌نویس امکان می‌دهند تا جنبه خاصی به یک بخش از کد، کل توابع مرتبط (که الگوهای تابع تخصصی شده نامیده می‌شوند) یا کل کلاس‌های مرتبط (که الگوهای کلاس تخصصی شده نامیده می‌شوند) اعطا کند. این تکنیک بنام *برنامه‌نویسی عمومی* شناخته می‌شود.

می‌توانیم یک الگوی تابع منفرد در یک تابع مرتب‌سازی آرایه بنویسیم، سپس با داشتن الگوی تابع تخصصی شده که توسط C++ تولید می‌شود می‌توان آرایه‌های از نوع int، float، string و غیره را مرتب کرد. در فصل ششم به معرفی الگوها پرداخته‌ایم. در این فصل مباحث دیگری به آن اضافه می‌کنیم.

می‌توانیم یک الگوی کلاس منفرد برای کلاس پشته بنویسیم، سپس همانند کلاس پشته int، کلاس پشته float، کلاس پشته string و غیره توسط C++ تولید می‌گردد.

باید به وجه تمایز مابین الگوها و الگوهای تخصصی شده توجه کرد: الگوهای تابع و الگوهای کلاس همانند شابلون یا استنسیل هستند که برای ترسیم استفاده می‌کنیم، الگوهای تابع تخصصی شده و الگوهای کلاس تخصصی شده همانند ترسیم‌های جداگانه‌ای هستند که تماماً همشکل بوده، اما می‌توانند برای مثال با رنگ‌های مختلفی ترسیم شوند.

در این فصل، به معرفی یک الگوی تابع و الگوی کلاس خواهیم پرداخت. همچنین به رابطه موجود مابین الگوها و سایر ویژگیهای C++ توجه می‌کنیم، ویژگیهای همانند سربارگذاری، توارث، دوستان و اعضای استاتیک. جزئیات و مکانیزم طراحی الگوها که در این فصل توضیح می‌دهیم براساس مقاله آقای Byarne Stroustrup بنام *Parameterized Types for C++* و انتشار یافته در کنفرانس *Proceedings of the USENIX C++* در دنور، کلورادو به سال 1998 است. این فصل خاص الگوها است.

۲-۱٤ الگوهای تابع

معمولاً سربارگذاری توابع عملیات مشابه یا یکسانی بر روی انواع مختلف داده انجام می‌دهد. اگر عملیات‌ها برای هر نوع یکسان باشند، می‌توان آنها را بسیار جمع و جورتر و مناسب‌تر و با استفاده از الگوهای تابع بیان کرد. در ابتدا، برنامه‌نویس یک تعریف منفرد از الگوی تابع می‌نویسد. بر پایه انواع آرگومان تدارک دیده شده بصورت صریح یا استنتاج شده از فراخوانی این تابع، کامپایلر مبادرت به تولید کد شی مجزا برای تابع می‌کند (یعنی الگوی تابع تخصصی شده) تا به فراخوانی هر تابع بطرز مناسبی پاسخ داده شود. در زبان C، اینکار با استفاده از ماکروها که با رهنمود **#define** تولید می‌شود، قابل انجام است. با این وجود، ماکروها می‌توانند تاثیرات جانبی خطرناکی را تولید کنند و به کامپایلر امکان بررسی نوع را نمی‌دهند. الگوهای تابع یک راه حل، کامل و فشرده همانند ماکروها عرضه می‌کنند اما از قابلیت بررسی نوع هم برخوردار هستند.

کلیه تعاریف الگوی تابع با کلمه کلیدی **template** و بدنبال آن یک لیست از پارامترهای الگو آغاز می‌شود که با کاراکترهای < و > احاطه می‌شوند. هم پارامتر الگو که نشاندهنده یک نوع است بایستی جلوتر از کلمات کلیدی **class** یا **typename** قرار داده شود، همانند

```
template< typename T >
```

یا

```
template< class ElementType >
```

یا

```
template< typename BorderType, typename FillType >
```

از پارامترهای نوع الگو که در تعریف الگوی تابع قرار دارند، برای مشخص کردن نوع آرگومان در تابع، نوع برگشتی تابع و اعلان متغیرهای موجود در درون تابع استفاده می‌شود. دقت کنید که از کلمات کلیدی **typename** و **class** برای مشخص کردن پارامترهای الگوی تابع استفاده شده است و در واقع به معنی «هر نوع توکار یا نوع تعریف شده توسط کاربر» است.

مثال: الگوی تابع printArray

اجازه دهید تا به بررسی الگوی تابع **printArray** در شکل ۱-۱۴، خطوط 15-8 بپردازیم. الگوی تابع **printArray** یک پارامتر الگو بنام **T** در خط 8 اعلان کرده است (**T** می‌تواند هر شناسه معتبری باشد) که برای نوع آرایه قابل چاپ توسط **printArray** است، از **T** بعنوان نوع پارامتر الگو یا پارامتر نوع یاد می‌شود. در بخش ۵-۱۴ با پارامترهای بدون نوع آشنا خواهید شد.

زمانی که کامپایلر احضار تابع **printArray** را در برنامه سرویس‌گیرنده (همانند خطوط 30, 35 و 40) تشخیص داد، با استفاده از قابلیت تفکیک سربارگذاری بهترین تعریف تابع **printArray** را پیدا می‌کند. در این مورد، تنها تابع **printArray** با تعداد مناسب پارامترها، الگوی تابع **printArray** است (خطوط -8

15). به فراخوانی تابع در خط 30 توجه کنید. کامپایلر مبادرت به مقایسه نوع اولین آرگومان **printArray** (یعنی **int *** در خط 30) با اولین پارامتر الگوی تابع **printArray** (یعنی **const T *** در خط 9) کرده و استنباط می‌کند که نوع پارامتر **T** را با **int** جایگزین سازد تا آرگومان مطابق با پارامتر گردد. سپس کامپایلر **int** را جانشین **T** در کل تعریف الگو کرده و تابع تخصصی شده **printArray** را پردازش می‌کند که می‌تواند یک آرایه از مقادیر صحیح را به نمایش در آورد. در برنامه شکل ۱-۱۴، کامپایلر سه تابع تخصصی شده از **printArray** ایجاد می‌کند، یکی که در انتظار آرایه از نوع **int**، یکی برای آرایه از نوع **double** و دیگری برای آرایه از نوع **char** است. برای مثال، الگوی تابع تخصصی شده برای نوع **int** بصورت زیر است:

```
void printArray( const int *array, int count )
{
    for ( int i = 0; i < count; i++ )
        cout << array[i] << " ";
    cout << endl;
} //end function printArray
```

نام یک پارامتر الگو می‌تواند فقط یکبار در لیست پارامتری الگو در سرآیند الگو اعلان شود، اما می‌تواند به دفعات در سرآیند و بدنه تابع بکار گرفته شود. لازم نیست تا اسامی پارامتر الگو در میان الگوهای تابع منحصر بفرد باشد.

```
1  // Fig 14.1: fig14_01.cpp
2  // Using template functions.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // function template printArray definition
8  template< typename T >
9  void printArray( const T *array, int count )
10 {
11    for ( int i = 0; i < count; i++ )
12       cout << array[ i ] << " ";
13
14    cout << endl;
15 } // end function template printArray
16
17 int main()
18 {
19    const int aCount = 5; // size of array a
20    const int bCount = 7; // size of array b
21    const int cCount = 6; // size of array c
22
23    int a[ aCount ] = { 1, 2, 3, 4, 5 };
24    double b[ bCount ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
25    char c[ cCount ] = "HELLO"; // 6th position for null
26
27    cout << "Array a contains:" << endl;
28
29    // call integer function-template specialization
30    printArray( a, aCount );
31
32    cout << "Array b contains:" << endl;
33
34    // call double function-template specialization
35    printArray( b, bCount );
36
```

```
37    cout << "Array c contains:" << endl;
38
39    // call character function-template specialization
40    printArray( c, cCount );
41    return 0;
42 } // end main
```

```
Array a contains:
1 2 3 4 5
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
H E L L O
```

شکل ۱-۱٤ | تخصصی کردن الگوی تابع prinArray.

در شکل ۱-۱۴ به بررسی الگوی تابع **printArray** پرداخته شده است (خطوط 15-8). برنامه با اعلان پنج عنصر آرایه **a** از نوع **int**، هفت عنصر آرایه b از نوع **double** و شش عنصر آرایه **c** از نوع **char** آغاز می‌شود (به ترتیب خطوط 25-23). سپس برنامه با فراخوانی **printArray** مبادرت به چاپ هر آرایه می‌کند، یکبار با آرگومان اول **a** از نوع *** int** (خط 30)، یکبار با آرگومان اول **b** از نوع *** double** (خط 35) و یکبار با آرگومان اول **c** از نوع *** char** (خط 40). برای مثال، با فراخوانی موجود در خط 30، کامپایلر استنتاج می‌کند که **T** یک **int** بوده و نمونه تخصصی از تابع الگوی **printArray** ایجاد می‌کند که در آن نوع پارامتر **T**، صحیح **(int)** است. فراخوانی موجود در خط 35 سبب می‌شود که کامپایلر استنتاج کند که **T** یک **double** بوده و نمونه دوم تخصصی شده از تابع الگوی **printArray** را ایجاد می‌کند که در آن نوع پارامتر **T** از نوع **double** است. فراخوانی موجود در خط 40 سبب می‌شود که کامپایلر استنتاج کند که **T** از نوع **char** است و نمونه سومی از تابع تخصصی شده **printArray** ایجاد می‌کند که در آن **T** از نوع **char** می‌باشد. نکته مهم در اینجاست که اگر **T** (خط 8) نشاندهنده یک نوع تعریف شده توسط کاربر باشد (که در شکل ۱-۱۴ این چنین نیست)، بایستی مبادرت به سربارگذاری عملگر درج برای آن نوع می‌کردیم، در غیر اینصورت، اولین عملگر درج در خط 12 کامپایلر نمی‌شود.

در این مثال، مکانیزم الگو سبب می‌شود تا برنامه‌نویس مجبور به نوشتن سه تابع مجزای سربارگذاری شده با نمونه‌های اولیه زیر نشود:

```
void printArray( const int *,int );
void printArray( const double *,int );
void printArray( const char *,int );
```

که همگی از کد یکسانی بجز نوع T استفاده می‌کنند (همانطوری که در خط 9 بکار گرفته شده است).

۱٤-۳ سربارگذاری الگوهای تابع

الگوهای تابع و سربارگذاری با یکدیگر مرتبط هستند. الگوی تابع تخصصی شده از یک الگوی تابع تولید می‌شود که همگی دارای نام یکسان هستند، از اینرو کامپایلر با استفاده از تفکیک‌پذیری سربارگذاری مبادرت به فراخوانی تابع مناسب می‌کند.

به چندین روش می‌توان یک الگوی تابع را سربارگذاری کرد. می‌توانیم الگوهای تابع دیگر تدارک ببینیم که دارای نام مشابه بوده اما در پارامترهای تابع با هم تفاوت داشته باشد. برای مثال، الگوی تابع **printArray** در شکل ۱-۱۴ می‌توانست با الگوی تابع دیگر **printArray** با پارامترهای اضافی **lowSubscript** و **highSubscript** که تعیین می‌کنند چه محدوده‌ای در آرایه چاپ شود، سربارگذاری گردد.

همچنین یک الگوی تابع می‌تواند با تدارک دیدن توابع غیر الگو با نام تابع مشابه اما متفاوت در آرگومان‌های تابع، سربارگذاری گردد. برای مثال، الگوی تابع **printArray** در شکل ۱-۱۴ می‌توانست با یک نسخه غیر الگو که خاص چاپ آرایه‌ای از رشته‌های کاراکتری بصورت مرتب با فرمت جدولی است، سربارگذاری شود.

کامپایلر با استفاده از یک روش تطبیق دهنده تعیین می‌کند که کدام تابع به هنگام احضار، فراخوانی گردد. ابتدا، کامپایلر تمام الگوهای تابع را که مطابق با نام تابع فراخوانی شده هستند، را یافته و بر پایه آرگومان‌های موجود در تابع فراخوانی شده، مبادرت به ایجاد توابع تخصصی شده می‌کند. سپس کامپایلر تمام توابع متداول را که مطابق با نام برده شده باشند، پیدا می‌کند. اگر یکی از توابع متداول یا الگوی تابع تخصصی شده بهترین مطابقت را با تابع فراخوانی شده داشته باشد، آن تابع یا تابع تخصصی شده بکار گرفته می‌شود. اگر هر دو تابع به یک میزان مطابقت داشته باشند، تابع متداول یا عادی انتخاب خواهد شد. اگر چندین مطابقت برای فراخوانی یک تابع رخ دهد، کامپایلر دچار ابهام شده و پیغام خطا صادر می‌کند.

۱۴-۴ الگوهای کلاس

درک مفهوم پشته مستقل از نوع آیتم‌های است که به آن وارد می‌شوند. با این وجود، در نمونه‌سازی یک پشته، بایستی نوع داده مشخص شود. چنین حالتی فرصت مناسبی برای بهره‌مند شدن از ویژگی استفاده مجدد از نرم‌افزار است. تنها چیزی که نیاز داریم توجه به اصل پشته و ایجاد کلاس‌های است که مرتبط با نوع‌های مختلف می‌باشند و با پشته کار می‌کنند. C++ این قابلیت را از طریق الگوهای کلاس فراهم آورده است.

الگوهای کلاس معروف به نوع‌های پارامتری شده هستند، چرا که آنها مستلزم یک یا چندین نوع پارامتر برای مشخص کردن نحوه بهینه‌سازی یک الگوی «کلاس کلی» بفرم یک الگوی کلاس تخصصی شده می‌باشند.

برنامه‌نویسی که مایل به تهیه نسخه‌های متفاوتی از الگوی کلاس تخصصی شده است، فقط یک تعریف از الگوی کلاس را کدنویسی می‌کند. هر بار که به یک الگوی کلاس تخصصی شده دیگر نیاز پیدا شود، برنامه‌نویس از عبارات بسیار مختصر و فشرده استفاده کرده و کامپایلر کد مورد نیاز برای آن را تولید

می‌کند. برای مثال، الگوی کلاس **Stack**، می‌تواند تبدیل به پایه‌ای برای ایجاد کلاس‌های متعدد **Stack** (همانند "پشته‌ای از مقادیر **double**"، "پشته‌ای از مقادیر **int**"، "پشته‌ای از مقادیر **char**"، "پشته‌ای از کارمندان" و غیره) در برنامه شود.

ایجاد الگوی کلاس Stack <T>

به تعریف الگوی کلاس **Stack** (پشته) در شکل ۲-۱۴ توجه کنید. ظاهر آن شبیه تعریف یک کلاس عادی می‌باشد، بجز سرآیند قرار گرفته در خط 6

```
Template< typename T >
```

عبارت فوق تصریح کننده تعریف الگوی کلاس با پارامتر نوع **T** است که همانند یک جانگهدار برای نوع کلاس **Stack** که ایجاد خواهد شد عمل می‌کند. نیازی نیست که برنامه‌نویس حتماً از شناسه **T** استفاده کند و می‌تواند از هر شناسه معتبر دیگری استفاده نماید. نوع عنصر ذخیره شده در این پشته در سرتاسر سرآیند کلاس **T** و تعریف توابع عضو بعنوان **T** شناخته می‌شود. همانطوری که خواهید دید، **T** می‌تواند به نوع مشخصی همانند **int** یا **double** تبدیل شود. با توجه به روش طراحی این الگوی کلاس، دو محدودیت برای نوع داده‌های غیربنیادین بکار رفته در این پشته در نظر گرفته شده است، آنها باید یک سازنده پیش‌فرض (برای استفاده در خط 44 به منظور ایجاد آرایه‌ای که عناصر پشته را ذخیره می‌کند) داشته باشند و بایستی از عملگر تخصیص پشتیبانی کنند (خط 55 و 69). تعریف تابع عضو از یک الگوی کلاس، الگوهای تابع هستند. تعریف تابع عضو که خارج از تعریف الگوی کلاس جای می‌گیرد، با سرآیند زیر آغاز می‌گردد:

```
template< typename T >
```

(خطوط 40, 51 و 65). از اینرو، هر تعریفی شباهت به تعریف یک تابع عادی دارد، بجز اینکه نوع عنصر پشته همیشه از نوع پارامتر **T** خواهد بود. از عملگر باینری تفکیک قلمرو به همراه نام الگوی کلاس **Stack<T>** بکار گرفته شده (خطوط 41، 52 و 66) تا تعریف تابع عضو به قلمرو الگوی کلاس پیوند زده شود. در این مورد، نام کلاس عمومی **Stack<T>** است. زمانیکه پشته **doubleStack** بصورت نوع **Stack<double>** نمونه‌سازی می‌شود، سازنده **Stack** الگوی تابع تخصصی شده از **new** برای ایجاد آرایه‌ای از عناصر نوع **double** برای عرضه پشته استفاده کرده است (خط 44). عبارت

```
stackPtr = new T[ size ];
```

در تعریف الگوی کلاس **Stack** توسط کامپایلر در الگوی کلاس در الگوی کلاس تخصصی شده **Stack<double>** بصورت زیر تولید می‌شود

```
stackPtr = new double[ size ];
```

```
1  // Fig. 14.2: Stack.h
2  // Stack class template.
3  #ifndef STACK_H
4  #define STACK_H
5
6  template< typename T >
```

```
7  class Stack
8  {
9  public:
10    Stack( int = 10 ); // default constructor (Stack size 10)
11
12    // destructor
13    ~Stack()
14    {
15       delete [] stackPtr; // deallocate internal space for Stack
16    } // end ~Stack destructor
17
18    bool push( const T& ); // push an element onto the Stack
19    bool pop( T& ); // pop an element off the Stack
20
21    // determine whether Stack is empty
22    bool isEmpty() const
23    {
24       return top == -1;
25    } // end function isEmpty
26
27    // determine whether Stack is full
28    bool isFull() const
29    {
30       return top == size - 1;
31    } // end function isFull
32
33 private:
34    int size; // # of elements in the stack
35    int top; // location of the top element (-1 means empty)
36    T *stackPtr; // pointer to internal representation of the Stack
37 }; // end class template Stack
38
39 // constructor template
40 template< typename T >
41 Stack< T >::Stack( int s )
42    : size( s > 0 ? s : 10 ), // validate size
43      top( -1 ), // Stack initially empty
44      stackPtr( new T[ size ] ) // allocate memory for elements
45 {
46    // empty body
47 } // end Stack constructor template
48
49 // push element onto Stack;
50 // if successful, return true; otherwise, return false
51 template< typename T >
52 bool Stack< T >::push( const T &pushValue )
53 {
54    if ( !isFull() )
55    {
56       stackPtr[ ++top ] = pushValue; // place item on Stack
57       return true; // push successful
58    } // end if
59
60    return false; // push unsuccessful
61 } // end function template push
62
63 // pop element off Stack;
64 // if successful, return true; otherwise, return false
65 template< typename T >
66 bool Stack< T >::pop( T &popValue )
67 {
68    if ( !isEmpty() )
69    {
70       popValue = stackPtr[ top-- ]; // remove item from Stack
71       return true; // pop successful
72    } // end if
73
74    return false; // pop unsuccessful
75 } // end function template pop
76
```

```
77 #endif
```

شکل ۲-۱٤ | الگوی کلاس Stack.

ایجاد یک راه‌انداز برای تست الگوی کلاس Stack<T>

حال اجازه دهید تا به بررسی راه‌اندازی بپردازیم (شکل ۳-۱۴) که الگوی کلاس **Stack** را بکار می‌گیرد. راه‌انداز کار را با نمونه‌سازی شی **doubleStack** با سایز 5 آغاز می‌کند (خط 11). این شی بصورت کلاس **Stack<double>** اعلان می‌شود. کامپایلر نوع **double** را با پارامتر نوع **T** در الگوی کلاس به منظور تولید کد منبع کلاس **Stack** از نوع **double** پیوند می‌دهد. اگرچه الگوها ارائه کننده مزیت استفاده مجدد از نرم‌افزار هستند اما بخاطر داشته باشید الگوی کلاس تخصصی شده مضاعفی در یک برنامه نمونه‌سازی می‌گردد (در زمان کامپایل)، با اینکه الگو فقط یکبار نوشته می‌شود.

خطوط 21-17 مبادرت به احضار تابع **push** برای قرار دادن مقادیر 1.1، 2.2، 3.3، 4.4 و 5.5 از نوع **double** به **doubleStack** می‌کنند. حلقه **while** زمانی خاتمه می‌یابد که راه‌انداز مبادرت به وارد کردن ششمین مقدار به **doubleStack** کند (حالتی که پشته پر است، چرا که حداکثر عنصری که می‌تواند این پشته نگهداری کند، پنج عنصر است). دقت کنید که تابع **push** زمانیکه قادر به وارد کردن مقداری به پشته نباشد، false برگشت می‌دهد.

خطوط 28-27 تابع **pop** را در حلقه **while** فراخوانی می‌کنند تا پنج مقدار از پشته خارج شود یا حذف گردد. زمانیکه راه‌انداز مبادرت به خارج کردن ششمین مقدار از پشته نماید و با توجه به اینکه **doubleStack** خالی است، حلقه خاتمه می‌یابد.

```
1  // Fig. 14.3: fig14_03.cpp
2  // Stack class template test program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Stack.h" // Stack class template definition
8
9  int main()
10 {
11    Stack< double > doubleStack( 5 ); // size 5
12    double doubleValue = 1.1;
13
14    cout << "Pushing elements onto doubleStack\n";
15
16    // push 5 doubles onto doubleStack
17    while ( doubleStack.push( doubleValue ) )
18    {
19       cout << doubleValue << ' ';
20       doubleValue += 1.1;
21    } // end while
22
23    cout << "\nStack is full. Cannot push " << doubleValue
24       << "\n\nPopping elements from doubleStack\n";
25
26    // pop elements from doubleStack
27    while ( doubleStack.pop( doubleValue ) )
28       cout << doubleValue << ' ';
29
30    cout << "\nStack is empty. Cannot pop\n";
```

```
31
32    Stack< int > intStack; // default size 10
33    int intValue = 1;
34    cout << "\nPushing elements onto intStack\n";
35
36    // push 10 integers onto intStack
37    while ( intStack.push( intValue ) )
38    {
39       cout << intValue << ' ';
40       intValue++;
41    } // end while
42
43    cout << "\nStack is full. Cannot push " << intValue
44       << "\n\nPopping elements from intStack\n";
45
46    // pop elements from intStack
47    while ( intStack.pop( intValue ) )
48       cout << intValue << ' ';
49
50    cout << "\nStack is empty. Cannot pop" << endl;
51    return 0;
52 } // end main
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

شکل ۳-۱۴ | برنامه تست الگوی کلاس Stack.

خط 32 مبادرت به نمونه‌سازی پشته **intStack** با اعلان زیر می‌کند

```
Stack< int > intStack;
```

چون سایزی مشخص نشده است، سایز پیش‌فرض 10 برای سازنده پیش‌فرض در نظر گرفته می‌شود (خط 10 از شکل ۲-۱۴). حلقه موجود در خطوط 41-37 و احضار تابع **push** سبب می‌شود تا مقادیر وارد **inStack** شوند تا زمانیکه پشته پر گردد. سپس حلقه خطوط 48-47 و احضار تابع **pop** سبب می‌شود تا مقادیر از intStack تا خالی شدن پشته ادامه یابد.

ایجاد الگوهای تابع برای تست الگوهای کلاس Stack<T>

اگر دقت کنید متوجه می‌شوید کد موجود در تابع **main** شکل ۳-۱۴ تقریباً با **doubleStack** در خطوط 30-11 و **intStack** در خطوط 50-32 یکسان است. اینحالت فرصت دیگری برای استفاده از یک الگوی تابع را فراهم می‌آورد. در برنامه شکل ۴-۱۴ الگوی تابع **testStack** تعریف شده (خطوط 38-14) تا همان وظایف **main** در شکل ۳-۱۴ را انجام دهد. یعنی وارد کردن مقادیری به **Stack<T>** و خارج کردن مقادیر از **Stack<T>**. الگوی تابع **testStack** از پارامتر الگوی **T** (مشخص شده در خط 14) برای ارائه نوع داده ذخیره شده در **Stack<T>** استفاده کرده است. الگوی تابع، چهار آرگومان دریافت می‌کند

(خطوط 19-6)، یک مراجعه به یک شی از نوع **Stack<T>**، یک مقدار از نوع **T** که بعنوان اولین مقدار وارد پشته خواهد شد (push)، یک از نوع **T** برای افزایش مقادیر وارد شده به پشته و یک رشته که نشاندهنده نام پشته است که در خروجی از آن استفاده می‌شود. تابع **main** در خطوط 49-40 یک شی از نوع **Stack<double>** بنام **doubleStack** (خط 42) و یک شی از نوع **Stack<int>** بنام **intStack** (خط 43) ایجاد کرده و از این شی‌ها در خطوط 45 و 46 استفاده می‌کند. تابع **testStack** نتیجه کار را به نمایش در می‌آورد. کامپایلر نوع **T** برای **testStack** را از نوع بکار رفته برای نمونه‌سازی اولین آرگومان تابع استنتاج می‌کند. (یعنی نوع بکار رفته در نمونه‌سازی **doubleStasck** یا **intStack**). خروجی شکل ۴-۱۴ دقیقاً با خروجی شکل ۳-۱۴ مطابقت می‌کند.

```
1    // Fig. 14.4: fig14_04.cpp
2    // Stack class template test program. Function main uses a
3    // function template to manipulate objects of type Stack< T >.
4    #include <iostream>
5    using std::cout;
6    using std::endl;
7
8    #include <string>
9    using std::string;
10
11   #include "Stack.h" // Stack class template definition
12
13   // function template to manipulate Stack< T >
14   template< typename T >
15   void testStack(
16      Stack< T > &theStack, // reference to Stack< T >
17      T value, // initial value to push
18      T increment, // increment for subsequent values
19      const string stackName ) // name of the Stack< T > object
20   {
21      cout << "\nPushing elements onto " << stackName << '\n';
22
23      // push element onto Stack
24      while ( theStack.push( value ) )
25      {
26         cout << value << ' ';
27         value += increment;
28      } // end while
29
30      cout << "\nStack is full. Cannot push " << value
31         << "\n\nPopping elements from " << stackName << '\n';
32
33      // pop elements from Stack
34      while ( theStack.pop( value ) )
35         cout << value << ' ';
36
37      cout << "\nStack is empty. Cannot pop" << endl;
38   } // end function template testStack
39
40   int main()
41   {
42      Stack< double > doubleStack( 5 ); // size 5
43      Stack< int > intStack; // default size 10
44
45      testStack( doubleStack, 1.1, 1.1, "doubleStack" );
46      testStack( intStack, 1, 1, "intStack" );
47
48      return 0;
49   } // end main
```

```
Pushing elements onto doubleStack
```

```
1.2 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

شکل ٤-١٤ | ارسال یک شی الگوی Stack به یک الگوی تابع.

٥-١٤ پارامترهای بدون نوع و نوع‌های پیش‌فرض در الگوهای کلاس

در الگوی کلاس **Stack** از بخش ۴-۱۴، فقط از یک نوع پارامتر در سرآیند الگو استفاده شده بود (خط 6). البته امکان استفاده از پارامترهای الگوی بدون نوع یا پارامتر بدون نوع وجود دارد که می‌توانند آرگومان‌ها پیش‌فرض داشته باشند و با آنها همانند ثابت‌ها رفتار کرد. برای مثال، سرآیند الگو می‌تواند برای دریافت پارامتر **elements** از نوع **int** بصورت زیر تغییر یابد:

```
template< typename T, int elements > //nontype parameter elements
```

سپس، اعلانی نظیر زیر انجام داد

```
Stack< double, 100 > mostRecentSalesFigures;
```

که برای نمونه‌سازی (در زمان کامپایل) یک الگوی کلاس تخصصی شده **Stack** با 100 عنصر از مقادیر **double** بنام **mostRecentSalesFigures** بکار گرفته شود. سرآیند کلاس می‌توانست حاوی یک عضو داده **private** (خصوصی) با اعلان آرایه‌ای همانند زیر باشد

```
T stackHolderp[ elements ]; //array to hold Stack contents
```

علاوه بر این، یک پارامتر نوع می‌تواند بصورت یک نوع پیش‌فرض تعیین شود. برای مثال.

```
Template< typename T=string > //defaults to type string
```

مشخص می‌کند که **Stack** حاوی شی‌های رشته بصورت پیش‌فرض است. سپس، اعلانی همانند

```
Stack<> jobDescriptions;
```

می‌تواند برای نمونه‌سازی یک الگوی کلاس تخصصی شده **Stack** حاوی رشته‌ای بنام **jobDescriptions** بکار گرفته شود. این الگوی کلاس تخصصی شده می‌توانست از نوع **Stack<string>** باشد. پارامترها از نوع پیش‌فرض باید سمت راسترین پارامترها در لیست نوع پارامترها در الگو باشند.

در برخی از موارد، امکان استفاده از یک نوع خاص در یک الگوی کلاس وجود ندارد. برای مثال، الگوی **Stack** در شکل ۲-۱۴ مستلزم این است که نوع‌های تعریف شده توسط کاربر که در یک پشته ذخیره خواهند شد، بایستی یک سازنده پیش‌فرض و یک عملگر تخصیص در اختیار داشته باشد. اگر نوع خاص تعریف شده توسط کاربر با الگوی **Stack** ما کار نکند یا نیاز به بهینه‌سازی داشته باشد، می‌توانید یک

الگوی کلاس تخصصی شده صریح برای آن نوع خاص تعریف کنید. اجازه دهید فرض کنیم که می‌خواهیم یک **Stack** تخصصی صریح برای شی‌های **Employee** ایجاد کنیم. برای انجام اینکار، از یک کلاس جدید بنام **Stack<Employee>** بصورت زیر استفاده می‌کنیم:

```
template<>
class Stack< Employee >
{
   //body of class definition
};
```

توجه کنید که **Stack<Employee>** تخصصی شده صریح بطور کامل جایگزین الگوی کلاس **Stack** می‌شود که خاص نوع **Employee** است.

١٤-٦ الگوها و توارث

الگوها و توارث در چند مورد با هم مرتبط هستند:

- یک الگوی کلاس می‌تواند از یک الگوی کلاس تخصصی شده مشتق گردد.
- یک الگوی کلاس می‌تواند از یک کلاس غیرالگو مشتق شود.
- یک الگوی کلاس تخصصی شده می‌تواند از یک الگوی کلاس تخصصی شده، مشتق شود.
- یک کلاس غیرالگو می‌تواند از یک الگوی کلاس تخصصی شده، مشتق شود.

١٤-٧ الگوها و دوستان

مشاهده کردید که توابع و کل کلاس‌ها می‌توانند بعنوان دوستان **(friend)** کلاس‌های غیرالگو باشند. در الگوهای کلاس، رابطه دوستی می‌تواند مابین یک الگوی کلاس و یک تابع سراسری، یک تابع عضو از کلاس دیگری (که می‌تواند یک الگوی کلاس تخصصی شده نیز باشد) یا حتی کل یک کلاس برقرار گردد.

در سرتاسر این بخش، فرض ما بر این است که یک الگوی کلاس برای کلاس بنام **X** با پارامتر نوع **T** بصورت زیر تعریف کرده‌ایم:

```
template< typename T > class X
```

با توجه به این فرض، می‌توان **f1** را بعنوان دوست هر کلاس تخصصی شده از الگوی کلاس برای کلاس **X** ایجاد کرد. برای انجام اینکار، از اعلان رابطه دوستی بصورت زیر استفاده می‌کنیم

```
friend void f1();
```

برای مثال، تابع **f1** دوست **X<double>**، **X<string>** و **X<Employee>** و غیره است. همچنین امکان ایجاد تابع **f2** به نحوی که فقط دوست الگوی کلاس تخصصی شده باشد با همان نوع آرگومان وجود دارد. برای انجام اینکار، از اعلان رابطه دوستی بفرم زیر استفاده می‌شود

```
friend void f2( X< T > & );
```

برای مثال، اگر **T** یک **float** باشد، تابع **f2(x<float> &)** دوستی از الگوی کلاس تخصصی شده **X<float>** است اما دوست الگوی کلاس تخصصی شده **X<string>** نمی باشد.

می‌توانید یک تابع عضو یک کلاس دیگر را بعنوان دوست هر الگوی تخصصی تولید شده از الگوی کلاس اعلان کنید. برای انجام اینکار، اعلان **friend** باید تعیین کننده نام تابع عضو کلاس دیگر با استفاده از نام کلاس و عملگر باینری تفکیک قلمرو، باشد همانند

```
friend void A::f3();
```

این اعلان تابع عضو **f3** از کلاس **A** را دوست هر الگوی کلاس تخصصی شده و ایجاد شده از الگوی قبلی می‌کند. برای مثال تابع **f3** از کلاس **A** دوستی از **X<double>**، **X<string>** و **X<Employe>** و غیره می‌باشد.

همانند یک تابع سراسری، تابع عضو کلاس دیگر می‌تواند فقط دوست یک الگوی کلاس تخصصی شده با همان نوع آرگومان باشد. اعلان دوستی بفرم

```
friend void C< T >::f4( X< T > & );
```

برای نوع خاص **T** همانند **float** تابع عضو بصورت زیر ایجاد می‌شود

```
C< float >::f4( X< float > & )
```

یک تابع دوست فقط برای الگوی کلاس تخصصی شده **X<float>** است.

۸-۱٤ الگوها و اعضای استاتیک

نظرتان در مورد اعضای داده استاتیک چیست؟ بخاطر داشته باشید که با یک کلاس غیرالگو، یک کپی از هر عضو داده استاتیک در میان تمام شی‌های کلاس به اشتراک گذاشته می‌شود، و عضو داده استاتیک بایستی در قلمرو فایل اعلان شود.

هر الگوی کلاس تخصصی شده که از یک الگوی کلاس نمونه‌سازی شده است دارای یک کپی از هر عضو داده استاتیک از الگوی کلاس خواهد بود، تمام شی‌ها از آن الگوها تخصصی شده یک عضو داده استاتیکی را به اشتراک می‌گذارند. علاوه بر این، همانند اعضای داده استاتیک از کلاس‌های غیرالگو، اعضای داده از الگوی کلاس تخصصی شده بایستی تعریف شده و ضرورتاً در قلمرو فایل مقداردهی اولیه شود. هر الگوی تخصصی شده کپی متعلق بخود را از توابع عضو استاتیک الگوی کلاس بدست می‌آورد.