فصل
یازدهم

سربارگذاری عملگر، رشته‌ها و آرایه‌ها

اهداف

- سربارگذاری عملگر چیست و چگونه می‌تواند برنامه‌ها را خواناتر و برنامه‌نویسی را راحتر کند.
- تعریف مجدد (سربارگذاری) عملگرها برای کار با کلاس‌های تعریف شده توسط کاربر.
- تفاوت مابین عملگرهای سربارگذاری شده باینری و غیرباینری.
- تبدیل شی‌ها از یک کلاس به کلاس دیگر.
- زمان سربارگذاری عملگرها.
- ایجاد کلاس‌های PhoneNumber، Array، String و Date برای توصیف سربارگذاری عملگر.

www.yourshop.4kia.ir

- **استفاده از عملگرهای سربارگذاری شده و توابع عضو از کلاس کتابخانه استاندارد String.**
- **استفاده از کلمه کلیدی Explicit.**

رئوس مطالب

۱۱-۱ مقدمه
۱۱-۲ اصول سربارگذاری عملگر
۱۱-۳ محدودیت‌های سربارگذاری عملگر
۱۱-٤ توابع عملگر بعنوان اعضای کلاس در مقابل توابع سراسری
۱۱-٥ سربارگذاری عملگرهای درج و استخراج
۱۱-٦ سربارگذاری عملگرهای غیربایری
۱۱-۷ سربارگذاری عملگرهای بایری
۱۱-۸ مبحث آموزشی: کلاس Array
۱۱-۹ تبدیل مابین نوع‌ها
۱۱-۱۰ مبحث آموزشی: کلاس String
۱۱-۱۱ سربارگذاری ++ و --
۱۱-۱۲ مبحث آموزشی: کلاس Date
۱۱-۱۳ کلاس String از کتابخانه استاندارد
۱۱-۱٤ سازنده‌های Explicit یا صریح

۱۱-۱ مقدمه

در فصل‌های ۱۰-۹ به معرفی اصول اولیه کلاس‌ها در C++ پرداختیم. سرویس‌ها از طریق ارسال پیغام (بشکل فراخوانی توابع عضو) به شی‌ها دریافت می‌شوند. این نحوه فراخوانی تابع بر روی انواع خاصی از کلاس‌ها (همانند کلاس‌های محاسباتی) کار پر زحمتی است. همچنین، برخی از دستکاری‌ها رایج به کمک عملگرها صورت می‌گیرند (همانند ورودی و خروجی). برای انجام چنین اعمالی می‌توانیم از عملگرهای توکار C++ استفاده کنیم.

این فصل نشان می‌دهد که چگونه عملگرهای C++ می‌تواند با شی‌ها کار کنند، عملی که معروف به *سربارگذاری عملگر* است. این روش یک روش سر راست و طبیعی برای بسط C++ با این قابلیت‌های جدید است، اما اینکار بایستی با احتیاط صورت گیرد.

یک مثال از عملگر سربارگذاری شده در C++، عملگر >> است که هم بعنوان عملگر درج و هم بعنوان عملگر بیتی شیفت به‌چپ بکار گرفته می‌شود. به همین ترتیب، عملگر << می‌تواند سربارگذاری

www.yourshop.4kia.ir

گردد و بعنوان عملگر استخراج و هم بعنوان عملگر بیتی شیفت به راست بکار گرفته شود. هر دو این عملگرها در کتابخانه استاندارد C++ سربارگذاری شده‌اند.

اگرچه سربارگذاری عملگر قابلیتی نامتعارف بنظر می‌رسد، اما اکثر برنامه‌نویسان بصورت ضمنی و مرتباً از آن استفاده می‌کنند. برای مثال، خود زبان C++ مبادرت به سربارگذاری عملگر جمع (+) و عملگر تفریق (-) کرده است. این عملگرها براساس متن می‌توانند انجام دهنده محاسبه صحیح، اعشاری و اشاره‌گر باشند. برخی از عملگر به دفعات سربارگذاری می‌شوند، بویژه عملگر تخصیص و برخی از عملگرهای محاسباتی همانند + و -. کاری که عملگرهای سربارگذاری شده می‌توانند انجام دهند، توسط فراخوانی صریح توابع قابل اجرا است، اما نشان‌گذاری عملگر از وضوح بیشتری برخوردار بوده و برای برنامه‌نویسان آشناتر هستند.

برای توصیف نحوه سربارگذاری عملگرها، مبادرت به ایجاد کلاس‌های **PhoneNumber**، **Array**، **String** و **Date**، شامل عملگرهای درج، استخراج، تخصیص، تساوی، رابطه‌ای، شاخص، نفی منطقی، پرانتز و عملگرهای افزاینده خواهیم کرد. فصل با مثالی از کلاس **String** از کتابخانه استاندارد C++ که حاوی تعدادی عملگر سربارگذاری شده خاتمه می‌یابد.

۲-۱۱ اصول سربارگذاری عملگر

برنامه‌نویسی C++ یک فرآیند حساس به نوع و متمرکز بر نوع است. برنامه‌نویسان می‌توانند از نوع‌های بنیادین استفاده کرده و نوع‌های جدیدی تعریف کنند. نوع‌های بنیادین قادر به استفاده از انواع عملگرهای C++ هستند. عملگرهای تدارک دیده شده توسط برنامه‌نویسان با نشانه‌گذاری مختصر در عباراتی بکار می‌روند که دارای شی‌های از نوع‌های بنیادین می‌باشند.

بعلاوه برنامه‌نویسان می‌تواند از عملگرها به همراه نوع‌های تعریف شده از سوی کاربر کار کنند. اگرچه C++ اجازه ایجاد عملگرهای جدید را نمی‌دهد، اما اجازه می‌دهد تا اکثر عملگرهای موجود را به هنگام کار بر روی شی‌ها سربارگذاری کرد که خود قابلیتی توانمند است.

یک عملگر با نوشتن تعریف تابع عضو غیراستاتیک یا تعریف تابع سراسری سربارگذاری می‌شود، بجز اینکه نام تابع همراه با کلمه کلیدی **operator** و بدنبال آن سمبل عملگری که می‌خواهیم سربارگذاری شود، آورده می‌شود. برای مثال، نام تابع **operator+** می‌تواند برای سربارگذاری کردن عملگر جمع (+) بکار گرفته شود. زمانیکه عملگرها بعنوان تابع عضو سربارگذاری می‌شوند، بایستی غیراستاتیک باشند، چرا که باید بر روی یک شی از کلاس فراخوانی شده و بر روی آن شی عمل نمایند.

www.yourshop.4kia.ir

برای استفاده از یک عملگر بر روی شی‌های کلاس، آن عملگر باید سربارگذاری شده باشد، البته با سه استثناء. عملگر تخصیص (=) می‌تواند با هر کلاسی به منظور انجام تخصیص اعضای داده کلاس بکار گرفته شود – هر عضو داده از شی «منبع» به شی «هدف» تخصیص می‌یابد.

بزودی شاهد خواهید بود که چنین تخصیص پیش‌فرضی بر روی کلاس‌هائی با اعضای اشاره‌گر کار خطرناکی است. عملگرهای آدرس (&) و کاما (,) نیز می‌توانند با شی‌های هر کلاسی بکار گرفته شوند، بدون اینکه سربارگذاری شده باشند. عملگر آدرس، مبادرت به بازگرداندن آدرس شی از حافظه می‌کند. عملگر کاما مبادرت به ارزیابی عبارت از سمت چپ کرده، سپس از سمت راست می‌نماید. هر دو این عملگرها می‌توانند سربارگذاری شوند.

سربارگذاری فرآیند بسیار مناسبی برای کلاس‌های محاسباتی (ریاضی) است. انجام اینکار مستلزم سربارگذاری مجموعه‌ای از عملگرها است تا از عملکرد دقیق چنین کلاس‌های که در کارهای واقعی بکار گرفته می‌شوند، مطمئن گردیم. برای مثال، فقط سربارگذاری کردن عملگر جمع در یک کلاس از اعداد مختلط کار غیرعادی است، چرا که در اعداد مختلط از سایر عملگرهای ریاضی استفاده می‌شود.

سربارگذاری کردن عملگر همان عبارات کوتاه و آشنا را برای نوع تعریف شده توسط کاربر را فراهم می‌آورد که C++ با مجموعه‌ای غنی از عملگرهای خود برای نوع‌های بنیادین تدارک دیده است. سربارگذاری کردن عملگر یک فرآیند اتوماتیک نیست و بایستی توابع سربارگذاری عملگر را برای انجام مقاصد خود بنویسید. گاهی اوقات چنین توابعی می‌تواند بصورت توابع عضو، توابع **friend**، گاهی بصورت توابع سراسری و غیردوست ایجاد شوند. در این فصل به چنین مباحثی خواهیم پرداخت.

۱۱-۳ محدودیت‌های سربارگذاری عملگر

اکثر عملگرهای C++ قادر به سربارگذاری شدن هستند. این عملگرها در جدول شکل ۱-۱۱ نشان داده شده‌اند. در جدول شکل ۲-۱۱ عملگرهای که نمی‌توانند سربارگذاری شوند، لیست شده‌اند.

خطای برنامه‌نویسی

مبادرت به سربارگذاری کردن یک عملگر که نمی‌تواند سربارگذاری شود، خطای نحوی است.

عملگرهای قابل سربارگذاری							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=

www.yourshop.4kia.ir

<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

شکل ۱-۱۱ | عملگرهای که می‌توانند سربارگذاری شوند.

عملگرهای غیرقابل سربارگذاری			
.	.*	::	?:

شکل ۲-۱۱ | عملگرهای که نمی‌توانند سربارگذاری شوند.

تقدم، شرکت‌پذیری و تعداد عملوند

تقدم یا اولویت یک عملگر را نمی‌توان با سربارگذاری تغییر داد. چنین عملی می‌تواند شرایط ناخواسته‌ای را سبب شود. با این همه، می‌توان از پرانتزها استفاده کرده و ترتیب ارزیابی عملگرهای سربارگذاری شده در یک عبارت را بدست گرفت.

شرکت‌پذیری یک عملگر (یعنی اعمال عملگر از راست به چپ یا از چپ به راست) را نمی‌توان با سربارگذاری کردن تغییر داد. امکان تغییر در تعداد عملوندهای که یک عملگر می‌تواند برای آنها اثر کند وجود ندارد.

ایجاد عملگرهای جدید

امکان ایجاد عملگر جدید وجود ندارد، فقط می‌توان عملگرهای موجود را سربارگذاری کرد. متاسفانه، چنین رفتاری سبب می‌شود تا برنامه‌نویس قادر به استفاده از نمادهای رایجی همانند عملگر ** که در سایر زبان‌های برنامه‌نویسی برای توان بکار گرفته می‌شود، نباشد [نکته: می‌توانید عملگر ^ را برای انجام توان سربارگذاری نمائید، که در برخی از زبان‌ها کاربرد دارد.]

خطای برنامه‌نویسی

اقدام به ایجاد عملگرهای جدید از طریق سربارگذاری عملگر، یک خطای نحوی است.

عملگرها با نوع‌های بنیادین

مفهوم و معنی نحوه عملکرد یک عملگر بر روی شی‌های از نوع‌های بنیادین را نمی‌توان با سربارگذاری عملگر تغییر داد. برای مثال، برنامه‌نویس نمی‌تواند مفهوم افزودن یا جمع دو مقدار صحیح را تغییر دهد. سربارگذاری عملگر فقط با شی‌های از نوع‌های تعریف شده از سوی کاربر یا ترکیبی از یک شی از نوع تعریف شده از سوی کاربر و یک شی از نوع بنیادین کار می‌کند.

www.yourshop.4kia.ir

عملگرهای وابسته

سربارگذاری یک عملگر تخصیص و یک عملگر جمع به عبارتی همانند عبارت زیر اجازه می‌دهد

```
object2 = object2 + object+1;
```

به این مفهوم نیست که عملگر =+ هم سربارگذاری شده است و عبارتی مانند عبارت زیر داشت

```
object2 += object+1;
```

چنین رفتاری فقط با اعلان صریح سربارگذاری عملگر =+ برای آن کلاس صورت می‌گیرد.

خطای برنامه‌نویسی

فرض اینکه با سربارگذاری یک عملگر همانند +، سایر عملگرهای وابسته همانند =+ یا سربارگذاری == سبب سربارگذاری شدن عملگری مانند =! خواهد شد، خطا است. هر عملگر بایستی بصورت صریح سربارگذاری شود و سربارگذاری ضمنی وجود ندارد.

۱۱-٤ توابع عملگر بعنوان اعضای کلاس در مقابل توابع سراسری

توابع عملگر می‌توانند توابع عضو یا توابع سراسری باشند، غالباً توابع سراسری بدلیل کارایی بصورت دوست (friend) ایجاد می‌شوند. توابع عضو از اشاره‌گر **this** بصورت ضمنی استفاده می‌کنند تا یکی از آرگومان‌های شی کلاس را بدست آورند (عملوند سمت چپ در عملگرهای باینری) آرگومان‌ها برای هر دو عملوند در یک عملگر باینری بایستی بصورت صریح در فراخوانی یک تابع سراسری لیست شده باشند.

عملگرهای که باید بعنوان توابع عضو سربارگذاری شوند

به هنگام سربارگذاری **()**، **[]**، <- یا هر عملگر تخصیصی، عملگر سربارگذاری کننده تابع باید بصورت یک عضو کلاس اعلان شود. برای سایر عملگرها توابع سربارگذاری توابع می‌توانند اعضای کلاس یا توابع سراسری باشند.

عملگرها بعنوان توابع عضو و توابع سراسری

خواه یک تابع عملگر بصورت یک تابع عضو یا یک تابع سراسری پیاده‌سازی شده باشد، عملگر هنوز هم به همان روش در عبارات بکار گرفته می‌شود. بنابر این کدام روش پیاده‌سازی بهتر است؟

زمانیکه یک تابع عملگر بصورت یک تابع عضو پیاده‌سازی می‌شود، سمت چپ‌ترین عملوند بایستی یک شی (با یک مراجعه به یک شی) از کلاس عملگر باشد. اگر عملوند سمت چپ باید شی از یک کلاس متفاوت یا یک نوع بنیادین باشد، این تابع عملگر بایستی بصورت یک تابع سراسری پیاده‌سازی

www.yourshop.4kia.ir

شود. یک تابع عملگر سراسری می‌تواند بصورت یک **friend** از یک کلاس ایجاد شود اگر آن تابع بصورت مستقیم به اعضای **private** یا **protected** آن کلاس دسترسی دارد.

توابع عضو عملگر از یک کلاس خاص فقط در صورتیکه عملوند سمت چپ یک عملگر باینری، یک شی از آن کلاس باشد، یا زمانیکه عملوند منفرد از یک عملگر غیرباینری، یک شی از آن کلاس باشد، توسط کامپایلر فراخوانی خواهد شد (بصورت ضمنی).

چرا سربارگذاری عملگرهای درج و استخراج بصورت توابع سراسری سربارگذاری می‌شوند

عملگر درج جریان (<<) سربارگذاری شده در عبارتی که عملوند سمت چپ آن دارای نوع **ostream &** بصورت **cout<<classObject** است، بکار گرفته می‌شود. برای استفاده از عملگر به این روش که در آن عملوند از سمت راست، یک شی از یک کلاس تعریف شده از سوی کاربر است، بایستی بصورت یک تابع سراسری سربارگذاری شود. برای یک تابع عضو، عملگر << مجبور است تا بصورت عضوی از کلاس **ostream** باشد. اینکار برای کلاس‌های تعریف شده توسط کاربر امکان‌پذیر نیست، از آنجا که اجازه نداریم تا کلاس‌های کتابخانه استاندارد C++ را تغییر دهیم. به همین ترتیب از عملگر استخراج جریان (<<) سربارگذاری شده در عبارتی که عملوند سمت چپ دارای نوع **istream &**، بصورت **cin>>classObject** است و عملوند سمت راست یک شی از یک کلاس تعریف شد توسط کاربر است، بایستی بصورت یک تابع سراسری باشد. همچنین امکان دارد هر یک از این توابع عملگر سربارگذاری شده نیاز به دسترسی به اعضای داده **private** داشته باشند، از اینرو چنین توابعی می‌توانند بصورت توابع **friend** کلاس ایجاد شوند تا کارایی افزایش یابد.

جابجایی عملگرها

یکی دیگر از دلایل انتخاب توابع سراسری برای سربارگذاری یک عملگر امکان جابجا کردن عملگر است. برای مثال، فرض کنید یک شی **number** از نوع **long int** داریم و یک شی بنام **bigInteger1** از کلاس **HugeInteger** (کلاسی که تعداد ارقام بیش از ظرفیت سایز word در کامپیوتر است). عملگر جمع (+) بطور موقت یک شی **HugeInteger** بعنوان مجموع یک **HugeInteger** و یک **long int** (در عبارتی بصورت **bigInteger1 + number**)، یا بعنوان مجموع یک **long int** و یک **HugeInteger** (در عبارتی بصورت **number + bigIneger1**) تولید می‌کند. از اینرو، نیازمند عملگر جمعی هستیم که جابجاپذیر باشد. مشکل اینجاست که شی کلاس باید در سمت چپ عملگر جمع قرار گیرد، اگر آن عملگر بعنوان یک تابع عضو سربارگذاری شده باشد. از اینرو، اقدام به سربارگذاری عملگر بفرم یک تابع

www.yourshop.4kia.ir

سراسری می‌کنیم تا به **HugeInteger** اجازه دهد تا در سمت راست جمع قرار داده شود. تابع **operator+** که با **HugeInteger** در سمت چپ کار می کند، هنوز هم می‌تواند یک تابع عضو باشد.

۱۱-۵ سربارگذاری عملگرهای درج و استخراج

زبان C++ قادر است تا با استفاده از عملگرهای درج (<<) و استخراج و استخراج (<<) مبادرت به ورود و خروج نوع‌های بنیادین کند. کتابخانه‌های کلاس تدارک دیده شده همراه کامپایلرهای C++ مبادرت به سربارگذاری این عملگرها برای پردازش هر نوع بنیادین می‌کنند که شامل اشاره‌گرها و رشته‌های Char * هم می‌شود. همچنین می‌توان برای انجام عملیات ورودی و خروجی بر روی نوع‌های تعریف شده توسط کاربر مبادرت به سربارگذاری عملگرهای درج و استخراج کرد. برنامه موجود در شکل‌های ۳-۱۱ الی ۵-۱۱ به توصیف نحوه سربارگذاری این عملگرها برای رسیدگی به داده تعریف شده از سوی کاربر که یک شماره تلفن است و در کلاسی بنام **PhoneNumber** قرارداد، می‌پردازد. فرض برنامه بر این است که شماره تلفن‌ها بدرستی وارد شده‌اند.

```
1  // Fig. 11.3: PhoneNumber.h
2  // PhoneNumber class definition
3  #ifndef PHONENUMBER_H
4  #define PHONENUMBER_H
5
6  #include <iostream>
7  using std::ostream;
8  using std::istream;
9
10 #include <string>
11 using std::string;
12
13 class PhoneNumber
14 {
15    friend ostream &operator<<( ostream &, const PhoneNumber & );
16    friend istream &operator>>( istream &, PhoneNumber & );
17 private:
18    string areaCode; // 3-digit area code
19    string exchange; // 3-digit exchange
20    string line; // 4-digit line
21 }; // end class PhoneNumber
22
23 #endif
```

شکل ۳-۱۱ | کلاس PhoneNumber با عملگرهای سربارگذاری شده درج و استخراج بعنوان توابع frirend.

```
1  // Fig. 11.4: PhoneNumber.cpp
2  // Overloaded stream insertion and stream extraction operators
3  // for class PhoneNumber.
4  #include <iomanip>
5  using std::setw;
6
7  #include "PhoneNumber.h"
8
9  // overloaded stream insertion operator; cannot be
10 // a member function if we would like to invoke it with
11 // cout << somePhoneNumber;
12 ostream &operator<<( ostream &output, const PhoneNumber &number )
13 {
```

www.yourshop.4kia.ir

```
14    output << "(" << number.areaCode << ") "
15       << number.exchange << "-" << number.line;
16    return output; // enables cout << a << b << c;
17 } // end function operator<<
18
19 // overloaded stream extraction operator; cannot be
20 // a member function if we would like to invoke it with
21 // cin >> somePhoneNumber;
22 istream &operator>>( istream &input, PhoneNumber &number )
23 {
24    input.ignore(); // skip (
25    input >> setw( 3 ) >> number.areaCode; // input area code
26    input.ignore( 2 ); // skip ) and space
27    input >> setw( 3 ) >> number.exchange; // input exchange
28    input.ignore(); // skip dash (-)
29    input >> setw( 4 ) >> number.line; // input line
30    return input; // enables cin >> a >> b >> c;
31 } // end function operator>>
```

شکل ٤-١١ | سربارگذاری عملگرهای درج و استخراج برای کلاس PhoneNumber.

```
1  // Fig. 11.5: fig11_05.cpp
2  // Demonstrating class PhoneNumber's overloaded stream insertion
3  // and stream extraction operators.
4  #include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include "PhoneNumber.h"
10
11 int main()
12 {
13    PhoneNumber phone; // create object phone
14
15    cout << "Enter phone number in the form (123) 456-7890:" << endl;
16
17    // cin >> phone invokes operator>> by implicitly issuing
18    // the global function call operator>>( cin, phone )
19    cin >> phone;
20
21    cout << "The phone number entered was: ";
22
23    // cout << phone invokes operator<< by implicitly issuing
24    // the global function call operator<<( cout, phone )
25    cout << phone << endl;
26    return 0;
27 } // end main
```

```
Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number enterd was: (800) 555-1212
```

شکل ٥-١١ | سربارگذاری عملگرهای درج و استخراج.

تابع عملگر استخراج **operator>>** (شکل ۴-۱۱، خطوط 31-22) مبادرت به دریافت مراجعه **istream** به **input** و **PhoneNumber** به **num** بعنوان آرگومان کرده و یک مراجعه **istream** برگشت می‌دهد. تابع عملگر **operator>>** شماره تلفن را بفرم زیر دریافت می‌کند.

```
(800) 555-1212
```

و در کلاس **PhoneNumber** قرار می‌دهد. زمانیکه کامپایلر با عبارت زیر مواجه می‌شود (خط 19 از شکل ۵-۱۱)

```
cin >> phone
```

www.yourshop.4kia.ir

کامپایلر مبادرت به فراخوانی تابع سراسری زیر می‌کند

```
operator>>(cin,phone);
```

زمانیکه این فراخوانی اجرا می‌شود، پارامتر مراجعه **input** (شکل ۴-۱۱، خط 22) تبدیل به یک عبارت مستعار برای **cin** شده و پارامتر مراجعه **number** تبدیل به یک مستعار برای **phone** می‌گردد. تابع عملگر، رشته‌ها را بصورت سه بخشی از شماره تلفن خوانده و در اعضای **areaCode** (خط 25)، **exchange** (خط 27) و **line** (خط 29) از شی **PhoneNumber** وارد می‌کند که توسط پارامتر **number** مورد مراجعه قرار دارد. دستکاری کننده استریم **setw** محدود کننده تعداد کاراکترهای خوانده شده به هر آرایه کاراکتری است. زمانیکه با **cin** و رشته‌ها بکار گرفته می‌شود، setw تعداد کاراکترهای خوانده شده را محدود به تعداد کاراکترهای تعیین شده توسط آرگومان خود می کند (مثلاً **setw(3)** اجازه خواندن سه کاراکتر را می‌دهد). پرانتزها، فاصله‌ها و کاراکترهای خط تیره با فراخوانی تابع عضو **istream** نادیده گرفته می‌شوند (شکل ۴-۱۱، خطوط 26,24 و 28)، که تعداد مشخصی از کاراکترها در استریم ورودی به کنار گذاشته می‌شوند. تابع **operator>>** مراجعه ورودی را برگشت می‌دهد (یعنی **cin**). اینکار به عملیات ورودی بر روی شی‌های **PhoneNumber** امکان می‌دهد تا متصل با عملیات ورودی بر روی دیگر شی‌های **PhoneNumber** یا شی‌های از سایر نوع‌های داده، به پیش رود. برای مثال، برنامه می‌تواند دو شی **PhoneNumber** را در یک عبارت وارد کند، همانند

```
cin >> phone1 >> phone2;
```

ابتدا عبارت **cin>>phone1** با فراخوانی تابع سراسری

```
operator>>(cin, phone1);
```

اجرا می‌شود. سپس این فراخوانی یک مراجعه به **cin** بعنوان مقداری از **cin>>phone1** برگشت می‌دهد، از اینرو مابقی بخشی باقیمانده عبارت بصورت **cin>>phone2** ارزیابی می‌گردد. اینکار با فراخوانی تابع سراسری صورت می‌گیرد.

```
operator>>(cin, phone2);
```

تابع عملگر درج (شکل ۴-۱۱، خطوط 17-12) یک مراجعه **ostream** (خروجی– output) و یک مراجعه ثابت **PhoneNumber** بعنوان آرگومان دریافت و یک مراجعه **ostream** برگشت می‌دهد. تابع **opereator >>** شی‌های از نوع **PhoneNumber** برگشت می‌دهد. زمانیکه کامپایلر به عبارت زیر می‌رسد (خط 25 از شکل ۵-۱۱)

```
cout << phone
```

کامپایلر یک فراخوانی تابع سراسری را بوجود می‌آورد

www.yourshop.4kia.ir

```
operator<<(cout, phone);
```

تابع **operator<<** بخش‌های یک شماره تلفن را بصورت رشته‌های به نمایش در می‌آورد، چرا که آنها بعنوان شی‌های رشته ذخیره شده بودند.

توجه کنید که توابع **operator<<** و **operator<<** در **phoneNumber** بصورت سراسری، توابع دوست (*friend*) اعلان شده‌اند (شکل ۳-۱۱، خطوط 16-15). اینها توابع سراسری هستند چرا که شی از کلاس **PhoneNumber** در هر حالت بصورت عملوند سمت راست عملگر ظاهر می‌شود. بخاطر دارید که، توابع عملگر سربارگذاری شده برای عملگرهای باینری می‌تواند توابع عضو باشند. در صورتیکه فقط عملوند سمت چپ یک شی از کلاسی باشد که در آن تابع عضو باشد. اگر عملگرهای ورودی و خروجی سربارگذاری شده نیاز به دسترسی مستقیم به اعضای کلاس غیر *public* داشته باشند، می‌تواند بصورت **friend** اعلان شوند. اینکار می‌تواند به دلایل کارایی باشد یا اینکه کلاس حاوی توابع *get* مقتضی نباشد. همچنین دقت کنید که مراجعه **PhoneNumber** در لیست پارامتری **operator<<** (شکل ۴-۱۱، خط 12) ثابت (const) است، چرا که **PhoneNumber** فقط خروجی است و مراجعه **PhoneNumber** در لیست پارامتری **operator<<** (خط 22) یک مقدار غیرثابت است، به این دلیل که شی **phoneNumber** بایستی برای ذخیره‌سازی شماره تلفن در یک شی، تغییرپذیر باشد.

٦-۱۱ سربارگذاری عملگرهای غیربایری

یک عملگر غیربایری در ارتباط با یک کلاس می‌تواند بصورت یک تابع غیراستاتیک بدون آرگومان یا بصورت یک تابع سراسری با یک آرگومان، که آن آرگومان بایستی یک شی از کلاس یا مراجعه‌ای به شی از آن کلاس باشد، سربارگذاری شود. توابع عضو که عملگرهای سربارگذاری شده را پیاده‌سازی می‌کنند بایستی بصورت غیراستاتیک باشند، از اینروست که می‌توانند به داده غیراستاتیک در هر شی از کلاس دسترسی پیدا کنند. بخاطر داشته باشید که توابع عضو استاتیک فقط می‌توانند به اعضای داده استاتیک کلاس دسترسی پیدا کنند.

در ادامه این فصل مبادرت به سربارگذاری عملگر غیربایری **!** برای تست این مطلب خواهیم کرد که آیا یک شی که از کلاس **String** ایجاد می‌کنیم (بخش ۱۰-۱۱) تهی بوده و نتیجه **bool** برگشت می‌دهد یا خیر. به عبارت **s!** توجه کنید که در آن **s** یک شی از کلاس **String** است. زمانیکه یک عملگر غیربایری همانند **!** بصورت یک تابع عضو سربارگذاری می‌شود (بدون آرگومان) و کامپایلر عبارت **s!** را مشاهده می‌کند، مبادرت به فراخوانی **s.operator!()** می‌نماید. عملوند **s** شی از کلاس **String** است. تابع در تعریف کلاس بصورت زیر اعلان شده است:

```
class String
```

www.yourshop.4kia.ir

```
{
public:
        bool operator!() const;
}; //end class String
```

یک عملگر غیربایتری همانند **!** می‌توانند به دو روش به همراه یک آرگومان بصورت یک تابع سراسری سربارگذاری گردد، خواه با یک آرگومان که یک شی است (اینکار مستلزم یک کپی از شی بوده، از اینرو اثرات جانبی تاثیری بر شی واقعی نخواهند داشت) یا با آرگومانی که یک مراجعه به یک شی است (کپی از شی اصلی وجود ندارد، از اینرو تمام تاثیرات جانبی این تابع بر روی شی اصلی یا واقعی تاثیرگذار خواهند بود). اگر **s** یک شی از کلاس **String** باشد (یا یک مراجعه به یک شی از کلاس **String**)، پس با **s!** همانند فراخوانی **operator!(s)** رفتار خواهد شد که فراخوانی آن بصورت زیر اعلان شده است:

```
bool operator!(const String &);
```

۱۱-۷ سربارگذاری عملگرهای باینری

عملگر باینری می‌تواند بصورت یک تابع عضو غیراستاتیک با یک آرگومان یا بصورت یک تابع سراسری با دو آرگومان (یکی از این آرگومان‌ها باید یک شی کلاس یا یک مراجعه به شی کلاس باشد) سربارگذاری گردد.

در ادامه این فصل، مبادرت به سربارگذاری > برای مقایسه دو شی رشته‌ای خواهیم کرد. در زمان سربارگذاری عملگر باینری > بصورت یک تابع عضو غیراستاتیک از کلاس **String** با یک آرگومان، اگر **y** و **z** شی‌های از کلاس **String** باشند، پس با **y>z** بصورت **y.operator(z)** رفتار خواهد شد، که فراخوانی تابع عضو > **operator** بصورت زیر اعلان شده است.

```
class String
{
public:
        bool operator<(const String &) const;
}; //end class String
```

اگر عملگر باینری > بصورت یک تابع سراسری سربارگذاری شود، بایستی دو آرگومان دریافت کند. اگر **y** و **z** شی‌های از کلاس **String** باشند یا مراجعه‌ای به شی‌های از کلاس **String**، پس با **y<z** بصورت **operator(y,z)** رفتار خواهد شد، اعلان تابع سراسری < **operator** بصورت زیر است:

```
bool operator<(const String &, const Strig &);
```

۱۱-۸ مبحث آموزشی: کلاس Array

آرایه‌های مبتنی بر اشاره‌گر چندین مشکل دارند. برای مثال، برنامه می‌تواند به راحتی از مرزهای آرایه خارج شود، چرا که C++ تستی بر روی مرزهای آرایه انجام نمی‌دهد (خود برنامه‌نویس می‌تواند اینکار را انجام دهد). آرایه‌های با سایز n بایستی عناصری به تعداد *0, ..., n-1* داشته باشند، تغییر محدودهٔ شاخص

www.yourshop.4kia.ir

امکان‌پذیر نمی‌باشد. کل یک آرایه غیرکاراکتری را نمی‌توان به یکباره از ورودی دریافت یا در خروجی قرار داد، هر عنصر آرایه بایستی بصورت جداگانه خوانده یا نوشته شود. دو آرایه را نمی‌توان با عملگرهای تساوی یا رابطه‌ای و آن هم بصورت معنی‌دار مقایسه کرد (چرا که اسامی آرایه‌ها اشاره‌گرهایی به شروع آرایه در حافظه هستند و البته دو آرایه همیشه در دو مکان متفاوت حافظه خواهند بود). زمانیکه یک آرایه به یک تابع چندمنظوره طراحی شده برای کار با آرایه‌ها با هر سایزی ارسال می‌شود، سایز آرایه بایستی بعنوان یک آرگومان اضافی به تابع ارسال گردد. یک آرایه را نمی‌توان به آرایه دیگری با عملگر تخصیص، انتساب داد (چرا که اسامی آرایه‌ها اشاره‌گرهای ثابت (*const*) بوده و از اشاره‌گر ثابت نمی‌توان در سمت چپ یک عملگر تخصیص استفاده کرد). بنظر می‌رسد که انجام چنین کارهای با آرایه‌ها نبایستی کار غیرعادی باشد، اما آرایه‌های مبتنی بر اشاره‌گر دارای چنین قابلیت‌های نیستند. با این همه C++ مفهومی برای پیاده‌سازی آرایه‌ها با چنین قابلیت‌های از طریق استفاده از کلاس‌ها و سربارگذاری عملگر تدارک دیده است.

در این مثال، یک کلاس آرایه قدرتمند ایجاد خواهیم کرد که قادر به انجام تست بر روی مرزهای آرایه است. کلاس به یک شی آرایه اجازه می‌دهد تا با استفاده از عملگر تخصیص به یک آرایه دیگر انتساب یابد. شی‌ها از کلاس **Array** از سایز خود مطلع بوده و از اینرو نیازی نیست تا سایز آرایه بعنوان یک آرگومان مجزا به هنگام ارسال آرایه به یک تابع همراه شود. کل آرایه را می‌توان با استفاده از عملگرهای درج و استخراج از ورودی دریافت و در خروجی قرار داد. مقایسه آرایه را می‌توانیم با عملگرهای تساوی== و =! انجام دهیم.

این مثال درک شما را از انتزاعی کردن داده افزایش خواهد داد. امکان دارد که بخواهید قابلیت‌های دیگری به این کلاس آرایه اضافه کنید. برنامه موجود در شکل‌های ۶-۱۱ الی ۸-۱۱ به توصیف کلاس **Array** و سربارگذاری عملگرهای آن می‌پردازد. ابتدا به سراغ **main** می‌رویم (شکل ۸-۱۱). سپس به تعریف کلاس (شکل ۶-۱۱) و هر یک از تعاریف توابع عضو و توابع **friend** کلاس می‌پردازیم (شکل ۷-۱۱).

```
1  // Fig. 11.6: Array.h
2  // Array class for storing arrays of integers.
3  #ifndef ARRAY_H
4  #define ARRAY_H
5
6  #include <iostream>
7  using std::ostream;
8  using std::istream;
9
10 class Array
11 {
12    friend ostream &operator<<( ostream &, const Array & );
13    friend istream &operator>>( istream &, Array & );
14 public:
15    Array( int = 10 ); // default constructor
```

www.yourshop.4kia.ir

```
   Array( const Array & ); // copy constructor
   ~Array(); // destructor
   int getSize() const; // return size

   const Array &operator=( const Array & ); // assignment operator
   bool operator==( const Array & ) const; // equality operator

   // inequality operator; returns opposite of == operator
   bool operator!=( const Array &right ) const
   {
      return ! ( *this == right ); // invokes Array::operator==
   } // end function operator!=

   // subscript operator for non-const objects returns modifiable lvalue
   int &operator[]( int );

   // subscript operator for const objects returns rvalue
   int operator[]( int ) const;
private:
   int size; // pointer-based array size
   int *ptr; // pointer to first element of pointer-based array
}; // end class Array

#endif
```

شکل ٦-١١ | تعریف کلاس Array با عملگرهای سربارگذاری شده.

```
// Fig 11.7: Array.cpp
// Member-function definitions for class Array
#include <iostream>
using std::cerr;
using std::cout;
using std::cin;
using std::endl;

#include <iomanip>
using std::setw;

#include <cstdlib> // exit function prototype
using std::exit;

#include "Array.h" // Array class definition

// default constructor for class Array (default size 10)
Array::Array( int arraySize )
{
   size = ( arraySize > 0 ? arraySize : 10 ); // validate arraySize
   ptr = new int[ size ]; // create space for pointer-based array

   for ( int i = 0; i < size; i++ )
      ptr[ i ] = 0; // set pointer-based array element
} // end Array default constructor

// copy constructor for class Array;
// must receive a reference to prevent infinite recursion
Array::Array( const Array &arrayToCopy )
   : size( arrayToCopy.size )
{
   ptr = new int[ size ]; // create space for pointer-based array

   for ( int i = 0; i < size; i++ )
      ptr[ i ] = arrayToCopy.ptr[ i ]; // copy into object
} // end Array copy constructor

// destructor for class Array
Array::~Array()
{
   delete [] ptr; // release pointer-based array space
} // end destructor

// return number of elements of Array
```

www.yourshop.4kia.ir

```
int Array::getSize() const
{
   return size; // number of elements in Array
} // end function getSize

// overloaded assignment operator;
// const return avoids: ( a1 = a2 ) = a3
const Array &Array::operator=( const Array &right )
{
   if ( &right != this ) // avoid self-assignment
   {
      // for Arrays of different sizes, deallocate original
      // left-side array, then allocate new left-side array
      if ( size != right.size )
      {
         delete [] ptr; // release space
         size = right.size; // resize this object
         ptr = new int[ size ]; // create space for array copy
      } // end inner if

      for ( int i = 0; i < size; i++ )
         ptr[ i ] = right.ptr[ i ]; // copy array into object
   } // end outer if

   return *this; // enables x = y = z, for example
} // end function operator=

// determine if two Arrays are equal and
// return true, otherwise return false
bool Array::operator==( const Array &right ) const
{
   if ( size != right.size )
      return false; // arrays of different number of elements

   for ( int i = 0; i < size; i++ )
      if ( ptr[ i ] != right.ptr[ i ] )
         return false; // Array contents are not equal

   return true; // Arrays are equal
} // end function operator==

// overloaded subscript operator for non-const Arrays;
// reference return creates a modifiable lvalue
int &Array::operator[]( int subscript )
{
   // check for subscript out-of-range error
   if ( subscript < 0 || subscript >= size )
   {
      cerr << "\nError: Subscript " << subscript
         << " out of range" << endl;
      exit( 1 ); // terminate program; subscript out of range
   } // end if

   return ptr[ subscript ]; // reference return
} // end function operator[]

// overloaded subscript operator for const Arrays
// const reference return creates an rvalue
int Array::operator[]( int subscript ) const
{
   // check for subscript out-of-range error
   if ( subscript < 0 || subscript >= size )
   {
      cerr << "\nError: Subscript " << subscript
         << " out of range" << endl;
      exit( 1 ); // terminate program; subscript out of range
   } // end if

   return ptr[ subscript ]; // returns copy of this element
} // end function operator[]
```

www.yourshop.4kia.ir

```
115
116 // overloaded input operator for class Array;
117 // inputs values for entire Array
118 istream &operator>>( istream &input, Array &a )
119 {
120    for ( int i = 0; i < a.size; i++ )
121       input >> a.ptr[ i ];
122
123    return input; // enables cin >> x >> y;
124 } // end function
125
126 // overloaded output operator for class Array
127 ostream &operator<<( ostream &output, const Array &a )
128 {
129    int i;
130
131    // output private ptr-based array
132    for ( i = 0; i < a.size; i++ )
133    {
134       output << setw( 12 ) << a.ptr[ i ];
135
136       if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
137          output << endl;
138    } // end for
139
140    if ( i % 4 != 0 ) // end last line of output
141       output << endl;
142
143    return output; // enables cout << x << y;
144 } // end function operator<<
```

شکل ۷-۱۱ | تعاریف عضو و friend کلاس Array.

```
1  // Fig. 11.8: fig11_08.cpp
2  // Array class test program.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  #include "Array.h"
9
10 int main()
11 {
12    Array integers1( 7 ); // seven-element Array
13    Array integers2; // 10-element Array by default
14
15    // print integers1 size and contents
16    cout << "Size of Array integers1 is "
17       << integers1.getSize()
18       << "\nArray after initialization:\n" << integers1;
19
20    // print integers2 size and contents
21    cout << "\nSize of Array integers2 is "
22       << integers2.getSize()
23       << "\nArray after initialization:\n" << integers2;
24
25    // input and print integers1 and integers2
26    cout << "\nEnter 17 integers:" << endl;
27    cin >> integers1 >> integers2;
28
29    cout << "\nAfter input, the Arrays contain:\n"
30       << "integers1:\n" << integers1
31       << "integers2:\n" << integers2;
32
33    // use overloaded inequality (!=) operator
34    cout << "\nEvaluating: integers1 != integers2" << endl;
35
36    if ( integers1 != integers2 )
37       cout << "integers1 and integers2 are not equal" << endl;
38
```

www.yourshop.4kia.ir

```
39    // create Array integers3 using integers1 as an
40    // initializer; print size and contents
41    Array integers3( integers1 ); // invokes copy constructor
42
43    cout << "\nSize of Array integers3 is "
44       << integers3.getSize()
45       << "\nArray after initialization:\n" << integers3;
46
47    // use overloaded assignment (=) operator
48    cout << "\nAssigning integers2 to integers1:" << endl;
49    integers1 = integers2; // note target Array is smaller
50
51    cout << "integers1:\n" << integers1
52       << "integers2:\n" << integers2;
53
54    // use overloaded equality (==) operator
55    cout << "\nEvaluating: integers1 == integers2" << endl;
56
57    if ( integers1 == integers2 )
58       cout << "integers1 and integers2 are equal" << endl;
59
60    // use overloaded subscript operator to create rvalue
61    cout << "\nintegers1[5] is " << integers1[ 5 ];
62
63    // use overloaded subscript operator to create lvalue
64    cout << "\n\nAssigning 1000 to integers1[5]" << endl;
65    integers1[ 5 ] = 1000;
66    cout << "integers1:\n" << integers1;
67
68    // attempt to use out-of-range subscript
69    cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
70    integers1[ 15 ] = 1000; // ERROR: out of range
71    return 0;
72 } // end main
```

```
Size of Array integers1 is 7
Array after initialization:
           0           0           0           0
           0           0           0
Size of Array integers2 is 10
Array after initialization:
           0           0           0           0
           0           0           0           0
           0           0
Enter 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the Arrays contain:
integers1:
           1           2           3           4
           5           6           7

integers2:
           8           9           10           11
          12          13           14           15
          16          17
Evaluating: integers1 != integers2
integers1 and integers2 are not equal

Size of Array integers3 is 7
Array after initialization:
           1           2           3           4
           5           6           7
Assigning integers2 to integers1:
integers1:
           8           9           10           11
          12          13           14           15
          16          17
integers2:
           8           9           10           11
```

www.yourshop.4kia.ir

```
          12          13          14          15
          16          17
Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13

Assigning 1000 to integers1[5]
integers1:
           8           9          10          11
          12        1000          14          15
          16          17
Attempt to assign 1000 to integers1[15]
Error: Subscript 15 out range
```

شکل ۸-۱۱ | برنامه تست کلاس Array.

ایجاد آرایه‌ها، نمایش سایز و محتویات آنها

برنامه با نمونه‌سازی دو شی از کلاس **Array** بنام‌های **integer1** (شکل ۸-۱۱، خط 12) با هفت عنصر، **integer2** (شکل ۸-۱۱، خط 13) با سایز پیش‌فرض **Array** یعنی 10 عنصر (مشخص شده توسط سازنده پیش‌فرض **Array** در شکل ۶-۱۱، خط 15) شروع می‌شود.

خطوط 18-16 از تابع عضو **getSize** برای تعیین سایز **integer1** استفاده کرده و محتویات **integer1** با استفاده از عملگر درج سربارگذاری شده **Array** در خروجی قرار داده می‌شود. خروجی نمونه این برنامه تایید می‌کند که عناصر **Array** بدرستی توسط سازنده با صفر مقداردهی اولیه شده‌اند. سپس خطوط 23-21 سایز آرایه **integer2** و محتویات آنرا توسط عملگر درج سربارگذاری شده، چاپ می‌کنند.

استفاده از عملگر درج سربارگذاری شده برای پر کردن آرایه

خط 26 به کاربر اعلان می‌کند تا 17 مقدار صحیح وارد سازد. خط 27 از عملگر استخراج سربارگذاری شده **Array** برای خواندن این مقادیر به هر دو آرایه استفاده کرده است. هفت مقدار اول در **integer1** و ده مقدار باقی مانده در **integer2** ذخیره می‌شوند. خطوط 31-29 دو آرایه را توسط عملگر درج سربارگذاری شده **Array** در خروجی قرار می‌دهند تا نشان دهند که عملیات ورودی بدرستی صورت گرفته است.

استفاده از عملگر نابرابری سربارگذاری شده

خط 36 مبادرت به تست عملگر نابرابری سربارگذاری شده با ارزیابی شرط

```
integers1 != integers2
```

می‌کند. خروجی برنامه نشان می‌دهد که آرایه‌ها به راستی برابر نیستند.

www.yourshop.4kia.ir

مقداردهی اولیه آرایه جدید با کپی از محتویات یک آرایه موجود

خط 41 مبادرت به نمونه‌سازی آرایه سومی بنام **integers3** کرده و آنرا با کپی از آرایه **integers1** مقداردهی اولیه می‌نماید. با اینکار سازنده کپی کننده آرایه فعال شده و عناصر آرایه **integers1** را به **integers3** کپی می‌نماید. بزودی در مورد جزئیات سازنده کپی‌کننده صحبت خواهیم کرد. دقت کنید که سازنده کپی‌کننده می‌تواند با نوشتن خط 41 بصورت زیر هم فعال شود:

```
Array integers3 = integers1;
```

نماد تساوی در عبارت فوق عملگر تخصیص نمی‌باشد. زمانیکه یک نماد تساوی در اعلان یک شی ظاهر می‌شود، مبادرت به فراخوانی سازنده برای آن شی می‌کند. در اینحالت فقط یک آرگومان می‌تواند به سازنده ارسال شود.

خطوط 45-43 سایز **integers3** و محتویات آنرا توسط عملگر درج سربارگذاری شده **Array** چاپ می‌کنند تا نشان دهند که عناصر آرایه بدرستی توسط سازنده کپی کننده مقداردهی شده است.

استفاده از عملگر تخصیص سربارگذاری شده

خط 49 مبادرت به تست عملگر تخصیص سربارگذاری شده (=) با تخصیص دادن **integers2** به **integers1** می‌کند. خطوط 52-51 هر دو آرایه را برای نشان دادن اینکه عملیات تخصیص با موفقیت صورت گرفته چاپ می‌کنند. دقت کنید که **integers1** در ابتدای کار هفت مقدار صحیح در خود نگهداری کرده بود و برای نگهداری ده عنصر **integers2** تغییر سایز داده است. همانطوری که مشاهده می‌کنید، عملگر تخصیص سربارگذاری شده این عملیات تغییر سایز را به روشی انجام می‌دهد که از دید کد سرویس گیرنده پنهان است.

استفاده از عملگر برابری سربارگذاری شده

خط 57 از عملگر برابری سربارگذاری شده (==) برای تایید اینکه آرایه‌های **integers1** و **integers2** به راستی پس از تخصیص با هم برابر هستند، استفاده کرده است.

استفاده از عملگر شاخص سربارگذاری شده

خط 61 از عملگر شاخص سربارگذاری شده برای اشاره یا مراجعه به **integers1[5]** استفاده کرده است که عنصری در محدودهٔ **integers1** است. نام شاخص بعنوان *rvalue* (مقدار سمت راست) برای چاپ مقدار ذخیره شده در **integers1[5]** بکار گرفته شده است. خط 65 از **integers1[5]** بعنوان یک *lvalue* (مقدار سمت چپ) تغییرپذیر در سمت چپ یک عبارت تخصیصی به منظور تخصیص یک مقدار جدید، 1000 به عنصر 5 از **integers1** استفاده کرده است. شاهد خواهید بود که **operator[]** یک مراجعه برای

www.yourshop.4kia.ir

استفاده بعنوان *lvalue* اصلاح‌پذیر پس از عملگر برگشت می‌دهد که نشان دهد که 5 یک شاخص معتبر برای **integers1** است.

خط 70 مبادرت به تخصیص مقدار 1000 به **integers1[15]** می‌کند، که عنصری خارج از محدودهٔ یا مرز آرایه می‌باشد. در این مثال، **[]operator** تعیین می‌کند که شاخص خارج از محدودهٔ بوده، یک پیغام چاپ کرده و برنامه خاتمه می‌پذیرد. دقت کنید که خط 70 در برنامه را متمایز کرده‌ایم تا بر این نکته تاکید کند که دسترسی به عنصری خارج از محدوده، خطا بدنبال خواهد داشت. این خطا از نوع خطای منطقی زمان اجرا بوده و یک خطای کامپایل نمی‌باشد.

جالب اینکه، عملگر شاخص آرایه **[]** فقط محدود به استفاده در آرایه‌ها نیست. برای مثال می‌توان از آن برای انتخاب عناصر از انواع کلاس‌های حامل نظیر لیست‌های پیوندی، رشته‌ها و واژه‌نامه استفاده کرد. همچنین پس از تعریف **[]operator**، شاخص دیگر مجبور نیست که حتماً یک مقدار صحیح باشد، کاراکتر، رشته، مقادیر اعشاری و حتی شی‌های تعریف شده توسط کاربر هم می‌توانند بکار گرفته شوند.

تعریف کلاس Array

اکنون که متوجه نحوه عملکرد برنامه شده‌اید، اجازه دهید به سراغ سرآیند کلاس برویم (شکل ۶-۱۱). همانطوری که به هر تابع عضو در سرآیند مراجعه می‌کنیم به توضیح پیاده‌سازی تابع در شکل ۷-۱۱ می‌پردازیم. در شکل ۶-۱۱، خطوط 36-35 عرضه کننده اعضای داده **private** کلاس **Array** هستند. هر شی **Array** متشکل از یک عضو **size** است که نشاندهنده تعداد عناصر در آرایه بوده و یک اشاره‌گر صحیح بنام **ptr** که به یک آرایه صحیح مبتنی بر اشاره‌گر و اخذ شده بفرم دینامیکی اشاره دارد، این آرایه توسط شی **Array** مدیریت می‌شود.

سربارگذاری عملگرهای درج و استخراج بعنوان friend

خطوط 13-12 از شکل ۶-۱۱ مبادرت به اعلان عملگرهای درج و استخراج سربارگذاری شده بعنوان دوستان (friend) کلاس **Array** کرده‌اند. زمانیکه کامپایلر به عبارتی مانند **cout<<arrayObject** برمی‌خورد، مبادرت به احضار تابع سراسری **operator<<** با فراخوانی

```
operator<<(cout, arrayObjet)
```

و زمانیکه کامپایلر به عبارتی مانند **cin>>arrayObject** برمی‌خورد، مبادرت به احضار تابع سراسری **operator>>** با فراخوانی

```
operator>>(cin, arrayObject)
```

می‌نماید. مجدداً توجه کنید که این توابع عملگر درج و استخراج نمی‌توانند عضو کلاس **Array** باشند، چرا که شی **Array** همیشه در طرف راست عملگر درج و استخراج جای داده می‌شود. اگر این توابع

www.yourshop.4kia.ir

عملگر اعضای از کلاس **Array** باشند، مجبور هستیم از عبارات غیراستادانه و ضعیف زیر برای چاپ و ورود آرایه استفاده کنیم:

```
arrayObject << cout;
arrayObject >> cin;
```

چنین عباراتی می‌توانند اکثر برنامه‌نویسان C++ را سردرگم کنند.

تابع **operator>>** (تعریف شده در شکل ۷-۱۱، خطوط 144-127) تعداد عناصر را براساس **size** از آرایه صحیح که **ptr** به آن اشاره می‌کند چاپ می‌نماید. تابع **operator<<** (تعریف شده در شکل ۷-۱۱، خطوط 124-118) بصورت مستقیم داده‌ها را به آرایه‌ای که **ptr** به آن اشاره دارد، وارد می‌سازد. هر یک از این توابع عملگر یک مراجعه برگشت می‌دهند تا بتوان عبارات خروجی یا ورودی پشت سرهم داشت.

دقت کنید که هر یک از این توابع دارای دسترسی به داده **private** آرایه هستند، چرا که این توابع بعنوان **friend** کلاس **Array** اعلام شده‌اند. همچنین توجه کنید که می‌توان توابع **getSize** و **operator[]** را توسط **operator>>** و **operator<<** بکار گرفت، در چنین حالتی نیازی نیست که این توابع عملگر، دوستان کلاس **Array** باشند. با این وجود، فراخوانی بیشتر تابع سبب افزایش زمان اجرا می‌شود.

سازنده پیش‌فرض Array

خط 15 از شکل ۶-۱۱ مبادرت به اعلان سازنده پیش‌فرض برای کلاس کرده و سایز اولیه آنرا با 10 عنصر مشخص ساخته است. زمانیکه کامپایلر اعلانی همانند خط 13 در شکل ۸-۱۱ را مشاهده می‌کند، مبادرت به احضار سازنده پیش‌فرض **Array** می‌کند. سازنده پیش‌فرض (تعریف شده در شکل ۷-۱۱، خطوط 18-25) شروع به ارزیابی و تخصیص آرگومان به عضو داده **size** کرده، از **new** برای بدست آوردن حافظه برای این آرایه مبتنی بر اشاره‌گر استفاده کرده و اشاره‌گر برگشتی توسط **new** را به عضو داده **ptr** تخصیص می‌دهد. سپس سازنده از یک عبارت **for** برای تنظیم تمام مقادیر آرایه با صفر استفاده کرده است.

سازنده کپی کننده Array

خط 16 از شکل ۶-۱۱ یک سازنده کپی کننده (تعریف شده در شکل ۷-۱۱، خطوط 36-29) اعلان کرده است که مبادرت به مقداردهی اولیه آرایه با تهیه کپی از روی یک شی آرایه موجود می‌کند. انجام چنین عملی (کپی) بایستی بدقت صورت گیرد تا از اشاره دادن، اشاره‌گر هر دو آرایه به یک مکان در حافظه جلوگیری شود. سازنده‌های کپی کننده زمانی بکار گرفته می‌شود که یک کپی از شی مورد نیاز باشد، همانند زمانیکه یک شی به روش مقدار به یک تابع ارسال می‌شود، یک شی به روش مقدار از یک تابع برگشت داده می‌شود یا مقداردهی اولیه یک شی با کپی از روی شی دیگر از همان کلاس. سازنده کپی

www.yourshop.4kia.ir

کننده زمانیکه یک شی از کلاس **Array** نمونه‌سازی و با شی دیگر از کلاس **Array** مقداردهی می‌شود، همانند اعلان موجود در خط 41 از شکل ۸-۱۱ فراخوانی می‌گردد.

سازنده کپی کننده **Array** از یک مقداردهی کننده (شکل ۷-۱۱، خط 30) برای کپی سایز مقداردهی کننده **Array** به عضو داده **size** استفاده کرده است. همچنین از **new** (خط 32) برای بدست آوردن حافظه برای این آرایه مبتنی بر اشاره‌گر و تخصیص اشاره‌گر برگشتی توسط **new** به عضو داده **ptr** استفاده کرده است. سپس سازنده کپی کننده با استفاده از یک عبارت **for** مبادرت به کپی تمام عناصر از آرایه مقداردهی کننده به آرایه جدید می‌کند.

نابود کننده Array

خط 17 از شکل ۶-۱۱ مبادرت به اعلان نابوده کننده برای کلاس کرده است (تعریف شده در شکل ۷-۱۱، خطوط 42-39). نابود کننده زمانی برای یک شی از کلاس **Array** احضار می‌شود که از قلمرو خارج شده باشد. نابود کننده از **[]delete** برای رهاسازی حافظه اخذ شده دینامیکی توسط **new** استفاده کرده است.

تابع عضو getSize

خط 18 از شکل ۶-۱۱ تابع **getSize** (تعریف شده در شکل ۷-۱۱، خطوط 48-45) را اعلان کرده است که تعداد عناصر در آرایه را برگشت می‌دهد.

عملگر تخصیص سربارگذاری شده

خط 20 از شکل ۶-۱۱ مبادرت به اعلان عملگر تخصیص سربارگذاری شده برای کلاس کرده است. زمانیکه کامپایلر به عبارت **integers1= integers2** در خط 49 از شکل ۸-۱۱ می‌رسد، تابع عضو **operator=** را با فراخوانی عبارت زیر احضار می‌کند.

```
integers1.operator=(integers2)
```

پیاده‌سازی تابع عضو **operator=** (شکل ۷-۱۱، خطوط 70-52) اقدام به تست خود تخصیصی (خط 54) می‌کند که در آن یک شی از کلاس **Array** به خودش تخصیص می‌یابد. زمانیکه **this** معادل با آدرس عملوند **right** باشد، پس مبادرت به خود تخصیصی شده است و از اینرو تخصیص به کنار گذاشته می‌شود (یعنی در حال حاضر شی خودش است). اگر نتیجه کار یک خود تخصیصی نباشد، پس تابع عضو تعیین می‌کند که آیا سایز دو آرایه با هم برابرند یا خیر (خط 58)، اگر برابر باشند، مقادیر آرایه اصلی در سمت چپ شی **Array** مجدداً اخذ نمی‌شود. در غیر اینصورت **operator=** از **delete** (خط 60) برای رها کردن حافظه اولیه اخذ شده برای آرایه هدف استفاده کرده، سایز آرایه منبع را به سایز (size) آرایه هدف

www.yourshop.4kia.ir

کپی می کند (خط 61)، از **new** برای اخذ حافظه برای آرایه هدف استفاده کرده و اشاره‌گر برگشتی از **new** را در **ptr** قرار می‌دهد. سپس عبارت **for** در خطوط 66-65 شروع به کپی عناصر آرایه از آرایه منبع به آرایه هدف می‌کند صرفنظر از اینکه خود تخصیصی رخ می‌دهد یا خیر، تابع عضو مبادرت به برگشت شی جاری (یعنی **this*** در خط 69) بعنوان یک مراجعه ثابت می‌کند، چنین کاری امکان تخصیص پشت سرهم همانند **x=y=z** را فراهم می‌آورد. اگر خود تخصیصی رخ دهد و تابع **operator=** اینحالت را تست نکند، **operator** مبادرت به حذف حافظه دینامیکی مرتبط با شی **Array** می‌کند، قبل از اینکه عملیات تخصیص کامل شود. در این وضعیت **ptr** به حافظه‌ای اشاره دارد که قبلاً بازپس گرفته شده است، و چنین کاری می‌تواند برنامه را بسوی *خطاهای زمان اجرای عظیم* (*fatal runtime error*) رهنمون سازد.

عملگرهای تساوی نابرابری سربارگذاری شده

خط 21 از شکل ۶-۱۱ مبادرت به اعلان عملگر تساوی سربارگذاری شده (==) برای کلاس کرده است. زمانیکه کامپایلر به عبارت **integers1== integers2** در خط 57 از شکل ۸-۱۱ می‌رسد، تابع عضو **operator==** را با فراخوانی عبارت زیر احضار می‌کند

```
integers1.operator==(integers2)
```

تابع عضو **operator==** (تعریف شده در شکل ۷-۱۱، خطوط 84-74) بلافاصله اگر مقدار **size** آرایه‌ها با هم برابر نباشند، **false** برگشت می‌دهد. در غیر اینصورت، **operator==** شروع به مقایسه هر جفت عناصر می‌کند. اگر همگی با هم برابر باشند، تابع مقدار **true** برگشت می‌دهد. با اولین برخورد به جفت عنصر غیر برابر، تابع بلافاصله مقدار **false** برگشت خواهد داد.

خطوط 27-24 از سرآیند فایل تعریف کننده عملگر نابرابری سربارگذاری شده (**=!**) برای کلاس هستند. تابع عضو **operator!=** از تابع **operator==** سربارگذاری شده برای تعیین اینکه یک آرایه با دیگری برابر است یا خیر استفاده می‌کند، سپس نتیجه مقتضی را برگشت می‌دهد. نوشتن **operator!=** به این روش به برنامه‌نویس امکان می‌دهد تا از **operator==** استفاده مجدد کند که نتیجه آن کاهش کدنویسی برای کلاس است. همچنین توجه کنید که کل تعریف تابع برای **operator!=** در فایل سرآیند **Array** قرار دارد. اینحالت به کامپایلر اجازه می‌دهد تا بصورت *inline* از **operator!=** استفاده کرده و جلوی فراخوانی اضافی تابع گرفته شود.

www.yourshop.4kia.ir

عملگر شاخص سربارگذاری شده

خطوط 30 و 33 از شکل ۶-۱۱ دو عملگر شاخص سربارگذاری شده اعلان کرده‌اند (تعریف شده در شکل ۷-۱۱ در خطوط 99-88 و 114-103). زمانیکه کامپایلر به عبارت **integers1[5]** (شکل ۸-۱۱، خط 61) می‌رسد، مبادرت به احضار تابع عضو سربارگذاری شده مقتضی **[]operator** با فراخوانی

```
integers1.operator[](5)
```

می‌کند. کامپایلر فراخوانی را بر روی نسخه ثابت **const** از **[]operator** انجام می‌دهد (شکل ۷-۱۱، خطوط 114-103)، زمانیکه عملگر شاخص بر روی یک شی آرایه ثابت بکار گرفته شده باشد. برای مثال، اگر شی ثابت **z** با عبارت زیر نمونه‌سازی شده باشد

```
const Array z(5);
```

پس نسخه ثابت از **[]operator** برای اجرای عبارتی مانند عبارت زیر لازم است

```
cout << z[3] << endl;
```

بخاطر داشته باشید که برنامه فقط می‌تواند توابع عضو ثابت **s** از یک شی ثابت را احضار کند.

هر تعریفی از **[]operator** تعیین می‌کند که آیا شاخص یک آرگومان در محدودهٔ دریافت می‌کند یا خیر. اگر چنین نباشد، هر تابع یک پیغام خطا چاپ کرده و برنامه با فراخوانی تابع **exit** خاتمه می‌پذیرد (سرآیند <**cstdlib**>). اگر شاخص در محدودهٔ قرار داشته باشد، نسخه غیرثابت **[]operator** عنصر آرایه مناسب را بعنوان یک مراجعه برگشت می‌دهد. از اینروست که می‌تواند بعنوان یک *lvalue* تغییرپذیر بکار گرفته شود (مثلاً در سمت چپ یک عبارت تخصیص). اگر شاخص در محدودهٔ قرار داشته باشد، نسخه ثابت **[]operator** یک کپی از عنصر مقتضی از آرایه را برگشت می‌دهد. کاراکتر برگشتی یک *rvalue* است.

۹-۱۱ تبدیل مابین نوع‌ها

اکثر برنامه‌ها مبادرت به پردازش اطلاعات از نوع‌های مختلف می‌کنند. گاهی اوقات تمام عملیات «در درون یک نوع» باقی می‌ماند. برای مثال، جمع یک **int** با یک **int**، یک **int** تولید می‌کند (مادامیکه نتیجه بدست آمده بسیار بزرگتر از **int** نباشد). با این همه، گاهی اوقات ضروری است که بتوان یک داده از یک نوع را به نوع دیگری تبدیل کرد. اینکار می‌تواند در هنگام تخصیص، در محاسبات، ارسال مقادیر به توابع و مقادیر برگشتی از توابع صورت گیرد. کامپایلر از نحوه تبدیلات مشخص در میان نوع‌های بنیادین مطلع است (همانطوری که در فصل ششم توضیح داده شد). اما تکلیف نوع‌های تعریف شده توسط کاربر چیست؟ کامپایلر با نحوه تبدیل مابین نوع‌های تعریف شده توسط کاربر و نوع‌های بنیادین مطلع نیست. از اینرو بایستی خود برنامه‌نویس نحوه انجام اینکار را مشخص نماید. چنین تبدیلاتی را می‌توان با *سازنده‌های تبدیل* انجام داد، سازنده‌های تک آرگومانی که شی‌ها از نوع‌های دیگر را (شامل نوع‌های بنیادین) به

www.yourshop.4kia.ir

شی‌های از یک کلاس خاص تبدیل می‌کنند. در بخش ۱۰-۱۱ از یک سازنده تبدیل برای تبدیل رشته‌های **char *** به شی‌های کلاس **String** استفاده کرده‌ایم.

عملگر تبدیل (که عملگر *cast* هم نامیده می‌شود) می‌تواند برای تبدیل یک شی از یک کلاس به یک شی از کلاس دیگر یا یک شی از نوع بنیادین بکار گرفته شود. چنین عملگر تبدیلی باید یک تابع عضو غیراستاتیک باشد. نمونه اولیه تابع

```
A::operator char *() const;
```

یک عملگر تبدیل سربارگذاری شده برای تبدیل یک شی از نوع تعریف شده توسط کاربر **A** به یک شی موقت **char*** اعلان کرده است. تابع اعلان شده **const** (ثابت) می‌باشد چرا که نمی‌تواند شی اصلی را دچار تغییر سازد. یک تابع عملگر تبدیل سربارگذاری شده نمی‌تواند نوع برگشتی را مشخص نماید، نوع برگشتی، نوعی است که شی به آن تبدیل خواهد شد. اگر **s** یک شی از کلاسی باشد، زمانیکه کامپایلر با عبارت **static_cast<char*>(s)** مواجه شود، عبارت زیر را فراخوانی می‌کند

```
s.operator char *()
```

عملوند **s** شی از کلاس **s** است که تابع عضو **operator char *** را احضار می‌کند.

می‌توان توابع عملگر تبدیل سربارگذاری شده را برای تبدیل شی‌ها از نوع تعریف شده توسط کاربر به نوع‌های بنیادین یا شی‌ها از شی دیگر تعریف کرد. نمونه اولیه

```
A::operator int() const;
A::operator OtherClass() const;
```

توابع عملگر تبدیل سربارگذاری شده را اعلان کرده که می‌تواند به ترتیب یک شی از نوع تعریف شده کاربر **A** را به یک نوع **int** یا یک شی از نوع تعریف شده توسط کاربر **OtherClass** تبدیل کند.

۱۰-۱۱ مبحث آموزشی: کلاس String

با هدف، داشتن یک تمرین مناسب از مبحث سربارگذاری، اقدام به ایجاد کلاس **String** متعلق بخود می‌کنیم که قادر به ایجاد و دستکاری رشته‌ها است (شکل‌های ۹-۱۱ الی ۱۱-۱۱). البته کتابخانه استاندارد C++ کلاس **string** مشابه و قدرتمندی دارد. از کلاس استاندارد **string** در بخش ۱۱-۱۳ در یک مثال استفاده می‌کنیم و در فصل هیجدهم به دقت به این کلاس می‌پردازیم. اما برای این لحظه، از ویژگی سربارگذاری عملگر به منظور ساخت کلاس **String** متعلق بخودمان استفاده می‌کنیم.

ابتدا، به معرفی فایل سرآیند کلاس **String** می‌پردازیم. در مورد داده خصوصی بکار رفته در عرضه شی‌های **String** توضیح می‌دهیم. سپس به سراغ واسط **public** کلاس رفته و هر یک از سرویس‌های کلاس را توضیح می‌دهیم. به بررسی تعاریف تابع عضو برای کلاس **String** می‌پردازیم. برای هر تابع

www.yourshop.4kia.ir

عملگر سربارگذاری شده، کدی را که سبب احضار تابع عملگر سربارگذاری شده است، عرضه کرده و توضیحی از نحوه عملکرد آنها خواهیم داد.

تعریف کلاس String

اکنون اجازه دهید به سراغ فایل سرآیند کلاس **String** در شکل ۹-۱۱ برویم. کار را با ارائه دهنده یک رشته داخلی مبتنی بر اشاره‌گر شروع می‌کنیم. خطوط 56-55 اعضای داده **private** را اعلان می‌کنند. کلاس **String** دارای یک فیلد **Length** می‌باشد که نشاندهنده تعداد کاراکترها در رشته است و شامل کاراکتر **null** در انتهای رشته نمی‌باشد و دارای یک اشاره‌گر بنام **sPtr** است که به حافظه اخذ شده دینامیکی برای رشته کاراکتری اشاره دارد.

```
1  // Fig. 11.9: String.h
2  // String class definition.
3  #ifndef STRING_H
4  #define STRING_H
5
6  #include <iostream>
7  using std::ostream;
8  using std::istream;
9
10 class String
11 {
12    friend ostream &operator<<( ostream &, const String & );
13    friend istream &operator>>( istream &, String & );
14 public:
15    String( const char * = "" ); // conversion/default constructor
16    String( const String & ); // copy constructor
17    ~String(); // destructor
18
19    const String &operator=( const String & ); // assignment operator
20    const String &operator+=( const String & ); // concatenation operator
21
22    bool operator!() const; // is String empty?
23    bool operator==( const String & ) const; // test s1 == s2
24    bool operator<( const String & ) const; // test s1 < s2
25
26    // test s1 != s2
27    bool operator!=( const String &right ) const
28    {
29       return !( *this == right );
30    } // end function operator!=
31
32    // test s1 > s2
33    bool operator>( const String &right ) const
34    {
35       return right < *this;
36    } // end function operator>
37
38    // test s1 <= s2
39    bool operator<=( const String &right ) const
40    {
41       return !( right < *this );
42    } // end function operator <=
43
44    // test s1 >= s2
45    bool operator>=( const String &right ) const
46    {
47       return !( *this < right );
48    } // end function operator>=
49
```

www.yourshop.4kia.ir

```
50    char &operator[]( int ); // subscript operator (modifiable lvalue)
51    char operator[]( int ) const; // subscript operator (rvalue)
52    String operator()( int, int = 0 ) const; // return a substring
53    int getLength() const; // return string length
54 private:
55    int length; // string length (not counting null terminator)
56    char *sPtr; // pointer to start of pointer-based string
57
58    void setString( const char * ); // utility function
59 }; // end class String
60
61 #endif
```

شکل ۹-۱۱ | تعریف کلاس String با سربارگذاری عملگر.

```
1   // Fig. 11.10: String.cpp
2   // Member-function definitions for class String.
3   #include <iostream>
4   using std::cerr;
5   using std::cout;
6   using std::endl;
7
8   #include <iomanip>
9   using std::setw;
10
11  #include <cstring> // strcpy and strcat prototypes
12  using std::strcmp;
13  using std::strcpy;
14  using std::strcat;
15
16  #include <cstdlib> // exit prototype
17  using std::exit;
18
19  #include "String.h" // String class definition
20
21  // conversion (and default) constructor converts char * to String
22  String::String( const char *s )
23     : length( ( s != 0 ) ? strlen( s ) : 0 )
24  {
25     cout << "Conversion (and default) constructor: " << s << endl;
26     setString( s ); // call utility function
27  } // end String conversion constructor
28
29  // copy constructor
30  String::String( const String &copy )
31     : length( copy.length )
32  {
33     cout << "Copy constructor: " << copy.sPtr << endl;
34     setString( copy.sPtr ); // call utility function
35  } // end String copy constructor
36
37  // Destructor
38  String::~String()
39  {
40     cout << "Destructor: " << sPtr << endl;
41     delete [] sPtr; // release pointer-based string memory
42  } // end ~String destructor
43
44  // overloaded = operator; avoids self assignment
45  const String &String::operator=( const String &right )
46  {
47     cout << "operator= called" << endl;
48
49     if ( &right != this ) // avoid self assignment
50     {
51        delete [] sPtr; // prevents memory leak
52        length = right.length; // new String length
53        setString( right.sPtr ); // call utility function
54     } // end if
55     else
56        cout << "Attempted assignment of a String to itself" << endl;
```

www.yourshop.4kia.ir

```
   return *this; // enables cascaded assignments
} // end function operator=

// concatenate right operand to this object and store in this object
const String &String::operator+=( const String &right )
{
   size_t newLength = length + right.length; // new length
   char *tempPtr = new char[ newLength + 1 ]; // create memory

   strcpy( tempPtr, sPtr ); // copy sPtr
   strcpy( tempPtr + length, right.sPtr ); // copy right.sPtr

   delete [] sPtr; // reclaim old space
   sPtr = tempPtr; // assign new array to sPtr
   length = newLength; // assign new length to length
   return *this; // enables cascaded calls
} // end function operator+=

// is this String empty?
bool String::operator!() const
{
   return length == 0;
} // end function operator!

// Is this String equal to right String?
bool String::operator==( const String &right ) const
{
   return strcmp( sPtr, right.sPtr ) == 0;
} // end function operator==

// Is this String less than right String?
bool String::operator<( const String &right ) const
{
   return strcmp( sPtr, right.sPtr ) < 0;
} // end function operator<

// return reference to character in String as a modifiable lvalue
char &String::operator[]( int subscript )
{
   // test for subscript out of range
   if ( subscript < 0 || subscript >= length )
   {
      cerr << "Error: Subscript " << subscript
         << " out of range" << endl;
      exit( 1 ); // terminate program
   } // end if

   return sPtr[ subscript ]; // non-const return; modifiable lvalue
} // end function operator[]

// return reference to character in String as rvalue
char String::operator[]( int subscript ) const
{
   // test for subscript out of range
   if ( subscript < 0 || subscript >= length )
   {
      cerr << "Error: Subscript " << subscript
           << " out of range" << endl;
      exit( 1 ); // terminate program
   } // end if

   return sPtr[ subscript ]; // returns copy of this element
} // end function operator[]

// return a substring beginning at index and of length subLength
String String::operator()( int index, int subLength ) const
{
   // if index is out of range or substring length < 0,
   // return an empty String object
```

www.yourshop.4kia.ir

```
   if ( index < 0 || index >= length || subLength < 0 )
      return ""; // converted to a String object automatically

   // determine length of substring
   int len;

   if ( ( subLength == 0 ) || ( index + subLength > length ) )
      len = length - index;
   else
      len = subLength;

   // allocate temporary array for substring and
   // terminating null character
   char *tempPtr = new char[ len + 1 ];

   // copy substring into char array and terminate string
   strncpy( tempPtr, &sPtr[ index ], len );
   tempPtr[ len ] = '\0';

   // create temporary String object containing the substring
   String tempString( tempPtr );
   delete [] tempPtr; // delete temporary array
   return tempString; // return copy of the temporary String
} // end function operator()

// return string length
int String::getLength() const
{
   return length;
} // end function getLength

// utility function called by constructors and operator=
void String::setString( const char *string2 )
{
   sPtr = new char[ length + 1 ]; // allocate memory

   if ( string2 != 0 ) // if string2 is not null pointer, copy contents
      strcpy( sPtr, string2 ); // copy literal to object
   else // if string2 is a null pointer, make this an empty string
      sPtr[ 0 ] = '\0'; // empty string
} // end function setString

// overloaded output operator
ostream &operator<<( ostream &output, const String &s )
{
   output << s.sPtr;
   return output; // enables cascading
} // end function operator<<

// overloaded input operator
istream &operator>>( istream &input, String &s )
{
   char temp[ 100 ]; // buffer to store input
   input >> setw( 100 ) >> temp;
   s = temp; // use String class assignment operator
   return input; // enables cascading
} // end function operator>>
```

شکل ۱۰-۱۱ | تعریف تابع عضو کلاس String و تابع friend.

```
// Fig. 11.11: fig11_11.cpp
// String class test program.
#include <iostream>
using std::cout;
using std::endl;
using std::boolalpha;

#include "String.h"

int main()
{
```

www.yourshop.4kia.ir

```
   String s1( "happy" );
   String s2( " birthday" );
   String s3;

   // test overloaded equality and relational operators
   cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
      << "\"; s3 is \"" << s3 << '\"'
      << boolalpha << "\n\nThe results of comparing s2 and s1:"
      << "\ns2 == s1 yields " << ( s2 == s1 )
      << "\ns2 != s1 yields " << ( s2 != s1 )
      << "\ns2 >  s1 yields " << ( s2 > s1 )
      << "\ns2 <  s1 yields " << ( s2 < s1 )
      << "\ns2 >= s1 yields " << ( s2 >= s1 )
      << "\ns2 <= s1 yields " << ( s2 <= s1 );


   // test overloaded String empty (!) operator
   cout << "\n\nTesting !s3:" << endl;

   if ( !s3 )
   {
      cout << "s3 is empty; assigning s1 to s3;" << endl;
      s3 = s1; // test overloaded assignment
      cout << "s3 is \"" << s3 << "\"";
   } // end if

   // test overloaded String concatenation operator
   cout << "\n\ns1 += s2 yields s1 = ";
   s1 += s2; // test overloaded concatenation
   cout << s1;

   // test conversion constructor
   cout << "\n\ns1 += \" to you\" yields" << endl;
   s1 += " to you"; // test conversion constructor
   cout << "s1 = " << s1 << "\n\n";

   // test overloaded function call operator () for substring
   cout << "The substring of s1 starting at\n"
      << "location 0 for 14 characters, s1(0, 14), is:\n"
      << s1( 0, 14 ) << "\n\n";

   // test substring "to-end-of-String" option
   cout << "The substring of s1 starting at\n"
      << "location 15, s1(15), is: "
      << s1( 15 ) << "\n\n";

   // test copy constructor
   String *s4Ptr = new String( s1 );
   cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";

   // test assignment (=) operator with self-assignment
   cout << "assigning *s4Ptr to *s4Ptr" << endl;
   *s4Ptr = *s4Ptr; // test overloaded assignment
   cout << "*s4Ptr = " << *s4Ptr << endl;

   // test destructor
   delete s4Ptr;

   // test using subscript operator to create a modifiable lvalue
   s1[ 0 ] = 'H';
   s1[ 6 ] = 'B';
   cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
      << s1 << "\n\n";

   // test subscript out of range
   cout << "Attempt to assign 'd' to s1[30] yields:" << endl;
   s1[ 30 ] = 'd'; // ERROR: subscript out of range
   return 0;
} // end main
```

```
Conversion (and default) constructor: happy
```

www.yourshop.4kia.ir

```
Conversion (and default) constructor: birthday
Conversion (and default) constructor:
s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing !s3:
s3 is empty; assiging s1 to s3;
operator= called
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
Conversion (and default) constructor: to you
Destructor: to you
s1 = happy birthday to you

Conversion (and default) constractor: happy birthday
copy constructor: happy birthday
Destructor: happy birthday
The substring of s1 starting at
location 0 for 14 characters, s1(0, 14), is:
happy birthday

Destructor: happy birthday
Conversion (and default) constructor: to you
Copy constructor: to you
Destructor: to you
The substring of s1 starting at
location 15, s1(15), is: to you

Destructor: to you
Copy constructor: happy birthday to you

*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
operator= called
Attempted assignment of a String to itself

*s4Ptr = happy birthday to you
Destructor: happy birthday to you

s1 after s1[0] = 'H'and s1[6] = 'B'is: Happy Birthday to you

Attempt to assign 'd'to s1[30] yields:
Error: Subscript 30 out of range
```

شکل ۱۱-۱۱ | برنامه تست کننده کلاس String.

سربارگذاری عملگرهای درج و استخراج بعنوان friend

خطوط 12-13 (شکل ۹-۱۱) مبادرت به اعلان تابع عملگر درج سربارگذاری شده **operator<<** (تعریف شده شکل ۱۰-۱۱، خطوط 170-174) و تابع عملگر استخراج سربارگذاری شده **operator>>** (تعریف شده در شکل ۱۰-۱۱، خطوط 177-183) بعنوان **friend** کلاس کرده‌اند. پیاده‌سازی **operator<<** سر راست است. دقت کنید که **operator<<** محدود به تعداد کاراکترهای است که می‌توان به آرایه از **temp**

www.yourshop.4kia.ir

تا 99 با **setw** (خط 180) خواند، مکان صدم برای کاراکتر خاتمه دهنده ***null***، در نظر گرفته شده است. همچنین به نحوه استفاده از **operator=** (خط 181) در تخصیص رشته **temp** به شی **String** که **s** به آن اشاره دارد، توجه کنید. این عبارت سازنده تبدیل کننده را برای ایجاد کد شی **String** موقت که حاوی رشته‌ای به سبک C است احضار کرده، سپس رشته موقت **String** به **s** تخصیص می‌یابد. می‌توانیم ایجاد موقت شی **String** بکار رفته در اینجا را به کمک یک عملگر تخصیص سربارگذاری شده که پارامتری از نوع ***const char** دریافت می‌کند، برطرف نمائیم.

سازنده تبدیل کننده String

در خط 15 (شکل ۱۹-۱۱) یک سازنده تبدیل کننده اعلان شده است. این سازنده (تعریف شده در شکل ۱۰-۱۱، خطوط 27-22) یک آرگومان ***const char** دریافت (پیش‌فرض رشته تهی، شکل ۹-۱۱، خط 15) و شی **String** را مقداردهی اولیه می‌کند. هر سازنده تک آرگومانی را می‌توان بعنوان یک سازنده تبدیل کننده بکار گرفت. همانطوری که خواهید دید، چنین تبدیل کننده‌های به هنگام کار با رشته‌ای با آرگومان ***char** سودمند هستند. سازنده تبدیل کننده قادر به تبدیل یک رشته ***char** به یک شی **String** است، که سپس می‌تواند به شی **String** هدف تخصیص داده شود. مزیت این سازنده تبدیل کننده در این است که نیازی به تدارک دیدن یک عملگر تخصیص سربارگذاری شده برای تخصیص رشته‌های کاراکتری به شی‌های **String** ندارد. کامپایلر، سازنده تبدیل کننده را برای ایجاد یک شی **String** موقت که حاوی رشته کاراکتری است احضار کرده، سپس عملگر تخصیص سربارگذاری شده را برای انتساب شی **String** موقت به شی **String** دیگر احضار می‌کند.

سازنده تبدیل کننده **String** می‌تواند در اعلانی نظیر **s1("happy")** احضار شود. سازنده تبدیل کننده طول کاراکترهای موجود در آرگومان خود را محاسبه و آنرا به عضو داده **length** در لیست مقداردهی اولیه تخصیص می‌دهد. سپس خط 26 تابع کمکی **setString** (تعریف شده در شکل ۱۰-۱۱، خطوط 167-159) را که از **new** برای اخذ حافظه کافی برای عضو داده خصوصی **sptr** استفاده می‌کند فراخوانی کرده و از **strcpy** برای کپی رشته کاراکتری به حافظه‌ای که **sPtr** به آن اشاره دارد، استفاده می‌کند.

سازنده کپی کننده String

خط 16 در شکل ۹-۱۱ یک سازنده کپی کننده (تعریف شده در شکل ۱۰-۱۱، خطوط 35-30) اعلان کرده است که مبادرت به مقداردهی یک شی **String** با ایجاد یک کپی از روی یک شی **String** موجود می‌کند. همانند کلاس **Array** (شکل‌های ۶-۱۱ و ۷-۱۱)، انجام چنین عملی باید با دقت صورت گیرد تا هر دو شی **String** به یک حافظه اشاره نکند. عملگر سازنده کپی کننده همانند سازنده تبدیل کننده است،

www.yourshop.4kia.ir

بجز اینکه این سازنده فقط عضو **Length** از شی **String** منبع را به شی **String** هدف کپی می‌کند. دقت کنید که سازنده کپی کننده **setString** را برای ایجاد یک فضای جدید برای شی هدف فراخوانی می‌کند. اگر این سازنده فقط مبادرت به کپی **sPtr** در شی منبع به **sPtr** شی هدف می‌کرد، آنگاه هر دو شی به یک حافظه اشاره می‌کردند. با اجرای اولین نابود کننده، حافظه حذف می‌گردید و **sPtr** شی‌های دیگر بلاتکلیف باقی می‌ماندند (یعنی **sPtr** تبدیل به یک اشاره‌گر *dangling* یا آویزان می‌شود) و در نتیجه، خطاهای زمان اجرای بسیاری جدی رخ می‌دهد.

نابود کننده String

خط 17 از شکل ۹-۱۱ مبادرت به اعلان نابود کننده **String** (تعریف شده در شکل ۱۰-۱۱، خطوط 38-42) کرده است. نابود کننده با استفاده **[]delete** حافظه دینامیکی که **sPtr** به آن اشاره می‌کند، آزاد می‌سازد.

عملگر تخصیص سربارگذاری شده

خط 19 (شکل ۹-۱۱) تابع عملگر تخصیص سربارگذاری شده **operator=** را اعلان کرده است (تعریف شده در خطوط 45-59 شکل ۱۰-۱۱). زمانیکه کامپایلر به عبارتی مانند **string1=string2** می‌رسد، تابع زیر را فراخوانی می‌کند

```
string1.operator=(string2);
```

تابع عملگر تخصیص سربارگذاری شده **operator=** مبادرت به تست خود تخصیصی می‌کند. اگر این تخصیص، یک خود تخصیصی باشد، تابع نیازی به تغییر شی ندارد. اگر این تست حذف شود یا انجام نشود، تابع بلافاصله اقدام به حذف فضای شی هدف کرده و از اینرو رشته کاراکتری از دست می‌رود. در چنین حالتی دیگر اشاره‌گر به داده معتبر اشاره ندارد (نمونه کلاسیک این حالت اشاره‌گر *dangling* است). اگر نتیجه تست یک خود تخصیصی نباشد، تابع مبادرت به حذف حافظه کرده و فیلد **length** از شی منبع را به شی هدف کپی می‌کند. سپس **operator=** مبادرت به فراخوانی **setString** برای ایجاد یک فضای جدید برای شی هدف کرده و رشته کاراکتری را از شی منبع به شی هدف کپی می‌کند. خواه خود تخصیصی صورت گرفته باشد یا نباشد، **operator =** مبادرت به برگشت دادن **this *** می‌کند تا بتوان پشت سر هم تخصیص انجام داد.

www.yourshop.4kia.ir

عملگر جمع تخصیص سربارگذاری شده

در خط 20 از شکل ۹-۱۱ عملگر اتصال رشته سربارگذاری شده += اعلان شده است (تعریف شده در شکل ۱۰-۱۱، خطوط 74-62). زمانیکه کامپایلر به عبارتی نظیر **s1 += s2** می‌رسد (خط 40 از شکل ۱۱-۱۱)، کامپایلر تابع عضو را فراخوانی می‌کند

```
s1.operator+=(s2)
```

تابع **operator+=** مبادرت به محاسبه طول ترکیب شده از رشته متصل شده کرده و آنرا در متغیر محلی **newLength** ذخیره می‌کند، سپس یک اشاره‌گر موقت (**tempPtr**) ایجاد کرده و یک آرایه کاراکتری جدید اخذ می‌کند که بتوان رشته متصل شده را در آن ذخیره کرد.

سپس، **operator+=** از **strcpy** برای کپی کردن رشت کاراکتری اصلی از **sPtr** و **right.sPtr** به حافظه‌ای که **tempPtr** به آن اشاره دارد، استفاده می‌کند. دقت کنید مکانی که **strcpy** مبادرت به کپی اولین کاراکتر از **right.sPtr** می‌کند با محاسبه ریاضی اشاره‌گر یعنی **tempPtr + length** مشخص می‌شود. این محاسبه بر این نکته دلالت دارد که اولین کاراکتر از **rightsPtr** بایستی در موقعیت **length** در آرایه‌ای که **tempPtr** به آن اشاره دارد، قرار داده شود. سپس **operator+=** از **[]delete** برای رهاسازی فضای اشغال شده توسط رشته کاراکتری شی اصلی استفاده کرده، **tempPtr** را به **sPtr** تخصیص می‌دهد. از اینرو این شی **String** به رشته کاراکتری جدید اشاره می‌کند، **newLength** را به **length** تخصیص و از اینروست که این شی **String** حاوی طول رشته جدید بوده و *** this** را بعنوان یک **const String &** برگشت می‌دهد تا بتوان عملگرهای += پشت سرهم یا دنباله‌دار داشت.

عملگر نفی سربارگذاری شده

خط 22 از شکل ۹-۱۱ عملگر نفی سربارگذاری شده را اعلان کرده است (تعریف شده در شکل ۱۰-۱۱، خطوط 80-77). این عملگر تعیین می‌کند که آیا یک شی از کلاس **String** تهی است یا خیر. برای مثال، زمانیکه کامپایلر به عبارتی مانند **string1!** می‌رسد. فراخوانی تابع زیر را انجام می‌هد

```
string1.operator!()
```

این تابع فقط نتیجه تست را باز می‌گرداند خواه **length** برابر صفر باشد یا نباشد.

عملگرهای برابری و رابطه‌ای سربارگذاری شده

خطوط 24-23 از شکل ۹-۱۱ مبادرت به اعلان عملگر برابری سربارگذاری شده (تعریف شده در شکل ۱۰-۱۱، خطوط 86-83) و عملگر کوچکتر از، سربارگذاری شده (تعریف شده در شکل ۱۰-۱۱، خطوط 92-89) برای کلاس **String** کرده‌اند. این دو شبیه هم هستند، از اینرو اجازه دهید فقط به بررسی یک مثال

www.yourshop.4kia.ir

و آن هم برای عملگر == سربارگذاری شده بپردازیم. زمانیکه کامپایلر به عبارتی همانند **string1==string2** می‌رسد، تابع عضو زیر را فراخوانی می‌کند

```
string1.operator==(string2)
```

در صورتیکه **string1** معادل (برابر) با **string2** باشد، **true** برگشت می‌دهد. هر یک از عملگرها از تابع **strcmp** (از **<cstring>**) برای مقایسه رشته‌های کاراکتری در شی‌های **String** استفاده می‌کن‌ند. تعدادی از برنامه‌نویسان C++ طرفدار استفاده از برخی توابع عملگر سربارگذاری شده برای پیاده‌سازی موارد دیگر هستند. از اینرو، عملگرهای **!=**، **>**، **>=** و **<=** (شکل ۹-۱۱، خطوط 48-27) برحسب **operator==** و **operator<** پیاده‌سازی شده‌اند. برای مثال، تابع سربارگذاری شده **operator>=** (پیاده‌سازی شده در خطوط 48-45 در فایل سرآیند) از عملگر سربارگذاری شده **>** برای تعیین اینکه یک شی **String** بزرگ‌تر یا برابر شی **String** دیگری است یا خیر، استفاده کرده است. دقت کنید که توابع عملگر برای **!=**، **>**، و **<=** در فایل سرآیند تعریف شده‌اند.

عملگرهای شاخص سربارگذاری شده

خطوط 51-50 در فایل سرآیند دو عملگر شاخص سربارگذاری شده اعلان کرده است (تعریف شده در شکل ۱۰-۱۱، خطوط 106-95 و 120-109) یکی برای رشته‌های غیرثابت و یکی برای رشته‌های ثابت. زمانیکه کامپایلر به عبارتی همانند **string1[0]** می‌رسد، تابع عضو زیر را فراخوانی می‌کند

```
string1.operator[](0)
```

هر پیاده‌سازی **operator[]** ابتدا مبادرت به ارزیابی شاخص می‌کند تا مطمئن گردد در محدوده صحیح قرار دارد. اگر شاخص در محدودهٔ قرار نداشته باشد، هر تابع یک پیغام خطا چاپ کرده و برنامه با فراخوانی **exit** خاتمه می‌پذیرد. اگر شاخص در محدودهٔ صحیح قرار داشته باشد، نسخه غیر ثابت **operator[]** یک **char &** به کاراکتر مقتضی از شی **String** برگشت می‌دهد، این **char &** می‌تواند بعنوان یک *lvalue* برای تغییر در یک کاراکتر خاص از شی **String** بکار گرفته شود. نسخه ثابت **operator[]** کاراکتر مقتضی از شی **String** برگشت می‌دهد که از آن می‌توان فقط بعنوان یک *rvalue* به منظور خواندن کاراکتر استفاده کرد.

عملگر فراخوانی تابع سربارگذاری شده

خط 52 از شکل ۹-۱۱ عملگر فراخوانی تابع سربارگذاری شده (تعریف شده در شکل ۱۰-۱۱، خطوط 150-123) را اعلان کرده است. این عملگر را برای انتخاب یک زیر رشته از یک **String** سربارگذاری کرده‌ایم. دو پارامتر صحیح مشخص کننده موقعیت شروع و طول زیر رشته‌ای است که از **String** انتخاب خواهد شد. اگر موقعیت یا مکان شروع در خارج از محدودهٔ قرار داشته باشد یا طول زیر رشته منفی باشد،

www.yourshop.4kia.ir

عملگر فقط یک رشته تهی برگشت خواهد داد. اگر طول زیر رشته صفر باشد، سپس زیر رشته تا انتهای شی رشته انتخاب می‌شود. برای مثال فرض کنید **string1** یک شی از **String** است و حاوی رشته "AEIOU" باشد. در عبارتی مانند **string1(2,2)** کامپایلر فراخوانی تابع عضو زیر را انجام می‌دهد

```
string1.operator()(2,2)
```

با اجرای این فراخوانی، یک شی **String** حاوی رشته "IO" تولید و یک کپی از آن شی برگشت داده می‌شود.

سربارگذاری عملگر فراخوانی تابع **()** ابزار قدرتمندی است چراکه توابع می‌توانند به تعداد دلخواه و ترکیبی پارامتر دریافت کنند. از اینرو می‌توانیم از این قابلیت در زمینه‌های مختلفی استفاده کنیم. یکی از موارد استفاده از عملگر فراخوانی تابع تغییر در نماد شاخص آرایه است. بجای استفاده از نماد براکت در آرایه‌های دوبعدی همانند **a[b][c]**، برخی از برنامه‌نویسان ترجیح می‌دهند با سربارگذاری عملگر فراخوانی تابع از نماد **a(b,c)** استفاده کنند. عملگر فراخوانی تابع سربارگذاری شده باید یک تابع عضو غیراستاتیک باشد. این عملگر فقط زمانی بکار گرفته می‌شود که «نام تابع» یک شی از کلاس **String** باشد.

تابع عضو getLength

خط 53 در شکل ۹-۱۱ تابعی بنام **getLength** (تعریف شده در شکل ۱۰-۱۱، خطوط 156-153) اعلان کرده است، که طول یک رشته را برگشت می‌دهد.

۱۱-۱۱ سربارگذاری ++ و --

نسخه‌های پیشوند و پسوند عملگرهای افزایشی و کاهشی را می‌توان سربارگذاری کرد. خواهیم دید که چگونه کامپایلر قادر به تشخیص نسخه پیشوند و نسخه پسوند از یک عملگر افزایش دهنده یا کاهش دهنده است.

برای سربارگذاری کردن عملگر افزایش دهنده به نحوی که امکان استفاده از هر دو نوع افزایش پیشوندی و پسوندی در آن وجود داشته باشد، بایستی هر عملگر سربارگذاری شده دارای یک امضاء یا هویت متمایز باشد، از اینروست که کامپایلر قادر به تعیین نسخه مورد نظر ++ خواهد بود. نسخه‌های پیشوندی دقیقاً همانند سایر عملگرهای پیشوندی غیربایری سربارگذاری می‌شوند.

سربارگذاری عملگر پیشوند افزایشی

برای مثال، فرض کنید که می‌خواهیم 1 را به شی **d1** از کلاس **Date** اضافه کنیم. زمانیکه کامپایلر به عبارت پیش افزایشی **d1++** برخورد می کند، فراخوانی تابع زیر را انجام می‌دهد

```
d1.operator++()
```

نمونه اولیه این تابع عملگر بصورت زیر است

www.yourshop.4kia.ir

```
Date &operator++();
```

اگر عملگر افزایش دهنده پیشوندی بصورت یک تابع سراسری پیاده‌سازی شده باشد، پس زمانیکه کامپایلر به عبارت **d1**++ برسد، تابع زیر را فراخوانی خواهد کرد

```
operator++(d1)
```

نمونه اولیه این تابع عملگر می‌توانند بصورت زیر در کلاس **Date** اعلان شده باشد

```
Date &operator++(Date &);
```

سربارگذاری عملگر پسوند افزایشی

سربارگذاری عملگر پسوند افزایشی کمی کار دارد چرا که کامپایلر باید قادر به تشخیص و تمایز قائل شدن مابین امضاءهای توابع عملگر پیشوند و پسوند افزایشی باشد. قراردادی که در C++ پذیرفته شده این است که، زمانیکه کامپایلر به عبارت پس افزایشی **d1**++ می‌رسد، فراخوانی تابع عضو

```
d1.operator++(0)
```

را انجام می‌دهد. نمونه اولیه این تابع

```
Date operator++(int)
```

است. آرگومان صفر کاملاً یک «مقدار ساختگی» است که به کامپایلر امکان می‌دهد تا مابین توابع عملگر افزایش پیشوند و پسوند تمایز قائل شود. اگر افزایش پسوندی بصورت یک تابع سراسری پیاده‌سازی شده باشد، سپس به هنگام مشاهده عبارت **d1**++ توسط کامپایلر، فراخوانی زیر صورت می‌گیرد

```
operator++(d1, 0)
```

نمونه اولیه این تابع بصورت زیر است

```
Date operator++(Date &, int);
```

مجدداً، آرگومان صفر توسط کامپایلر برای تشخیص مابین عملگرهای افزایشی پسوندی و پیشوندی بعنوان توابع سراسری استفاده می‌شود. توجه کنید که عملگر افزایشی پسوندی شی‌های **Date** را به روش مقدار برگشت می‌دهد، در حالیکه عملگر افزایشی پیشوندی شی‌های **Date** را به روش مراجعه برگشت می‌دهد، چرا که عملگر افزایشی پسوندی یک شی موقت برگشت می‌دهد که حاوی مقدار اصلی از شی است، قبل از اینکه عملیات افزایش رخ داده باشد. C++ با چنین شی‌های بعنوان *rvalue* رفتار می‌کند که نمی‌توان از آنها در سمت چپ یک عبارت تخصیصی استفاده کرد. عملگر افزایشی پیشوندی، در واقع شی افزایش یافته را به همراه مقدار جدید برگشت می‌دهد. از چنین شی می‌توان بعنوان یک *lvalue* در یک عبارت شمارشی استفاده کرد.

هر عملی که بر روی سربارگذاری عملگرهای افزایشی پسوندی و پیشوندی در این بخش توضیح داده شد در ارتباط با سربارگذاری عملگرهای کاهش پسوندی و پیشوندی نیز کاربرد دارد. در بخش بعد از کلاس **Date** به همراه عملگرهای سربارگذاری شده ++ و -- استفاده می‌کنیم.

www.yourshop.4kia.ir

۱۲-۱۱ مبحث آموزشی: کلاس Date

برنامه موجود در شکل‌های ۱۲-۱۱ الی ۱۴-۱۱ به توصیف کاربردی از کلاس **Date** می‌پردازد. این کلاس از عملگرهای سربارگذاری شده پیش و پس افزایشی برای افزودن 1 به روز در یک شی **Date** استفاده می‌کند، و در صورتیکه نیاز باشد افزایش ماه و سال هم صورت می‌گیرد. فایل سرآیند **Date** (شکل ۱۲-۱۱) مشخص کرده که واسط سراسری **Date** شامل یک عملگر درج سربارگذاری شده (خط 11)، یک سازنده پیش‌فرض (خط 13)، تابع **setDate** (خط 14)، عملگر سربارگذاری شده پیش‌افزایشی (خط 15)، عملگر سربارگذاری شده پس‌افزایشی (خط 16)، عملگر سربارگذاری شده جمع تخصیصی += در خط 17، تابعی برای تست سالهای کبیسه (خط 18) و تابعی برای تعیین اینکه روز بدست آمده آخرین روز از ماه است یا خیر (خط 19) می‌باشد.

```
1  // Fig. 11.12: Date.h
2  // Date class definition.
3  #ifndef DATE_H
4  #define DATE_H
5
6  #include <iostream>
7  using std::ostream;
8
9  class Date
10 {
11    friend ostream &operator<<( ostream &, const Date & );
12 public:
13    Date( int m = 1, int d = 1, int y = 1900 ); // default constructor
14    void setDate( int, int, int ); // set month, day, year
15    Date &operator++(); // prefix increment operator
16    Date operator++( int ); // postfix increment operator
17    const Date &operator+=( int ); // add days, modify object
18    bool leapYear( int ) const; // is date in a leap year?
19    bool endOfMonth( int ) const; // is date at the end of month?
20 private:
21    int month;
22    int day;
23    int year;
24
25    static const int days[]; // array of days per month
26    void helpIncrement(); // utility function for incrementing date
27 }; // end class Date
28
29 #endif
```

شکل ۱۲-۱۱ | تعریف کلاس Date به همراه عملگرهای افزایشی سربارگذاری شده.

```
1   // Fig. 11.13: Date.cpp
2   // Date class member-function definitions.
3   #include <iostream>
4   #include "Date.h"
5
6   // initialize static member at file scope; one classwide copy
7   const int Date::days[] =
8      { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
9
10  // Date constructor
11  Date::Date( int m, int d, int y )
12  {
13     setDate( m, d, y );
14  } // end Date constructor
15
16  // set month, day and year
```

www.yourshop.4kia.ir

```
void Date::setDate( int mm, int dd, int yy )
{
   month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
   year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;

   // test for a leap year
   if ( month == 2 && leapYear( year ) )
      day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
   else
      day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
} // end function setDate

// overloaded prefix increment operator
Date &Date::operator++()
{
   helpIncrement(); // increment date
   return *this; // reference return to create an lvalue
} // end function operator++

// overloaded postfix increment operator; note that the
// dummy integer parameter does not have a parameter name
Date Date::operator++( int )
{
   Date temp = *this; // hold current state of object
   helpIncrement();

   // return unincremented, saved, temporary object
   return temp; // value return; not a reference return
} // end function operator++

// add specified number of days to date
const Date &Date::operator+=( int additionalDays )
{
   for ( int i = 0; i < additionalDays; i++ )
      helpIncrement();

   return *this; // enables cascading
} // end function operator+=

// if the year is a leap year, return true; otherwise, return false
bool Date::leapYear( int testYear ) const
{
   if ( testYear % 400 == 0 ||
      ( testYear % 100 != 0 && testYear % 4 == 0 ) )
      return true; // a leap year
   else
      return false; // not a leap year
} // end function leapYear

// determine whether the day is the last day of the month
bool Date::endOfMonth( int testDay ) const
{
   if ( month == 2 && leapYear( year ) )
      return testDay == 29; // last day of Feb. in leap year
   else
      return testDay == days[ month ];
} // end function endOfMonth

// function to help increment the date
void Date::helpIncrement()
{
   // day is not end of month
   if ( !endOfMonth( day ) )
      day++; // increment day
   else
      if ( month < 12 ) // day is end of month and month < 12
      {
         month++; // increment month
         day = 1; // first day of new month
      } // end if
```

www.yourshop.4kia.ir

```
87        else // last day of year
88        {
89           year++; // increment year
90           month = 1; // first month of new year
91           day = 1; // first day of new month
92        } // end else
93  } // end function helpIncrement
94
95  // overloaded output operator
96  ostream &operator<<( ostream &output, const Date &d )
97  {
98     static char *monthName[ 13 ] = { "", "January", "February",
99        "March", "April", "May", "June", "July", "August",
100       "September", "October", "November", "December" };
101    output << monthName[ d.month ] << ' ' << d.day << ", " << d.year;
102     return output; // enables cascading
103  } // end function operator<<
```

شکل ۱۳-۱۱ | تعاریف تابع عضو و friend کلاس Date.

```
1  // Fig. 11.14: fig11_14.cpp
2  // Date class test program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Date.h" // Date class definition
8
9  int main()
10 {
11    Date d1; // defaults to January 1, 1900
12    Date d2( 12, 27, 1992 ); // December 27, 1992
13    Date d3( 0, 99, 8045 ); // invalid date
14
15    cout << "d1 is " << d1 << "\nd2 is " << d2 << "\nd3 is " << d3;
16    cout << "\n\nd2 += 7 is " << ( d2 += 7 );
17
18    d3.setDate( 2, 28, 1992 );
19    cout << "\n\n  d3 is " << d3;
20    cout << "\n++d3 is " << ++d3 << " (leap year allows 29th)";
21
22    Date d4( 7, 13, 2002 );
23
24    cout << "\n\nTesting the prefix increment operator:\n"
25       << "  d4 is " << d4 << endl;
26    cout << "++d4 is " << ++d4 << endl;
27    cout << "  d4 is " << d4;
28
29    cout << "\n\nTesting the postfix increment operator:\n"
30       << "  d4 is " << d4 << endl;
31    cout << "d4++ is " << d4++ << endl;
32    cout << "  d4 is " << d4 << endl;
33    return 0;
34 } // end main
```

```
d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900

d2 += 7 is January 3, 1993

  d3 is February 28, 1992
++d3 is February 29, 1992 (leap year allows 29th)

Testing the prefix increment operator:
  d4 is July 13, 2002
++d4 is July 14, 2002
  d4 is July 14, 2002

Testing the postfix increment operator:
  d4 is July 14, 2002
d4++ is July 14, 2002
```

www.yourshop.4kia.ir

```
d4 is July 15, 2002
```

شکل ١٤-١١ | برنامه تست کلاس Date.

تابع **main** (شکل ۱۴-۱۱) سه شی **Date** ایجاد کرده است (خطوط 13-11)، **d1** بطور پیش‌فرض با January 1,1900، **d2** با December 27,1992 و **d3** با یک تاریخ غیر معتبر مقداردهی اولیه شده‌اند. سازنده **Date** (تعریف شده در شکل ۱۳-۱۱، خطوط 14-11) تابع **setDate** را به منظور ارزیابی ماه، روز و سال مشخص شده فراخوانی می‌کند. ماه غیر معتبر با 1، سال غیر معتبر با 1900 و روز غیر معتبر با 1 تنظیم می‌شود.

خطوط 16-15 از **main** هر یک از شی‌های ساخته شده از **Date** را با استفاده از عملگر درج سربارگذاری شده (تعریف شده در شکل ۱۳-۱۱، خطوط 103-96) در خروجی قرار می‌دهند. خط 16 از **main** از عملگر سربارگذاری شده =+ به منظور افزودن هفت روز به **d2** استفاده کرده است. خط 18 از تابع **setDate** برای تنظیم **d3** با February 28,1992 استفاده کرده که یک سال کبیسه است. سپس خط 20 بصورت پیش افزایشی **d3** را برای نمایش آن که تاریخ بدرستی به February 29 (29 فوریه) منتقل شده، افزایش داده است. سپس خط 22 یک شی **Date** بنام **d4** ایجاد کرده است که با تاریخ July 13,2002 مقداردهی اولیه شده است. سپس خط 26 مبادرت به افزایش **d4** به میزان 1 توسط عملگر پیش‌افزایشی سربارگذاری شده کرده است. خطوط 27-24 مبادرت به چاپ **d4** قبل و بعد از انجام پیش‌افزایش می‌کنند تا نشاندهند که کار بدرستی انجام گرفته است. سرانجام خط 31 مقدار **d4** را توسط عملگر پس‌افزایشی سربارگذاری شده افزایش داده است. خطوط 32-29 مبادرت به چاپ **d4** قبل و بعد از عملیات پس‌افزایشی کرده‌اند.

سربارگذاری عملگر افزایشی پیشوندی کار سر راستی است. عملگر پیش‌افزایشی (تعریف شده در شکل ۱۳-۱۱، خطوط 34-30) مبادرت به فراخوانی تابع کمکی یا یوتیلیتی **helpIncrement** برای افزایش تاریخ کرده است (تعریف شده در شکل ۱۳-۱۱، خطوط 93-76). عملگرد این تابع همانند یک مراقب یا حامل است و زمانیکه مبادرت به افزودن آخرین روز از ماه می‌کنیم، وارد صحنه می‌شود. اگر در ماه 12 قرار داشته باشیم، پس بایستی سال نیز افزایش داده شود و ماه با 1 تنظیم گردد. تابع **helpIncrement** از تابع **endOfMonth** برای افزایش صحیح روز استفاده می‌کند.

۱۳-۱۱ کلاس string از کتابخانه استاندارد

در این فصل، آموختید که می‌توانید یک کلاس **String** ایجاد کنید (شکل‌های ۹-۱۱ الی ۱۱-۱۱) که بسیار بهتر از رشته‌های * **char** به سبک C بوده و توسط C++ بکار گرفته شده‌اند. همچنین آموختید که

www.yourshop.4kia.ir

می‌توانید یک کلاس **Array** ایجاد کنید (شکل‌های ۶-۱۱ الی ۸-۱۱) که عملکردی بهتر از آرایه مبتنی بر اشاره‌گر به سبک C دارد و توسط ++C بکار گرفته شده‌اند.

ایجاد کلاس‌های سودمند با قابلیت استفاده مجدد همانند **String** و **Array** کاری زمان‌بر است. بطوریکه چنین کلاس‌های برای اینکه بتوانند توسط شما، دانشگاه، شرکت خودتان، سایر شرکت‌ها یا یک مجموعه علمی و صنعتی بکار گرفته شوند بایستی تست و خطایابی دقیق شده باشند. طراحان ++C دقیقاً اینکار را انجام داده‌اند و کلاس‌های **String** و **Vector** را همراه ++C استاندارد کرده‌اند. این کلاس‌ها در دسترس هر کسی که مشغول ایجاد برنامه‌های کاربردی با ++C است، قرار دارند.

برای به پایان بردن این فصل، دوباره به سراغ مثال **String** خود می‌رویم (شکل‌های ۹-۱۱ الی ۱۱-۱۱) و این بار آنرا با کلاس استاندارد **string** پیاده‌سازی می‌کنیم. همان وظایف مثال قبلی را نیز در این مثال اعمال می‌کنیم (البته با استفاده از قابلیت‌های کلاس استاندارد **string**). همچنین به بررسی تابع عضو از کلاس استاندارد **string** بنام‌های **empty** و **at** خواهیم پرداخت که بخشی از مثال **String** ما نبودند. تابع **empty** تعیین می‌کند که آیا رشته تهی است یا خیر. تابع **substr** رشته‌ای برگشت می‌دهد که بخشی از رشته موجود بوده و تابع **at** کاراکتر قرار گرفته در شاخص تعیین شده در رشته را برگشت می‌دهد (البته پس از بررسی قرار داشتن شاخص در محدودهٔ). در فصل هیجدهم به تفصیل به بررسی کلاس **string** خواهیم پرداخت.

کلاس string از کتابخانه استاندارد

برنامه شکل ۱۵-۱۱ پیاده‌سازی مجددی از برنامه ۱۱-۱۱ با استفاده از کلاس استاندارد **string** است. همانطوری که در این مثال خواهید دید، کلاس **string** تمام قابلیت‌های کلاس **String** ما را که در برنامه‌های ۹-۱۱ و ۱۰-۱۱ عرضه شده بودند، دارا است. کلاس **string** در سرآیند <**string**> تعریف شده (خط 7) و متعلق به فضای نامی **std** است (خط 8).

```
1  // Fig. 11.15: fig11_15.cpp
2  // Standard Library string class test program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <string>
8  using std::string;
9
10 int main()
11 {
12    string s1( "happy" );
13    string s2( " birthday" );
14    string s3;
15
16    // test overloaded equality and relational operators
17    cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
18      << "\"; s3 is \"" << s3 << '\"'
19       << "\n\nThe results of comparing s2 and s1:"
20       << "\ns2 == s1 yields " << ( s2 == s1 ? "true" : "false" )
21       << "\ns2 != s1 yields " << ( s2 != s1 ? "true" : "false" )
```

www.yourshop.4kia.ir

```
22       << "\ns2 >  s1 yields " << ( s2 > s1 ? "true" : "false" )
23       << "\ns2 <  s1 yields " << ( s2 < s1 ? "true" : "false" )
24       << "\ns2 >= s1 yields " << ( s2 >= s1 ? "true" : "false" )
25       << "\ns2 <= s1 yields " << ( s2 <= s1 ? "true" : "false" );
26
27    // test string member function empty
28    cout << "\n\nTesting s3.empty():" << endl;
29
30    if ( s3.empty() )
31    {
32       cout << "s3 is empty; assigning s1 to s3;" << endl;
33       s3 = s1; // assign s1 to s3
34       cout << "s3 is \"" << s3 << "\"";
35    } // end if
36
37    // test overloaded string concatenation operator
38    cout << "\n\ns1 += s2 yields s1 = ";
39    s1 += s2; // test overloaded concatenation
40    cout << s1;
41
42    // test overloaded string concatenation operator with C-style string
43    cout << "\n\ns1 += \" to you\" yields" << endl;
44    s1 += " to you";
45    cout << "s1 = " << s1 << "\n\n";
46
47    // test string member function substr
48    cout << "The substring of s1 starting at location 0 for\n"
49       << "14 characters, s1.substr(0, 14), is:\n"
50       << s1.substr( 0, 14 ) << "\n\n";
51
52    // test substr "to-end-of-string" option
53    cout << "The substring of s1 starting at\n"
54       << "location 15, s1.substr(15), is:\n"
55       << s1.substr( 15 ) << endl;
56
57    // test copy constructor
58    string *s4Ptr = new string( s1 );
59    cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";
60
61    // test assignment (=) operator with self-assignment
62    cout << "assigning *s4Ptr to *s4Ptr" << endl;
63    *s4Ptr = *s4Ptr;
64    cout << "*s4Ptr = " << *s4Ptr << endl;
65
66    // test destructor
67    delete s4Ptr;
68
69    // test using subscript operator to create lvalue
70    s1[ 0 ] = 'H';
71    s1[ 6 ] = 'B';
72    cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
73       << s1 << "\n\n";
74
75    // test subscript out of range with string member function "at"
76    cout << "Attempt to assign 'd' to s1.at( 30 ) yields:" << endl;
77    s1.at( 30 ) = 'd'; // ERROR: subscript out of range
78    return 0;
79 } // end main
```

```
s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing s3.empty:
```

www.yourshop.4kia.ir

```
s3 is empty; assiging s1 to s3;
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
s1 = happy birthday to you

The substring of s1 starting at location 0 for
14 characters, s1.substr(0, 14), is:
happy birthday

The substring of s1 starting at
location 15, s1.substr(15), is:
to you

*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
*s4Ptr = happy birthday to you
Destructor: happy birthday to you

s1 after s1[0] = 'H'and s1[6] = 'B'is: Happy Birthday to you

Attempt to assign 'd' to s1[30] yields:
abnormal program termination
```

شکل ۱۵-۱۱ کلاس string از کتابخانه استاندارد.

خطوط 14-12 سه شی **string** ایجاد می‌کنند. **S1** با کلمه "happy"، **s2** با کلمه "birthday" و **s3** با استفاده از سازنده پیش‌فرض رشته به منظور ایجاد یک رشته تهی. مقداردهی اولیه می‌شوند. خطوط 18-17 این سه شی را با استفاده از **cout** و عملگر >>، که طراحان کلاس **string** آنرا برای کار با شی‌های رشته سربارگذاری کرده‌اند، چاپ می‌کنند. سپس خطوط 25-19 نتایج حاصله از مقایسه **s2** با **s1** را با استفاده از عملگرهای برابری و رابطه‌ای سربارگذاری شده کلاس **string**، بنمایش در می‌آورند.

کلاس **String** ما (شکل‌های ۹-۱۱ الی ۱۰-۱۱) یک عملگر **operator!** سربارگذاری شده تدارک دیده بود که مبادرت به تست یک رشته می‌کرد که آیا تهی است یا خیر. کلاس استاندارد **string** فاقد این قابلیت بعنوان یک عملگر سربارگذاری شده است و بجای آن دارای تابع عضو **empty** می‌باشد که در خط 30 از آن استفاده کرده‌ایم. اگر رشته تهی باشد، تابع **empty** مقدار **true** و در غیر اینصورت **false** برگشت خواهد داد.

خط 33 به توضیح عملگر تخصیص سربارگذاری شده کلاس **string** پرداخته که در آن **s1** به **s3** تخصیص می‌یابد. خط 34 برای نشان دادن اینکه عملیات تخصیص بدرستی صورت گرفته، **s3** را چاپ می‌کند.

خط 39 به توضیح عملگر += سربارگذاری شده کلاس **string** به منظور اتصال رشته پرداخته است. در این مورد، محتویات **s2** به **s1** مرتبط می‌شوند. سپس خط 40 نتیجه این رشته را که در **s1** ذخیره شده است، چاپ می‌کند.

www.yourshop.4kia.ir

کلاس **String** ما (شکل‌های ۹-۱۱ و ۱۰-۱۱) حاوی عملگر سربارگذاری شده **()operator** به منظور تهیه زیر رشته بود. کلاس استاندارد **string** فاقد چنین قابلیتی با عملگر سربارگذاری شده است و بجای آن دارای تابع عضو **substr** می‌باشد (خطوط 50 و 55). با فراخوانی **substr** در خط 50 یک زیر رشته 14 کاراکتری (آرگومان اول) از **s1** با موقعیت آغازین صفر (آرگومان دوم) بدست می‌آید. با فراخوانی **substr** در خط 55 یک زیر رشته از موقعیت آغازین 15 رشته **s1** تهیه می‌شود. زمانیکه آرگومان دوم مشخص نشود، **substr** مبادرت به برگشت دادن مابقی رشته از آنجایی خواهد کرد که فراخوانی گردیده است.

خط 58 بصورت دینامیکی یک شی **string** اخذ کرده و آنرا با کپی از **s1** مقداردهی اولیه می‌کند. نتیجه اینکار فراخوانی سازنده کپی کننده کلاس **string** است. خط 63 از عملگر سربارگذاری شده = کلاس **string** برای تست وضعیت خود تخصیصی استفاده کرده است.

خطوط 71-70 از عملگر سربارگذاری شده **[]** کلاس **string** به منظور ایجاد *lvalueها* استفاده کرده‌اند تا بتوان کاراکترهای جدید را با کاراکترهای موجود در **s1** جایگزین کرد. خط 73 مقدار جدید **s1** را چاپ می‌کند. در کلاس **String** ما، عملگر سربارگذاری شده **[]** وظیفه بررسی مرزها را انجام می‌داد تا مشخص شود که آیا شاخص دریافتی به عنوان یک آرگومان، یک شاخص معتبر است یا خیر. اگر شاخص معتبر نبود، عملگر یک پیغام خطا چاپ می‌کرد و برنامه خاتمه می‌پذیرفت. عملگر سربارگذاری شده **[]** کلاس استاندارد **string** عملیات بررسی مرزها را انجام می‌دهد. از اینرو بایستی برنامه‌نویس مطمئن گردد که عملیات استفاده کننده از عملگر سربارگذاری شده **[]** کلاس استاندارد **string** ناخواسته در عناصر خارج از مرزهای رشته، تغییری را حادث نسازد. کلاس استاندارد **string** با استفاده از تابع عضو خود بنام **at** بررسی مرزها را انجام می‌دهد و در صورتیکه شاخص معتبر نباشد، یک «استثنا به راه» می‌اندازد. در چنین وضعیتی، در حالت پیش‌فرض برنامه C++ خاتمه می‌پذیرد. اگر شاخص معتبر باشد، تابع **at** کاراکتر را از مکان تعیین شده بعنوان یک *lvalue* تغییرپذیر یا یک *lvalue* ثابت و با توجه به محتویات فراخوانی برگشت می‌دهد. خط 77 نشان می‌دهد که فراخوانی تابع **at** بر روی یک شاخص نامعتبر صورت گرفته است.

۱۴-۱۱ سازنده‌های صریح

در بخش‌های ۸-۱۱ و ۹-۱۱ در ارتباط با سازنده تک آرگومانی صحبت کردیم که می‌توانست توسط کامپایلر به منظور انجام یک تبدیل ضمنی بکار گرفته شود. عملیات بصورت اتوماتیک صورت می‌گیرد و برنامه‌نویس نیازی به استفاده از یک عملگر تبدیل کننده ندارد. در برخی از شرایط تبدیلات ضمنی مناسب

www.yourshop.4kia.ir

نبوده یا زمینه‌ساز خطا می‌شوند. برای مثال کلاس **Array** ما در شکل ۶-۱۱ سازنده‌ای تعریف کرده است که یک آرگومان از نوع **int** دیافت می‌کند. هدف از این سازنده ایجاد یک شی **Array** حاوی تعدادی عنصر تعیین شده توسط آرگومان **int** است. با این وجود، این سازنده می‌تواند توسط کامپایلر در انجام یک تبدیل ضمنی، درست بکار گرفته نشود.

استفاده تصادفی از یک سازنده تک آرگومانی بعنوان یک سازنده تبدیل کننده

برنامه شکل ۱۶-۱۱ از کلاس **Array** شکل‌های ۶-۱۱ و ۷-۱۱ به منظور نشان دادن یک تبدیل ضمنی غلط استفاده کرده است. خط 13 در **main** مبادرت به نمونه‌سازی شی **integers1** و فراخوانی سازنده تک آرگومانی به مقدار صحیح 7 که تعداد عناصر در آرایه را تعیین می‌کند، کرده است. از شکل ۷-۱۱ بخاطر دارید که سازنده **Array** یک آرگومان **int** برای مقداردهی اولیه تمام عناصر آرایه با صفر دریافت می‌کرد. خط 14 تابع **outputArray** (تعریف شده در خطوط 24-20) را که بعنوان آرگومان یک **const Array &** به یک **Array** دریافت می‌کند، فراخوانی می‌نماید. خروجی تابع، تعداد عناصر در آرگومان **Array** و محتویات آن است. در این مورد، سایز آرایه 7 است، و از اینرو هفت، صفر در خروجی قرار دارد.

خط 15 تابع **outputArray** را با مقدار 3 بعنوان آرگومان فراخوانی کرده است. با این وجود، این برنامه فاقد یک تابع بنام **outputArray** است که یک آرگومان صحیح (**int**) دیافت می‌کند. از اینرو کامپایلر تعیین می‌کند که آیا کلاس **Array** یک سازنده تبدیل کننده تدارک دیده است که بتواند یک **int** را به یک **Array** تبدیل کند یا خیر. هر سازنده‌ای که یک آرگومان منفرد دریافت کند بعنوان یک سازنده تبدیل کننده در نظر گرفته می‌شود و کامپایلر فرض می‌کند سازنده **Array** که یک **int** دریافت می‌نماید یک سازنده تبدیل کننده است و از آن برای تبدیل آرگومان 3 به یک شی **Array** موقت که حاوی سه عنصر است، استفاده می‌کند. سپس کامپایلر مبادرت به ارسال شی **Array** موقت به تابع **outputArray** می‌کند تا محتویات آنرا چاپ کند. بنابر این، حتی در زمانیکه بصورت صریح یک تابع **outputArray** تدارک ندیده باشیم که یک آرگومان **int** دریافت می‌کند، کامپایلر قادر به کامپایل خط 15 است. خروجی برنامه نمایشی از محتویات آرایه سه عنصری حاوی صفرها است.

```
1  // Fig. 11.16: Fig11_16.cpp
2  // Driver for simple class Array.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Array.h"
8
9  void outputArray( const Array & ); // prototype
10
11 int main()
12 {
```

www.yourshop.4kia.ir

```
13    Array integers1( 7 ); // 7-element array
14    outputArray( integers1 ); // output Array integers1
15    outputArray( 3 ); // convert 3 to an Array and output Array's contents
16    return 0;
17 }  // end main
18
19 // print Array contents
20 void outputArray( const Array &arrayToOutput )
21 {
22    cout << "The Array received has " << arrayToOutput.getSize()
23       << " elements. The contents are:\n" << arrayToOutput << endl;
24 } // end outputArray
```

```
The Array received has 7 elements. The contents are:
           0           0           0           0
           0           0           0

The Array received has 3 elements. The contents are:
           0           0           0
```

شکل ۱۶-۱۱ | سازنده‌های تک آرگومانی و تبدیلات ضمنی.

اجتناب از استفاده تصادفی از یک سازنده تک آرگومانی بعنوان یک سازنده تبدیل کننده

زبان C++ دارای کلمه کلیدی **explicit** به منظور متوقف کردن تبدیلات ضمنی از طریق سازنده‌های تبدیل کننده در مواردی است که اجازه انجام چنین تبدیلاتی وجود ندارد. سازنده‌ای که بصورت **explicit** اعلان شده است نمی‌تواند در یک تبدیل ضمنی بکار گرفته شود. در شکل ۱۷-۱۱ یک سازنده **explicit** در کلاس **Array** اعلان شده است. تنها تغییر اعمال شده در **Array.h** افزودن کلمه کلیدی **explicit** به اعلان سازنده تک آرگومانی در خط 15 است.

```
1  // Fig. 11.17: Array.h
2  // Array class for storing arrays of integers.
3  #ifndef ARRAY_H
4  #define ARRAY_H
5
6  #include <iostream>
7  using std::ostream;
8  using std::istream;
9
10 class Array
11 {
12    friend ostream &operator<<( ostream &, const Array & );
13    friend istream &operator>>( istream &, Array & );
14 public:
15    explicit Array( int = 10 ); // default constructor
16    Array( const Array & ); // copy constructor
17    ~Array(); // destructor
18    int getSize() const; // return size
19
20    const Array &operator=( const Array & ); // assignment operator
21    bool operator==( const Array & ) const; // equality operator
22
23    // inequality operator; returns opposite of == operator
24    bool operator!=( const Array &right ) const
25    {
26       return ! ( *this == right ); // invokes Array::operator==
27    } // end function operator!=
28
29    // subscript operator for non-const objects returns lvalue
30    int &operator[]( int );
31
32    // subscript operator for const objects returns rvalue
33    const int &operator[]( int ) const;
34 private:
```

www.yourshop.4kia.ir

```
35    int size; // pointer-based array size
36    int *ptr; // pointer to first element of pointer-based array
37 }; // end class Array
38
39 #endif
```

شکل ۱۷-۱۱ | تعریف کلاس Array با سازنده explicit.

نیازی به تغییر در فایل کد منبع حاوی تعریف تابع عضو کلاس **Array** نیست. برنامه شکل ۱۸-۱۱ نمایشی از نسخه اصلاح شده برنامه ۱۶-۱۱ است.

زمانیکه این برنامه کامپایل شود، کامپایلر یک پیغام خطا مبنی بر اینکه مقدار صحیحی به **outputArray** در خط 15 ارسال شده و نمی‌تواند به یک **const Array &** تبدیل شود، صادر می‌کند. پیغام خطای کامپایلر در پنجره خروجی نشان داده شده است. خط 61 نشان می‌دهد که چگونه می‌توان از سازنده **explicit** در ایجاد یک آرایه موقت از 3 عنصر و ارسال آن به تابع **outputArray** استفاده کرد.

خطای برنامه‌نویسی

*اقدام به احضار سازنده **explicit** برای یک تبدیل ضمنی، خطای کامپایل بدنبال خواهد داشت.*

خطای برنامه‌نویسی

*استفاده از **explicit** در کنار اعضای داده یا توابع عضو بجز سازنده تک آرگومانی خطای کامپایل است.*

```
1  // Fig. 11.18: Fig11_18.cpp
2  // Driver for simple class Array.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Array.h"
8
9  void outputArray( const Array & ); // prototype
10
11 int main()
12 {
13    Array integers1( 7 ); // 7-element array
14    outputArray( integers1 ); // output Array integers1
15    outputArray( 3 );// convert 3 to an Array and output Array's contents
16    outputArray( Array( 3 ) );//explicit single-argument constructor call
17    return 0;
18 }  // end main
19
20 // print array contents
21 void outputArray( const Array &arrayToOutput )
22 {
23    cout << "The Array received has " << arrayToOutput.getSize()
24       << " elements. The contents are:\n" << arrayToOutput << endl;
25 } // end outputArray
```

```
C:\cpphtp5_examples\ch11\Fig11_17_18\Fig11_18.cpp(15) : error C2664:
   'outputArray': cannot convert parameter 1 from 'int'to 'const Array &'
       Reason: cannot convert from 'int'to 'const Array'
       Constructor for class 'Array' is declared 'explicit'
```

شکل ۱۸-۱۱ | توصیف عملکرد سازنده explicit.

www.yourshop.4kia.ir