فصل
پانزدهم

استریم ورودی/خروجی

اهداف

- استفاده از استریم ورودی/خروجی شی‌گرا در C++.
- قالب‌بندی ورودی و خروجی.
- سلسله مراتب کلاس استریم I/O.
- استفاده از دستکاری کننده‌های استریم.
- کنترل ترازبندی و لایه‌گذاری.
- تعیین موفقیت‌آمیز بودن عملیات ورودی/خروجی.
- پیوند استریم ورودی به استریم خروجی.

رئوس مطالب

۱ -۱۵ مقدمه
۲-۱۵ استریم‌ها
۱-۲-۱۵ استریم‌های کلاسیک در مقابل استریم‌های استاندارد
۲-۲-۱۵ فایل‌های سرآیند کتابخانه iostream
۳-۲-۱۵ کلاس‌ها و شی‌های استریم ورودی/خروجی
۳-۱۵ استریم خروجی
۱-۳-۱۵ چاپ متغیرهای char *
۲-۳-۱۵ چاپ کاراکتر با استفاده از تابع عضو put
٤-۱۵ استریم ورودی
۱-٤-۱۵ توابع عضو get و getline
۲-٤-۱۵ توابع عضو peek، putback و ignore
۳-٤-۱۵ I/O نوع ایمن
٥-۱۵ ورودی/خروجی قالب‌بندی نشده با استفاده از read، write و gcount
٦-۱۵ معرفی دستکاری کننده‌های استریم
۱-٦-۱۵ پایه انتگرال استریم: dec، oct، hex و setbase
۲-٦-۱۵ دقت نقطه اعشار (setprecision, precision)
۳-٦-۱۵ طول میدان (setw, width)
٤-٦-۱۵ دستکاری کننده‌های استریم خروجی تعریف شده توسط کاربر
۷-۱۵ تعیین فرمت استریم و دستکاری کننده‌های استریم
۱-۷-۱۵ دنباله صفرها و نقاط دسیمال (showpoint)
۲-۷-۱۵ ترازبندی (left، right و internal)
۳-۷-۱۵ لایه‌گذاری (setfill, fill)
٤-۷-۱۵ پایه انتگرال استریم (showbase, hex, oct, dec)
٥-۷-۱۵ اعداد اعشاری، نماد علمی و ثابت (fixed ،scientific)
٦-۷-۱۵ کنترل حروف بزرگ/کوچک (uppercase)
۷-۷-۱۵ قالب‌بندی بولی (boolalpha)
۸-۷-۱۵ تنظیم و تنظیم مجدد وضعیت قالب‌بندی از طریق تابع عضو flags
۸-۱۵ وضعیت خطا در استریم
۹-۱۵ پیوند استریم خروجی با استریم ورودی

۱-۱۵ مقدمه

کتابخانه استاندارد C++ مجموعه وسیعی از قالبیت‌های ورودی/خروجی (I/O) را فراهم آورده است. در این فصل به بررسی قابلیت‌ها و توانایی عملیات I/O خواهیم پرداخت. C++ از I/O نوع ایمن (type-

(safe) استفاد می‌کند. هر عملیات I/O به نوع داده حساس می‌باشد. اگر یک تابع عضو I/O برای کار با نوع داده خاصی در نظر گرفته شده باشد، فقط برای آن نوع داده فراخوانی می‌شود. اگر مطابقتی مابین نوع داده واقعی و تابع برای کار با آن نوع داده وجود نداشته باشد، کامپایلر خطا تولید خواهد کرد. از اینرو داده اشتباه قادر به نفوذ به سیستم نخواهد بود.

کاربران می‌توانند نحوه عملکرد I/O بر روی شی‌ها از نوع تعریف شده توسط کاربر را با اعمال سربارگذاری عملگرهای درج (>>) و استخراج (<<) مشخص سازند. این بسط‌پذیری یکی از ویژگیهای با ارزش C++ است.

۱۵-۲ استریم‌ها

در C++ عملیات I/O از طریق *استریم‌ها* (*streams*) یا *جریان‌ها* صورت می‌گیرد، که دنباله یا توالی از بایت‌ها هستند. در عملیات ورودی، جریان بایت‌ها از سوی یک دستگاه (همانند صفحه کلید، دیسک، اتصال شبکه) به حافظه اصلی است. در عملیات خروجی، جریان بایت‌ها از طرف حافظه اصلی به سمت یک دستگاه می‌باشد (همانند، صفحه نمایش، چاپگر، دیسک، اتصال شبکه و غیره). یک برنامه کاربردی با مفهوم پیوندی این بایت‌ها سر و کار دارد. بایت‌ها می‌توانند نشاندهنده کاراکترها، داده‌های خام، تصاویر گرافیکی، گفتارهای دیجیتالی، ویدئو دیجیتالی یا هر نوع اطلاعات دیگری باشند که مورد نیاز یک برنامه هستند.

مکانیزم سیستم I/O بایستی بایت‌ها را از دستگاه‌ها، حافظه بطور پایدار و قابل اعتماد انتقال دهد (و برعکس). غالباً چنین انتقالی مستلزم برخی اعمال و حرکات مکانیکی نظیر چرخش دیسک یا نوار یا تایپ به وسیله صفحه‌کلید است. زمانی که این نوع انتقال‌ها صرف می‌کنند به نسبت زمان مورد نیاز پردازنده برای کار بر روی داده‌های داخلی بیشتر است. از اینرو عملیات I/O مستلزم طرح دقیق و بهینه شده است تا از کارایی مناسب برخوردار باشد.

زبان C++ هر دو قابلیت I/O در «سطح پایین» و «سطح بالا» را تدارک دیده است. قابلیت I/O در سطح پایین (یعنی I/O قالب‌بندی نشده) مشخص می‌کند که تعدادی از بایت‌ها بایستی از دستگاه به حافظه یا از حافظه به دستگاه منتقل شوند. در چنین انتقالی، خود بایت آیتم مورد نظر و علاقه است. در I/O سطح پایین، سرعت و حجم انتقال بالا است اما مناسب برای برنامه‌نویسان نمی‌باشد.

معمولاً برنامه‌نویسان ترجیح می‌دهند از I/O سطح بالا (یعنی I/O قالب‌بندی شده) استفاده کنند که در آن بایت‌ها در واحدهای با معنی دسته‌بندی می‌شوند، همانند اعداد صحیح، اعشاری، کاراکترها، رشته و نوع‌های تعریف شده توسط کاربر. این نوع از عملیات I/O بسیار رضایت بخش‌تر از پردازش فایل با حجم بالا است.

۱-۲-۱۵ استریم‌های کلاسیک در مقابل استریم‌های استاندارد

در گذشته، کتابخانه استریم کلاسیک C++ قادر به انجام ورودی و خروجی بر روی کاراکترها (chars) بود. بدلیل اینکه یک **char** یک بایت فضا اشغال می‌کرد، فقط می‌توانست محدود به عرضه مجموعه‌ای از کاراکترها باشد (همانند کاراکترهای جدول ASCII). با این وجود، بسیاری از زبان‌ها از الفبا استفاده می‌کنند که حاوی کاراکترهای بیشتری به نسبت کاراکترهای قابل عرضه توسط یک char یک بایتی است. مجموعه کاراکتر ASCII نمی‌تواند این کاراکترها را تدارک ببیند، در حالیکه مجموعه کاراکتری Unicode قادر به انجام اینکار است. مجموعه Unicode یک مجموعه کاراکتری بسط یافته بین‌المللی است که حاوی بسیاری از حروف زبان‌های زنده و پرکاربرد، نمادهای محاسباتی و غیره است که در جهان کاربرد دارند. برای کسب اطلاعات بیشتر در مورد Unicode می‌توانید به www.unicode.org مراجعه کنید.

زبان C++ حاوی کتابخانه استریم استاندارد است که به برنامه‌نویسان امکان ایجاد سیستم‌های با قابلیت کار با کاراکترهای Unicode را فراهم می‌آورد. به همین منظور، C++ شامل یک نوع کاراکتر اضافی بنام **wchar_t** است که می‌تواند کاراکترهای Unicode را ذخیره کند. همچنین C++ استاندارد برای کار با کلاس‌های استریم کلاسیک C++ مجدداً طراحی شده است.

۲-۲-۱۵ فایل‌های سرآیند کتابخانه iostream

کتابخانه **iostream** در C++ حاوی صدها گزینه در ارتباط با I/O است. چندین فایل سرآیند حاوی بخش‌های از واسط کتابخانه هستند. اکثر برنامه‌های C++ شامل فایل سرآیند <**iostream**> می‌باشند که سرویس‌های پایه را برای تمام عملیات I/O اعلان می‌کند. فایل سرآیند <**iostream**> تعریف کننده شی‌های **cin**، **count**، **cerr** و **clog** است، که متناظر با استریم ورودی استاندارد و استریم خروجی استاندارد، خطای استریم استاندارد بافر نشده و خطای استریم استاندارد بافر شده می‌باشند. این فایل سرویس‌های برای هر دو نوع I/O قالب‌بندی شده و نشده عرضه می‌کند.

سرآیند <**iomanip**> سرویس‌های سودمندی برای I/O قالب‌بندی شده که دستکاری کننده‌های استریم پارامتری شده نامیده می‌شوند، همانند **setw** و **setprecision** فراهم می‌آورد.

سرآیند <**fstream**> سرویس‌های برای پردازش فایل کنترل شده توسط کاربر فراهم می‌آورد. در فصل ۱۷ از این سرآیند در برنامه‌ها استفاده خواهیم کرد.

۳-۲-۱۵ کلاس و شی‌های استریم ورودی/خروجی

کتابخانه **iostream** الگوهای متعددی برای کار با عملیات رایج I/O فراهم می‌آورد. برای مثال، الگوی کلاس **basic_istream** از عملیات استریم ورودی، الگوی کلاس **basic_ostream** از عملیات استریم

خروجی، و الگوی کلاس **basic_iostream** از هر دو استریم ورودی و خروجی پشتیبانی می‌کند. هر الگو دارای یک الگوی تخصصی از پیش تعریف شده است که امکان char I/O را فراهم می‌آورد.

علاوه بر این کتابخانه **iostream** مجموعه‌ای از *typedefها* فراهم آورده است که اسامی مستعار برای این الگوهای تخصصی شده هستند. تصریح کننده **typedef** اسامی مستعار برای نوع داده‌های قبلاً تعریف شده فراهم می‌آورد. گاهی اوقات برنامه‌نویسان از **typedef** برای ایجاد اسامی کوتاهتر یا با معنی‌تر استفاده می‌کنند. برای مثال، عبارت

```
Typedef Card *CardPtr;
```

یک نوع اضافی بنام، **CardPtr** بعنوان یک نام مستعار برای نوع *** Card** تعریف می‌کند. توجه کنید که ایجاد یک نام با استفاده از **typedef** باعث ایجاد یک نوع داده نمی‌شود، **typedef** فقط یک نام برای نوعی که در برنامه بکار گرفته می‌شود، ایجاد می‌کند.

سلسله مراتب الگوی استریم I/O وسر بارگذاری عملگر

هر دو الگوی **basic_istream** و **basic_ostream** از طریق ارث‌بری مشترک از الگوی مبنا **basic_ios** کار می‌کنند. الگوی **basic_iostream** از طریق توارث مضاعف از الگوهای **basic_istream** و **basic_ostream** عمل می‌کند. دیاگرام کلاس در UML در شکل ۱-۱۵ آورده شده است، که روابط ارث‌بری را در میان آنها نشان می‌دهد.

سربارگذاری عملگر یک روش مناسب برای کار با I/O است. عملگر شیفت به چپ (<<) سربارگذاری شده مناسب استریم خروجی بوده از آن بعنوان عملگر درج استریم یاد می‌شود. عملگر شیفت به راست (<<) سربارگذاری شده مناسب برای استریم ورودی بوده و از آن بعنوان عملگر استخراج استریم یاد می‌شود. از این عملگرها به همراه شی‌های استریم استاندارد **cin**، **cout**، **cerr** و **clog** و گاهاً با شی‌های استریم تعریف شده از سوی کاربر استفاده می‌شود.

شکل ۱-۱۵ | بخشی از سلسله مراتب الگوی استریم I/O.

شی‌های استریم استاندارد cin، cout، cerr و clog

شی از قبل تعریف شده **cin** یک **istream** بوده و گفته می‌شود که "متصل به" (یا الصاق شده به) دستگاه ورودی استاندارد است که معمولاً صفحه کلید می‌باشد. عملگر استریم استخراج (<<)که در عبارت زیر بکار گرفته شده است سبب می‌شودتا مقدارمتغییر صحیح **grade** (با فرض اینکه **grade** قبلاً بعنوان یک متغیر **int** اعلان شده است) تا از طریق **cin** وارد حافظه گردد:

```
cin >> grade; //data "flows" in the direction of the arrows
```

توجه کنید که کامپایلر تعیین کننده نوع داده **grade** بوده و عملگر بارگذاری شده استریم استخراج را که مناسب است، انتخاب می‌کند. با فرض اینکه **grade** بدرستی اعلان شده باشد، عملگر استریم استخراج

مستلزم اطلاعات دیگری درباره نوع نمی‌باشد. عملگر << برای وارد کردن آیتم‌های داده از نوع توکار، رشته‌ها و اشاره‌گرها سربارگذاری شده است.

شی از پیش تعریف شده **cout** یک نمونه از **ostream** می‌باشد و گفته می‌شود که "متصل به" دستگاه خروجی استاندارد است که معمولاً صفحه نمایش است. عملگر >> که در عبارت زیر بکار گرفته شده است، سبب می‌شود تا مقدار متغیر **grade** از حافظه به دستگاه خروجی استاندارد منتقل شود:

```
cout << grade; //data "flows" in the direction of the arrows
```

همچنین توجه کنید که کامپایلر نوع داده **grade** را تعیین می‌کند (با فرض اینکه **grade** قبلاً بدرستی اعلان شده باشد) و عملگر >> انتخاب می‌شود و از اینرو این عملگر دیگر نیازی به اطلاعات اضافی در مورد نوع ندارد. عملگر >> باعث سربارگذاری آیتم‌های داده خروجی از نوع‌های توکار، رشته‌ها و مقادیر اشاره‌گر می‌شود.

شی از پیش تعریف شده **cerr** یک نمونه از **ostream** می‌باشد و گفته می‌شود که "متصل به" دستگاه خطای استاندارد می‌باشد. خروجی‌ها به شی **cerr** بافر نشده (*unbuffered*) هستند، و این مطلب را می‌رساند که هر گونه درج استریم **cerr** باعث می‌شود که خروجی آن بلافاصله ظاهر شود، این ویژگی روش مناسبی برای نشان دادن خطاها به کاربر است.

شی از پیش تعریف شده **clog** یک نمونه از کلاس **ostream** می‌باشد و گفته می‌شود که "متصل به" دستگاه خطای استاندارد است. خروجی‌ها به **clog** بافر شده (*buffered*) هستند. به این معنی که هر درجی به **clog** سبب می‌شود تا خروجی در یک بافر نگهداری شود تا زمانیکه بافر پر شود یا اینکه بافر خالی گردد. عملیات بافر کردن I/O از جمله تکنیک‌های است که در درس سیستم‌های عامل توضیح داده می‌شود.

الگوهای پردازش فایل

پردازش فایل در C++ با استفاده از الگوهای کلاس **basic_ifstream** (برای فایل ورودی)، **basic_ofstream** (برای فایل خروجی) و **basic_fstream** (برای فایل ورودی و خروجی) صورت می‌گیرد. هر الگوی کلاس دارای یک الگوی تخصصی شده از پیش تعریف شده است که امکان char I/O را فراهم می‌آورد. C++ مجموعه‌ای از typedefها تدارک است که اسامی مستعار برای این الگوهای تخصصی شده فراهم می‌آورد. برای مثال **typedef ifstream** نشاندهنده یک **basic_ifstream** تخصصی شده است که امکان می‌دهد **char** به یک فایل وارد گردد. همچنین **typedef fstream** نشاهنده یک **basic_fstream** است که امکان می‌دهد **char** از یک فایل خارج و به آن وارد گردد. الگوی **basic_ifstream** از **basic_iostream** ارث‌بری دارد. دیاگرام کلاس UML در شکل ۲-۱۵ آورده شده و روابط ارث‌بری در میان کلاس‌های مرتبط با I/O را نشان می‌دهد.

شکل ۲-۱۵ بخشی از سلسله مراتب الگوی استریم I/O در پردازش فایل.

۳-۱۵ استریم خروجی

خروجی قالب‌بندی شده و نشده توسط **ostream** تدارک دیده می‌شود. قابلیت‌های در نظر گرفته شده برای خروجی شامل انواع داده استاندارد با عملگر >>، چاپ کاراکترها از طریق تابع عضو **put**، خروجی قالب‌بندی نشده از طریق تابع عضو **write** (بخش ۵-۱۵)، چاپ مقادیر صحیح با فرمت‌های دسیمال (دهدهی)، اکتال و هگزادسیمال (بخش ۱-۶-۱۵)، چاپ مقادیر اعشاری با دقت‌های متفاوت (بخش ۲-۶-۱۵)، با توجه به نقطه اعشار (بخش ۱-۷-۱۵)، علامت‌گذاری علمی و ثابت (بخش ۵-۷-۱۵)، ترازبندی و تنظیم طول میدان داده (بخش ۲-۷-۱۵)، چاپ داده در لایه‌های مشخص شده (بخش ۷-۲-۱۵) و چاپ حروف بزرگ در علامت‌گذاری علمی و هگزادسیمال (بخش ۶-۷-۱۵) است.

۱-۳-۱۵ چاپ متغیرهای * char

++C از قابلیت اتوماتیک تعیین نوع داده به نسبت C برخوردار است. متاسفانه گاهی اوقات این ویژگی به اشتباه می‌رود. برای مثال فرض کنید، می‌خواهیم مقدار یک * **char** را با یک رشته کاراکتری چاپ کنیم (یعنی آدرس حافظه اولین کاراکتر از رشته) با این وجود، عملگر >> برای چاپ داده از نوع * **char** بعنوان یک رشته خاتمه یافته با **null** سربارگذاری شده است. راه حل این مشکل تبدیل * **char** به * **void** است. برنامه شکل ۳-۱۵ مبادرت به چاپ یک متغیر * **char** در هر دو فرمت رشته و آدرس کرده است. توجه کنید که آدرس بصورت یک عدد هگزادسیمال (برمبنای 16) چاپ شده است. در بخش‌های ۱-۶-۱۵، ۴-۷-۱۵، ۵-۷-۱۵ و ۷-۷-۱۵ با نحوه کنترل مبنای اعداد آشنا خواهید شد.

```
1  // Fig. 15.3: Fig15_03.cpp
2  // Printing the address stored in a char * variable.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     char *word = "again";
10
11    // display value of char *, then display value of char *
12    // static_cast to void *
13    cout << "Value of word is: " << word << endl
14       << "Value of static_cast< void * >( word ) is: "
15       << static_cast< void * >( word ) << endl;
16    return 0;
17 } // end main
```

```
Value of word is: again
Value of static_cast< void * >( word ) is: 00428300
```

شکل ۳-۱۵ | چاپ آدرس ذخیره شده در یک متغیر * char.

۲-۳-۱۵ چاپ کاراکتر با استفاده از تابع عضو put

می‌توانیم از تابع عضو **put** در چاپ کاراکترها استفاده کنیم. برای مثال، عبارت

```
cout.put( 'A' );
```

کاراکتر منفرد **A** را به نمایش در می‌آورد. فراخوانی تابع **put** می‌تواند بصورت پشت سرهم یا آبشاری باشد، همانند عبارت

```
cout.put( 'A' ).put( '\n' );
```

که حرف **A** و بدنبال آن یک کاراکتر خط جدید چاپ می‌شود. همانند عملگر >> در عبارت قبلی، چون عملگر نقطه (.) از سمت چپ به راست ارزیابی می‌شود، و تابع **put** یک مراجعه به شی **ostream** (یعنی **cout**) برگشت می‌دهد که فراخوانی **put** را دریافت می‌کند. همچنین امکان دارد تابع **put** با یک عبارت عددی که نشاندهنده یک مقدار ASCII است فراخوانی شود، همانند عبارت زیر

```
cout.put( 65 );
```

که **A** را چاپ می‌کند.

۱۵-٤ استریم ورودی

حال اجازه دهید تا نگاهی به استریم ورودی داشته باشیم. قابلیت ورودی قالب‌بندی شده و نشده توسط **istream** تدارک دیده شده است. عملگر << معمولاً کاراکترهای *white-space* (همانند فضاهای خالی، تب‌ها و خطوط جدید) را در استریم ورودی در نظر نمی‌گیرد. همانطوری که بعداً خواهید دید می‌توانیم این رفتار را تغییر دهیم. پس از هر ورودی، عملگر << یک مراجعه به شی استریم که پیغام استخراج را دریافت کرده است، برگشت می‌دهد (مثلاً، **cin** در عبارت **cin>>grade**). اگر آن مراجعه در یک شرط بکار گرفته شود (مثلاً در شرط تکرار حلقه **while**)، عملگر سربارگذاری شده **void *** بطور ضمنی احضار شده و مراجعه را تبدیل به یک مقدار اشاره‌گر غیر **null** یا اشاره‌گر **null** براساس موفقیت یا عدم موفقیت، آخرین عملیات ورودی می‌کند. اشاره‌گر غیر **null** به **true** تبدیل می‌شود تا نشاندهنده موفقیت‌آمیز بودن باشد و اشاره‌گر **null** به مقدار **false** تبدیل می‌شود تا دلالت بر عدم موفقیت کند. زمانیکه مبادرت به خواندن انتهای استریم می‌شود، عملگر تبدیل **void *** یک اشاره‌گر **null** برگشت می‌دهد تا نشاندهنده انتهای فایل باشد.

هر شی استریم حاوی مجموعه‌ای از بیت‌های وضعیت است که در کنترل وضعیت استریم نقش دارند (یعنی، قالب‌بندی، تنظیم خطا و غیره). این بیت‌ها توسط عملگر تبدیل **void *** برای تعیین اینکه آیا یک اشاره‌گر غیر **null** یا اشاره‌گر **null** برگشت داده شده است، استفاده می‌شود. اگر داده وارد شده از نوع اشتباه باشد، **failbit** تنظیم می‌شود و اگر عملیات با شکست مواجه شود، **badbit** تنظیم می‌گردد. در بخش‌های ۷-۱۵ و ۸-۱۵ به بررسی این بیت‌ها خواهیم پرداخت.

۱۵-٤-۱ توابع عضو get و getline

تابع عضو **get** بدون آرگومان، سبب می‌شود تا یک کاراکتر از استریم معین شده وارد گردد (شامل کاراکترهای *white-space* و سایر کاراکترهای غیرگرافیکی) و آنرا بعنوان مقداری از فراخوانی تابع برگشت می‌دهد. این نسخه از **get** زمانیکه به انتهای فایل در استریم برسد، **EOF** برگشت می‌دهد.

استفاده از توابع عضو eof، get و put

برنامه شکل ۴-۱۵ به توصیف نحوه استفاده از توابع عضو **eof** و **get** بر روی استریم ورودی **cin** و تابع عضو **put** بر روی استریم خروجی **cout** می‌پردازد. برنامه ابتدا مقدار **cin.eof()** را چاپ می‌کند. یعنی **false** (0 در خروجی)، تا نشان دهد که انتهای فایل در **cin** رخ نداده است. کاربر یک عبارت متنی وارد کرده و کلید *Enter* را بدنبال انتهای فایل فشار می‌دهد (*ctrl*>-z بر روی سیستم ویندوز مایکروسافت، *ctrl*>-d بر روی سیستم‌های لینوکس و مکنتاش). خط 17 هر کاراکتر را می‌خواند و خط 18 با استفاده از تابع عضو **put** آنرا در **cout** قرار می‌دهد. زمانیکه با انتهای فایل مواجه شود، عبارت **while** خاتمه یافته و خط 22 مقدار **cin.eof()** را که اکنون **true** است (1 در خروجی) چاپ می‌کند تا نشان دهد که انتهای فایل با **cin** تنظیم شده است. دقت کنید که در این برنامه از نسخه **get** متعلق به **istream** استفاده شده است که آرگومانی دریافت نمی‌کند و کاراکتر ورودی را برگشت می‌دهد (خط 17). تابع **eof** فقط پس از رسیدن برنامه یا اقدام به خواندن آخرین کاراکتر پس از استریم، **true** برگشت می‌دهد.

```
1  // Fig. 15.4: Fig15_04.cpp
2  // Using member functions get, put and eof.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    int character; // use int, because char cannot represent EOF
11
12    // prompt user to enter line of text
13    cout << "Before input, cin.eof() is " << cin.eof() << endl
14       << "Enter a sentence followed by end-of-file:" << endl;
15
16    // use get to read each character; use put to display it
17    while ( ( character = cin.get() ) != EOF )
18       cout.put( character );
19
20    // display end-of-file character
21    cout << "\nEOF in this system is: " << character << endl;
22    cout << "After input of EOF, cin.eof() is " << cin.eof() << endl;
23    return 0;
24 } // end main
```

```
Before input, cin.eof() is 0
Enter a sentence followed by end-of-file:
Testing the get and put member functions
Testing the get and put member functions
^z

EOF in this system is: -1
After input of EOF, cin.eof() is 1
```

شکل ٤-١٥ | توابع عضو get، put و eof.

تابع عضو **get** با یک آرگومان مراجعه–کاراکتر مبادرت به وارد کردن کاراکتر بعدی از استریم ورودی می‌کند (حتی اگر یک کاراکتر *white-space* باشد) و آن را در آرگومان کاراکتری ذخیره می‌سازد. این نسخه از **get** یک مراجعه به شی **istream** برگشت می‌دهد.

سومین نسخه از تابع **get** سه آرگومان دریافت می‌کند، یک آرایه کاراکتری، حد سایز و یک حائل (با مقدار پیش‌فرض **'\n'**). این نسخه کاراکترها را از استریم ورودی می‌خواند. تابع به تعداد یکی کمتر از حداکثر تعداد کاراکترهای تعیین شده را خوانده و یا بلافاصله پس از خواندن حائل بکار خاتمه می‌دهد. از یک کاراکتر **null** وارد شده به رشته ورودی در یک آرایه کاراکتری بعنوان بافر در برنامه استفاده می‌شود. حائل در آرایه کاراکتری جای نمی‌گیرد اما در استریم ورودی باقی می‌ماند. از اینرو، نتیجه دومین فراخوانی پی‌درپی **get** یک خط خالی است، مگر اینکه کاراکتر حائل از استریم ورودی حذف گردد (اینکار با **cin.ignore()** امکان‌پذیر است).

مقایسه cin و cin.get

برنامه شکل ۵–۱۵ به مقایسه ورودی با استفاده از **cin** و **cin.get** پرداخته است. دقت کنید که در فراخوانی **cin.get** (خط 24) حائلی مشخص نشده است، از اینرو کاراکتر پیش‌فرض **'\n'** بکار گرفته شده است.

```
1  // Fig. 15.5: Fig15_05.cpp
2  // Contrasting input of a string via cin and cin.get.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    // create two char arrays, each with 80 elements
11    const int SIZE = 80;
12    char buffer1[ SIZE ];
13    char buffer2[ SIZE ];
14
15    // use cin to input characters into buffer1
16    cout << "Enter a sentence:" << endl;
17    cin >> buffer1;
18
19    // display buffer1 contents
20    cout << "\nThe string read with cin was:" << endl
21       << buffer1 << endl << endl;
22
23    // use cin.get to input characters into buffer2
24    cin.get( buffer2, SIZE );
25
26    // display buffer2 contents
27    cout << "The string read with cin.get was:" << endl
28       << buffer2 << endl;
29    return 0;
30 } // end main
```

```
Enter a sentence:
Contrasting string input with cin and cin.get

The string read with cin was:
Contrasting

The string read with cin.get was:
 string input with cin and cin.get
```

شکل ٥-١٥ | وارد کردن رشته با استفاده از cin در مقایسه با cin.get

استفاده از تابع عضو getline

عملکرد تابع عضو **getline** شبیه سومین نسخه از تابع عضو **get** است و یک کاراکتر **null** پس از خط در آرایه کاراکتری وارد می‌کند. تابع **getline** مبادرت به حذف حائل از استریم می‌کند (یعنی کاراکتر را خواند و دور می‌اندازد)، اما آنرا در آرایه کاراکتری ذخیره نمی‌کند. برنامه شکل ۶-۱۵ به بررسی استفاده از تابع **getline** در وارد کردن یک خط متنی پرداخته است.

```
1  // Fig. 15.6: Fig15_06.cpp
2  // Inputting characters using cin member function getline.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    const int SIZE = 80;
11    char buffer[ SIZE ]; // create array of 80 characters
12
13    // input characters in buffer via cin function getline
14    cout << "Enter a sentence:" << endl;
15    cin.getline( buffer, SIZE );
16
17    // display buffer contents
18    cout << "\nThe sentence entered is:" << endl << buffer << endl;
19    return 0;
20 } // end main
```

```
Enter a sentence:
Using the getline member function

The sentence entered is:
Using the getline member function
```

شکل ٦-١٥ وارد کردن داده کاراکتری با cin تابع عضو getline.

۲-٤-١٥ توابع عضو peek، putback و ignore

تابع عضو **ignore** از **istream** قادر به خواندن و دور انداختن کاراکترها به تعداد مشخص شده (مقدار پیش‌فرض یک کاراکتر است) یا خاتمه دادن به خواندن در مواجه شدن با حائل (حائل پیش‌فرض **EOF** می‌باشد که سبب می‌شود **ignore** به انتهای فایل پرش کند) می‌گردد.

تابع عضو **putback** کاراکتر قبلی بدست آمده از **get** از استریم ورودی را به استریم باز می‌گرداند. این تابع مناسب برای برنامه‌های است که یک استریم ورودی را اسکن می‌کنند تا به انتهای یک فیلد با کاراکتر مشخص برسند. زمانیکه این کاراکتر وارد شد، برنامه کاراکتر را به استریم باز می‌گرداند، از اینرو کاراکتر می‌تواند در داده ورودی لحاظ شود.

تابع عضو **peek** کاراکتر بعدی از یک استریم ورودی را برگشت می‌دهد، اما کاراکتر را از استریم حذف نمی‌کند.

۳-٤-١٥ I/O نوع ایمن

زبان ++C ارائه کننده I/O از نوع ایمن است. عملگرهای >> و << سربارگذاری شده تا آیتم‌های داده از نوع مشخص را پذیرا باشند. اگر داده غیرمنتظره‌ای پردازش شود، بیت‌های مختلفی تنظیم می‌شوند که کاربر می‌تواند با بررسی آنها به وضعیت عملیات I/O پی ببرد.

۱۵-۵ ورودی/خروجی قالب‌بندی نشده با استفاده از read، write و gcout

ورودی/خروجی قالب‌بندی نشده با استفاده از توابع عضو **read** و **write** از **istream** و **ostream** صورت می‌گیرد. تابع عضو **read** تعدادی از بایت‌ها را به آرایه کاراکتری در حافظه وارد می‌سازد، تابع عضو **write** بایت‌های از آرایه کاراکتری خارج می‌کند.

این بایت‌ها قالب‌بندی شده نیستند. آنها بصورت بایت‌های خام وارد یا خارج می‌شوند. برای مثال در فراخوانی

```
char buffer[] = "HAPPY BIRTHDAY";
cout.write( buffer, 10 );
```

ده بایت اول از **buffer** چاپ می‌شود. فراخوانی

```
cout.write( "ABCDEFGHIJKLMNOPQRSTUVWZYZ", 10 );
```

ده کاراکتر اول از الفبا را نشان می‌دهد.

تابع عضو **read** به تعداد مشخص شده‌ای از کاراکترها را به یک آرایه کاراکتری می‌خواند. اگر کمتر از تعداد مشخص شده، کاراکتر قرائت شود، **failbit** تنظیم خواهد شد. در بخش ۸-۱۵ با نحوه تعیین و تنظیم وضعیت **failbiy** آشنا خواهید شد. تابع عضو **gcount** گزارشی از کاراکترهای خوانده شده توسط آخرین عملیات ورودی تهیه می‌کند.

برنامه شکل ۷-۱۵ به بررسی عملکرد توابع عضو **read** و **gcount** از **isream** و تابع عضو **write** از **ostream** پرداخته است. برنامه **20** کاراکتر دریافت و در آرایه کاراکتری **buffer** با دستور **read** قرار می‌دهد (خط 15)، تعداد کاراکترهای ورودی توسط **gcount** تعیین می‌شود (خط 19) و کاراکترهای موجود در **burffer** توسط دستور **write** چاپ می‌شوند. (خط 19).

```
1  // Fig. 15.7: Fig15_07.cpp
2  // Unformatted I/O using read, gcount and write.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    const int SIZE = 80;
11    char buffer[ SIZE ]; // create array of 80 characters
12
13    // use function read to input characters into buffer
14    cout << "Enter a sentence:" << endl;
15    cin.read( buffer, 20 );
16
17    // use functions write and gcount to display buffer characters
18    cout << endl << "The sentence entered was:" << endl;
19    cout.write( buffer, cin.gcount() );
20    cout << endl;
```

```
21    return 0;
22 } // end main
```

```
Enter a sentence:
Using the read, write, and gcount member functions
The sentence enterd was:
Using the read, write
```

شکل ۷-۱۵ I/O قالب‌بندی نشده با استفاده از توابع read، gcount و wtire.

۶-۱۵ معرفی دستکاری کننده‌های استریم

زبان C++ تعداد متنابهی دستکاری کننده استریم در نظر گرفته است که وظایف قالب‌بندی را برعهده دارند. این وظایف عبارتند از تنظیم طول میدان، تنظیمات مربوط به دقت، تنظیم وضعیت قالب‌بندی، تنظیم کاراکتر پرکننده در میدان، دور ریختن استریم‌ها، وارد کردن خط جدید به استریم ورودی (و حذف خطی از استریم)، وارد کردن کاراکتر null به استریم خروجی و در نظر نگرفتن کاراکترهای *white-space* در استریم ورودی. این ویژگیهای در بخش‌های زیر توضیح داده می‌شوند.

۱-۶-۱۵ پایه انتگرال استریم: dec، oct، hex و setbase

معمولاً اعداد صحیح بعنوان مقادیر دسیمال (پایه **10**) تفسیر می‌شوند. برای تغییر پایه و تفسیر مقادیر موجود در استریم، دستکاری کننده **hex** را برای تنظیم پایه به هگزادسیمال (پایه **16**) یا وارد کردن **oct** برای تنظیم پایه به اکتال (پایه **8**) بکار می‌گیرند. با وارد کردن دستکاری کننده **dec** به استریم پایه به حالت دسیمال باز می‌گردد.

همچنین می‌توان با استفاده از دستکاری کننده استریم **setbase** مبادرت به تغییر پایه کرده که یک آرگومان از 10، 8 یا 16 دریافت می‌کند تا به ترتیب تنظیم کننده پایه به دسیمال، اکتال یا هگزادسیمال باشد. استفاده از **setbase** (یا هر نوع دستکاری کننده پارامتری) مستلزم استفاده از فایل سرآیند <**iomanip**> است. مقادیر پایه استریم تا زمانیکه تغییری در آن داده نشود، وضعیت خود را حفظ می‌کند. برنامه شکل ۸-۱۵ به بررسی **hex**، **oct**، **dec** و **setbase** پرداخته است.

```
1  // Fig. 15.8: Fig15_08.cpp
2  // Using stream manipulators hex, oct, dec and setbase.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::dec;
7  using std::endl;
8  using std::hex;
9  using std::oct;
10
11 #include <iomanip>
12 using std::setbase;
13
14 int main()
15 {
16    int number;
17
18    cout << "Enter a decimal number: ";
19    cin >> number; // input number
20
21    // use hex stream manipulator to show hexadecimal number
22    cout << number << " in hexadecimal is: " << hex
23       << number << endl;
```

```
24
25    // use oct stream manipulator to show octal number
26    cout << dec << number << " in octal is: "
27       << oct << number << endl;
28
29    // use setbase stream manipulator to show decimal number
30    cout << setbase( 10 ) << number << " in decimal is: "
31       << number << endl;
32    return 0;
33 } // end main
```

```
Enter a decimal number: 20
20 in hexadecimal is: 14
20 in octal is: 24
20 in decimal is: 20
```

شکل ۸-۱۵ دستکاری کننده‌های استریم hex، oct، dec و setbase.

۲-٦-۱۵ دقت نقطه اعشار (setprecision,precision)

می‌توانیم با استفاده از دستکاری کننده **setprecision** یا تابع عضو **precision** از **io_base** مبادرت به تنظیم دقت اعداد اعشاری کنیم (یعنی تعداد ارقام در سمت راست نقطه اعشار). این تنظیم صورت گرفته بر روی تمام خروجی‌ها اعمال می‌شود تا اینکه تنظیم دیگری در دقت اعمال گردد. فراخوانی تابع عضو **precision** بدون آرگومان، دقت جاری را برگشت می‌دهد. برنامه شکل ۹-۱۵ از تابع عضو **precision** در خط 28 و دستکاری کننده **setprecsion** در خط 37 برای چاپ یک جدول که نشاندهنده ریشه دوم عدد 2 با دقت اعمال شده از 9-0 استفاده کرده است.

```
1  // Fig. 15.9: Fig15_09.cpp
2  // Controlling precision of floating-point values.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::fixed;
7
8  #include <iomanip>
9  using std::setprecision;
10
11 #include <cmath>
12 using std::sqrt; // sqrt prototype
13
14 int main()
15 {
16    double root2 = sqrt( 2.0 ); // calculate square root of 2
17    int places; // precision, vary from 0-9
18
19    cout << "Square root of 2 with precisions 0-9." << endl
20       << "Precision set by ios_base member function "
21       << "precision:" << endl;
22
23    cout << fixed; // use fixed point format
24
25    // display square root using ios_base function precision
26    for ( places = 0; places <= 9; places++ )
27    {
28       cout.precision( places );
29       cout << root2 << endl;
30    } // end for
31
32    cout << "\nPrecision set by stream manipulator "
33       << "setprecision:" << endl;
34
35    // set precision for each digit, then display square root
36    for ( places = 0; places <= 9; places++ )
37       cout << setprecision( places ) << root2 << endl;
```

```
38
39    return 0;
40 } // end main
```

```
Square root of 2 with precisions 0-9.
Precision set by iso_base member function precision:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562

Precision set by stream manipulator setprecision:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

شکل ۹-۱۵ | دقت مقادیر اعشاری.

۳-٦-۱۵ طول میدان (width و setw)

تابع عضو **width** (از کلاس مبنای **iso_base**) مبادرت به تنظیم طول میدان می‌کند و طول میدان قبلی را برگشت می‌دهد. اگر مقادیر خروجی کمتر از طول میدان باشند، کاراکترهای پرکننده بعنوان لایه (padding) بکار گرفته می‌شوند. مقدار بزرگتر از پهنای در نظر گرفته شده قطع نمی‌شود، کل عدد چاپ می‌شود. تابع **width** بدون آرگومان، تنظیم جاری را برگشت می‌دهد.

در برنامه شکل ۱۰-۱۵ به بررسی نحوه استفاده از تابع عضو **width** هم بر روی ورودی و هم خروجی پرداخته شده است. همچنین از دستکاری کننده **setw** برای تنظیم طول میدان استفاده می‌شود.

```
1  // Fig. 15.10: Fig15_10.cpp
2  // Demonstrating member function width.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    int widthValue = 4;
11    char sentence[ 10 ];
12
13    cout << "Enter a sentence:" << endl;
14    cin.width( 5 ); // input only 5 characters from sentence
15
16    // set field width, then display characters based on that width
17    while ( cin >> sentence )
18    {
19       cout.width( widthValue++ );
20       cout << sentence << endl;
21       cin.width( 5 ); // input 5 more characters from sentence
22    } // end while
23
24    return 0;
```

```
25 } // end main
```

```
Enter a sentence:
This is a test of the width member function
This
   is
     a
   test
      of
      the
      widt
          h
        memb
           er
          func
           tion
```

شکل ۱۰-۱۵ تابع عضو width از کلاس iso_base.

٤-٦-۱۵ دستکاری کننده‌های استریم خروجی تعریف شده توسط کاربر

برنامه‌نویسان می‌تواند دستکاری کننده‌های استریم متعلق بخود را ایجاد کنند. در برنامه شکل ۱۱-۱۵ ایجاد و استفاده از دستکاری کننده‌های استریم غیر پارامتری و جدید **bell** (خطوط 13-10)، **carriageReturn** (خطوط 19-16)، **tab** (خطوط 25-22) و **endLine** (خطوط 32-29) نشان داده شده است. برای دستکاری کننده‌های استریم خروجی، نوع برگشتی و پارامتر بایستی از نوع **ostream &** باشد. زمانیکه خط 37 مبادرت به وارد کردن **endLine** در استریم خروجی می‌کند، تابع **endLine** فراخوانی شده و خط 31 دنباله **\n** را و دستکاری کننده **flush** را بر روی استریم استاندارد **cout** خارج می‌سازد. به همین ترتیب، زمانیکه خطوط 46-37 دستکاری کننده‌های **tab**، **bell** و **carrigeReturn** را وارد استریم خروجی می‌سازند، توابع متناظر **tab** (خط 22)، **bell** (خط 10) و **carriageReturn** (خط 16) فراخوانی می‌شوند.

```
1  // Fig. 15.11: Fig15_11.cpp
2  // Creating and testing user-defined, nonparameterized
3  // stream manipulators.
4  #include <iostream>
5  using std::ostream;
6  using std::cout;
7  using std::flush;
8
9  // bell manipulator (using escape sequence \a)
10 ostream& bell( ostream& output )
11 {
12    return output << '\a';  // issue system beep
13 } // end bell manipulator
14
15 // carriageReturn manipulator (using escape sequence \r)
16 ostream& carriageReturn( ostream& output )
17 {
18    return output << '\r';  // issue carriage return
19 } // end carriageReturn manipulator
20
21 // tab manipulator (using escape sequence \t)
22 ostream& tab( ostream& output )
23 {
24    return output << '\t';  // issue tab
25 } // end tab manipulator
26
27 // endLine manipulator (using escape sequence \n and member
28 // function flush)
29 ostream& endLine( ostream& output )
30 {
31    return output << '\n' << flush; // issue endl-like end of line
```

```
32 } // end endLine manipulator
33
34 int main()
35 {
36    // use tab and endLine manipulators
37    cout << "Testing the tab manipulator:" << endLine
38       << 'a' << tab << 'b' << tab << 'c' << endLine;
39
40    cout << "Testing the carriageReturn and bell manipulators:"
41       << endLine << ".........";
42
43    cout << bell; // use bell manipulator
44
45    // use ret and endLine manipulators
46    cout << carriageReturn << "-----" << endLine;
47    return 0;
48 } // end main
```

```
Testing the tab manipulator:
a       b       c
Testing the carriageReturn and bell manipulators:
-----.....
```

شکل ۱۱-۱۵ | دستکاری کننده‌های استریم غیر پارامتری تعریف شده توسط کاربر.

۷-۱۵ تعیین فرمت استریم و دستکاری کننده‌های استریم

می‌توان از انواع دستکاری کننده‌های استریم به منظور تصریح نوع قالب‌بندی در حین عملیات I/O استفاده کرد. دستکاری کننده‌های استریم بر قالب‌بندی خروجی کنترل دارند. در جدول شکل ۱۲-۱۵ دستکاری کننده‌های استریم که بر قالب استریم کنترل دارند، لیست شده‌اند. تمام این دستکاری کننده‌ها متعلق به کلاس **ios_base** می‌باشند. در بخش‌های بعدی از این دستکاری کننده‌ها در مثال‌ها استفاده کرده‌ایم.

دستکاری کننده	توضیح
skipws	از کاراکترهای *white-space* در استریم ورودی صرفنظر (پرش) می‌کند. اینحالت با استفاده از **noskipws** از کار می‌افتد.
left	خروجی را از سمت چپ تنظیم (تراز) می‌کند. کاراکترهای لایه‌گذاری در صورت نیاز در سمت راست ظاهر می‌شوند.
right	خروجی را از سمت راست تنظیم (تراز) می‌کند. کاراکترهای پایه لایه‌گذاری در صورت نیاز در سمت چپ ظاهر می‌شوند.
internal	نشان می‌دهد که علامت عدد باید از سمت چپ و بزرگی عدد باید از سمت راست تنظیم شود.
dec	مشخص می‌کند که با مقادیر صحیح همانند مقادیر دسیمال (پایه **10**) رفتار شود.
oct	مشخص می‌کند که با مقادیر صحیح همانند مقادیر اکتال (پایه **8**) رفتار شود.
hex	مشخص می‌کند که با مقادر صحیح همانند مقادیر هگزادسیمال (پایه **16**) رفتار شود.
showbase	مشخص می‌کند که پایه یک عدد در کنار آن در خروجی قرار گیرد (**0** برای اکتال، **0x** برای هگزادسیمال). این تنظیم با استفاده از **noshowbase** از کار می‌افتد.
showpoint	مشخص می‌کند که اعداد اعشاری بایستی با نقطه دسیمال چاپ شوند. معمولاً این دستکاری کننده با **fixed** بکار گرفته می‌شود تا تضمینی برای تعداد ارقام مشخص شده در سمت راست نقطه دسیمال باشد، حتی اگر آنها صفر باشند.

uppercase	مشخص می‌کند که باید ازحروف بزرگ (یعنی **X** و **A** تا **F**) در یک عدد صحیح هگزادسیمال و حروف بزرگ **E** در نمایش یک مقدار اعشاری در نماد علمی بکار گرفته شود.
showpos	مشخص می‌کند که اعداد مثبت بایستی همراه با نماد جمع (+) ظاهر شوند.
scientific	خروجی با نمایش علمی ظاهر می‌گردد.
fixed	تعیین کننده نقطه ثابت در یک مقدار اعشاری به تعداد ارقام مشخص شده در سمت راست نقطه دسیمال است.

جدول ۱۲-۱۵ | دستکاری کننده‌های وضعیت قالب‌بندی از <iosream>.

۱-۷-۱۵ دنباله صفرها و نقاط دسیمال (showpoint)

دستکاری کننده **showpoint** یک عدد اعشاری را مجبور می‌کند تا در خروجی با نقطه دسیمال و دنباله‌ای از صفرهای تعیین شده ظاهر شود. برای مثال، مقدار اعشاری 79.0 بدون استفاده از **showpoint** بصورت 79 و به هنگام استفاده از **showpoint** بصورت 79.000000 (یا تعداد بیشتری از صفرهای دنباله که با دقت جاری مشخص می‌شوند) ظاهر می‌شود. نقطه مقابل **showpoint** دستکاری کننده **noshowpoint** است. برنامه موجود در شکل ۱۳-۱۵ نحوه استفاده از **showpoint** در کنترل چاپ دنباله‌ای از صفرها و نقاط دسیمال در مقادیر اعشاری را نشان می‌دهد. بخاطر داشته باشید که دقت پیش‌فرض برای یک عدد اعشاری، 6 است. زمانیکه از **fixed** یا **scientific** استفاده نشده باشد، دقت نمایش، تعداد ارقام بامعنی است، و نه تعداد ارقام پس از نقطه دسیمال.

```
// Fig. 15.13: Fig15_13.cpp
// Using showpoint to control the printing of
// trailing zeros and decimal points for doubles.
#include <iostream>
using std::cout;
using std::endl;
using std::showpoint;

int main()
{
   // display double values with default stream format
   cout << "Before using showpoint" << endl
      << "9.9900 prints as: " << 9.9900 << endl
      << "9.9000 prints as: " << 9.9000 << endl
      << "9.0000 prints as: " << 9.0000 << endl << endl;

   // display double value after showpoint
   cout << showpoint
      << "After using showpoint" << endl
      << "9.9900 prints as: " << 9.9900 << endl
      << "9.9000 prints as: " << 9.9000 << endl
      << "9.0000 prints as: " << 9.0000 << endl;
   return 0;
} // end main
```

```
Before using showpoint
9.9900 prints as: 9.99
9.9000 prints as: 9.9
9.0000 prints as: 9

After using showpoint
9.9900 prints as: 9.99000
9.9000 prints as: 9.90000
9.0000 prints as: 9.00000
```

شکل ۱۳-۱۵ | کنترل چاپ دنباله‌ای از صفرها و نقاط دسیمال در مقادیر اعشاری.

۲-۷-۱۵ ترازبندی (left، right و internal)

دستکاری کننده‌های **left** و **right** به فیلدها امکان می‌دهند تا به ترتیب از سمت چپ با کاراکترهای لایه‌گذاری به سمت راست و از سمت راست با کاراکترهای لایه‌گذاری به سمت چپ تراز شوند. کاراکتر لایه‌گذاری توسط تابع عضو **fill** یا دستکاری کننده استریم پارامتری **setfill** مشخص می‌شود. در برنامه شکل ۱۴-۱۵ از دستکاری کننده‌های **setw**، **left** و **right** برای ترازبندی چپ و راست داده صحیح در یک فیلد استفاده شده است.

```
1  // Fig. 15.14: Fig15_14.cpp
2  // Demonstrating left justification and right justification.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::left;
7  using std::right;
8
9  #include <iomanip>
10 using std::setw;
11
12 int main()
13 {
14    int x = 12345;
15
16    // display x right justified (default)
17    cout << "Default is right justified:" << endl
18       << setw( 10 ) << x;
19
20    // use left manipulator to display x left justified
21    cout << "\n\nUse std::left to left justify x:\n"
22       << left << setw( 10 ) << x;
23
24    // use right manipulator to display x right justified
25    cout << "\n\nUse std::right to right justify x:\n"
26       << right << setw( 10 ) << x << endl;
27    return 0;
28 } // end main
```

```
Default is right justified:
     12345

Use std::left to left justify x:
12345

Use std::right to right justify x:
     12345
```

شکل ۱۴-۱۵ | ترازبندی چپ و راست با left و right.

دستکاری کننده استریم **internal** بر این نکته دلالت دارد که علامت عدد (یا پایه به هنگام استفاده از **showbase**) باید در سمت چپ فیلد تراز شود، و خود عدد از سمت راست تراز گردد و فاصله مناسب با لایه‌گذاری کاراکتر پرکننده حفظ گردد. برنامه شکل ۱۵-۱۵ نحوه استفاده از دستکاری کننده استریم **internal** با فاصله‌گذاری مشخص (خط 15) را نشان می‌دهد. توجه کنید که **showpos** نماد جمع را چاپ می‌کند (خط 15). برای غیرفعال کردن **showpos** می‌توان از **noshowpos** استفاده کرد.

```
1  // Fig. 15.15: Fig15_15.cpp
2  // Printing an integer with internal spacing and plus sign.
3  #include <iostream>
```

```
4  using std::cout;
5  using std::endl;
6  using std::internal;
7  using std::showpos;
8
9  #include <iomanip>
10 using std::setw;
11
12 int main()
13 {
14    // display value with internal spacing and plus sign
15    cout << internal << showpos << setw( 10 ) << 123 << endl;
16    return 0;
17 } // end main
```

```
+       123
```

شکل ۱۵-۱۵ | چاپ یک عدد صحیح با فاصله‌گذاری داخلی و نماد جمع.

۳-۷-۱۵ لایه‌گذاری (fill، setfill)

تابع عضو **fill** تعیین کننده کاراکتر پرکننده در ترازبندی فیلدها است، اگر هیچ مقداری مشخص نشود، برای لایه‌گذاری از فاصله‌ها استفاده خواهد شد. تابع **fill** کاراکتر لایه‌گذاری قبلی را برگشت می‌دهد. همچنین دستکاری کننده **setfill** مبادرت به تنظیم کاراکتر لایه‌گذاری می‌کند. در برنامه شکل ۱۶-۱۵ به بررسی نحوه استفاده از تابع عضو **fill** در خط 40 و دستکاری کننده استریم **setfill** در خطوط 47 و 44 پرداخته است.

```
1  // Fig. 15.16: Fig15_16.cpp
2  // Using member-function fill and stream-manipulator setfill to change
3  // the padding character for fields larger than the printed value.
4  #include <iostream>
5  using std::cout;
6  using std::dec;
7  using std::endl;
8  using std::hex;
9  using std::internal;
10 using std::left;
11 using std::right;
12 using std::showbase;
13
14 #include <iomanip>
15 using std::setfill;
16 using std::setw;
17
18 int main()
19 {
20    int x = 10000;
21
22    // display x
23    cout << x << " printed as int right and left justified\n"
24       << "and as hex with internal justification.\n"
25       << "Using the default pad character (space):" << endl;
26
27    // display x with base
28    cout << showbase << setw( 10 ) << x << endl;
29
30    // display x with left justification
31    cout << left << setw( 10 ) << x << endl;
32
33    // display x as hex with internal justification
34    cout << internal << setw( 10 ) << hex << x << endl << endl;
35
36    cout << "Using various padding characters:" << endl;
37
38    // display x using padded characters (right justification)
39    cout << right;
```

```
40    cout.fill( '*' );
41    cout << setw( 10 ) << dec << x << endl;
42
43    // display x using padded characters (left justification)
44    cout << left << setw( 10 ) << setfill( '%' ) << x << endl;
45
46    // display x using padded characters (internal justification)
47    cout << internal << setw( 10 ) << setfill( '^' ) << hex
48       << x << endl;
49    return 0;
50 } // end main
```

```
10000 printed as int right and left justified
and as hex with internal justification.
Using the default pad character (space):
     10000
10000
0x    2710

Using various padding characters:
*****10000
10000%%%%%
0x^^^^2710
```

شکل ١٦-١٥ | استفاده از تابع fill و دستکاری کننده setfill.

٤-٧-١٥ پایه انتگرال استریم (dec، oct، hex، showbase)

زبان C++ دستکاری کننده‌های استریم **dec**، **hex** و **oct** را برای تصریح اینکه مقادیر صحیح بصورت دسیمال، هگزادسیمال و اکتال به نمایش درآیند، در نظر گرفته است. اگر از هیچ یک از این دستکاری کننده‌ها استفاده نشود، حالت دسیمال بکار گرفته خواهد شد. از زمانیکه یک پایه خاص برای استریم تعیین شد، تمام مقادیر صحیح در آن استریم با آن پایه پردازش خواهند شد تا اینکه این پایه تغییر داده شود. در خروجی استریم با مقادیر صحیح با پسوند صفر همانند مقادیر اکتال، مقادیر پسوند صحیح با **0x** یا **0X** همانند مقادیر هگزادسیمال و تمام مقادیر صحیح همانند مقادیر دسیمال رفتار خواهد شد.

دستکاری کننده استریم **showbase** سبب می‌شود تا مبنای یک مقدار در خروجی ظاهر شود. اعداد دسیمال بطور عادی در خروجی دیده می‌شوند، اعداد اکتال با یک دنباله صفر، و اعداد هگزادسیمال با دنباله‌ای از **0x** یا **0X**. در برنامه شکل ١٧-١٥ به بررسی نحوه استفاده از **showbase** پرداخته شده است. برای از کار انداختن **showbase** می‌توانید از **noshowbase** استفاده کنید.

```
1  // Fig. 15.17: Fig15_17.cpp
2  // Using stream-manipulator showbase.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::hex;
7  using std::oct;
8  using std::showbase;
9
10 int main()
11 {
12    int x = 100;
13
14    // use showbase to show number base
15    cout << "Printing integers preceded by their base:" << endl
16       << showbase;
17
18    cout << x << endl; // print decimal value
19    cout << oct << x << endl; // print octal value
```

```
20    cout << hex << x << endl; // print hexadecimal value
21    return 0;
22 } // end main
```

```
Printing integers preceded by their base:
100
0144
0x64
```

شکل ۱۷-۱۵ دستکاری کننده استریم showbase.

۵-۷-۱۵ اعداد اعشاری، نماد علمی و ثابت (fixed, scientific)

دستکاری کننده‌های استریم **scientific** و **fixed** بر روی فرمت یا قالب‌بندی اعداد اعشاری کنترل دارند. دستکاری کننده **scientific** خروجی یک عدد اعشاری را مجبور می‌کند تا با فرمت علمی بنمایش درآید. دستکاری کننده **fixed** نیز یک عدد اعشاری را مجبور می‌کند تا به تعداد مشخص شده‌ای از ارقام در سمت راست نقطه اعشار بـه نماش درآورد. بدون استفاده از دستکاری کننده دیگری، مقدار یک عدد اعشاری تعیین کننده فرمت خروجی خواهد بود.

در برنامه شکل ۱۸-۱۵ یک عدد اعشاری در هر دو قالب ثابت و علمی با استفاده از **field** و **scientific** در خطوط 21 و 25 نشان داده شده است. امکان دارد فرمت توان در نماد علمی در میان کامپایلرهای مختلف با یکدیگر تفاوت داشته باشد.

```
1  // Fig. 15.18: Fig15_18.cpp
2  // Displaying floating-point values in system default,
3  // scientific and fixed formats.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7  using std::fixed;
8  using std::scientific;
9
10 int main()
11 {
12    double x = 0.001234567;
13    double y = 1.946e9;
14
15    // display x and y in default format
16    cout << "Displayed in default format:" << endl
17       << x << '\t' << y << endl;
18
19    // display x and y in scientific format
20    cout << "\nDisplayed in scientific format:" << endl
21       << scientific << x << '\t' << y << endl;
22
23    // display x and y in fixed format
24    cout << "\nDisplayed in fixed format:" << endl
25       << fixed << x << '\t' << y << endl;
26    return 0;
27 } // end main
```

```
Displayed in defualt format:
0.00123457     1.946e+009

Displayed in scientific format:
1.234567e-003   1.946000e+009

Displayed in fixed format:
0.001235      1946000000.000000
```

شکل ۱۸-۱۵ | مقادیر اعشاری در حالت نمایش عادی، علمی و ثابت شده.

٦-۷-۱۵ کنترل حروف بزرگ/کوچک (uppercase)

دستکاری کننده استریم **uppercase** یک حرف بزرگ **X** یا **E** را به همراه مقادیر هگزادسیمال یا با مقادیر اعشاری با نماد علمی به نمایش در می‌آورد (شکل ۱۹-۱۵). همچنین استفاده از دستکاری کننده **uppercase** سبب می‌شود تا تمام حروف موجود در یک مقدار هگزادسیمال تبدیل به حروف بزرگ شوند. بطور پیش‌فرض حروف موجود در مقادیر هگزادسیمال و توان در مقادیر اعشاری با نماد علمی بصورت کوچک ظاهر می‌شوند. برای از کار انداختن تنظیمات **uppercase** از **nouppercase** استفاده می‌شود.

```
1  // Fig. 15.19: Fig15_19.cpp
2  // Stream-manipulator uppercase.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::hex;
7  using std::showbase;
8  using std::uppercase;
9
10 int main()
11 {
12    cout << "Printing uppercase letters in scientific" << endl
13       << "notation exponents and hexadecimal values:" << endl;
14
15    // use std:uppercase to display uppercase letters; use std::hex and
16    // std::showbase to display hexadecimal value and its base
17    cout << uppercase << 4.345e10 << endl
18       << hex << showbase << 123456789  << endl;
19    return 0;
20 } // end main
```

```
Printing uppercase letters in scientific
notation exponents and hexadecimal valus:
4.345E+010
0x75BCD15
```

شکل ۱۹-۱۵ | دستکاری کننده استریم uppercase.

۷-۷-۱۵ قالب‌بندی بولی (boolalpha)

زمان C++ دارای نوع داده **bool** است که می‌توانند بصورت **false** یا **true** باشند و بعنوان جانشین مناسبی برای حالت قدیمی استفاده از صفر برای نشان دادن **false** و مقادیر غیرصفر برای نشان دادن **true** به شمار می‌آیند. یک متغیر بولی، صفر یا 1 را در حالت عادی چاپ می‌کند. با این وجود، می‌توانیم از دستکاری کننده استریم **boolalpha** برای تنظیم استریم خروجی در نمایش مقادیر بولی بصورت رشته‌ای "true" و "false" استفاده کنیم. دستکاری کننده استریم **noboolalpha** برای نمایش مقادیر بولی بفرم صحیح (0 یا مقدار غیر صفر، حالت پیش‌فرض) بکار گرفته می‌شود. برنامه شکل ۲۰-۱۵ به بررسی این دستکاری کننده استریم پرداخته است. خط 14 مقدار بولی را که در خط 11 با **true** تنظیم شده است را بصورت یک مقدار صحیح به نمایش در میآورد. خط 18 از دستکاری کننده **boolalpha** برای نمایش مقدار بولی بصورت رشته استفاده کرده است. سپس خطوط 22-21 مقدار بولی را تغییر داده و از **noboolalpha** استفاده کرده است، از اینرو خط 25 می‌تواند مقدار بولی را بعنوان یک مقدار صحیح به نمایش در آورد. در خط 29 از **boolalpha** برای نمایش مقدار بولی بصورت یک رشته استفاده شده است.

```
1  // Fig. 15.20: Fig15_20.cpp
2  // Demonstrating stream-manipulators boolalpha and noboolalpha.
3  #include <iostream>
4  using std::boolalpha;
5  using std::cout;
6  using std::endl;
7  using std::noboolalpha;
8
9  int main()
10 {
11    bool booleanValue = true;
12
13    // display default true booleanValue
14    cout << "booleanValue is " << booleanValue << endl;
15
16    // display booleanValue after using boolalpha
17    cout << "booleanValue (after using boolalpha) is "
18       << boolalpha << booleanValue << endl << endl;
19
20    cout << "switch booleanValue and use noboolalpha" << endl;
21    booleanValue = false; // change booleanValue
22    cout << noboolalpha << endl; // use noboolalpha
23
24    // display default false booleanValue after using noboolalpha
25    cout << "booleanValue is " << booleanValue << endl;
26
27    // display booleanValue after using boolalpha again
28    cout << "booleanValue (after using boolalpha) is "
29       << boolalpha << booleanValue << endl;
30    return 0;
31 } // end main
```

```
booleanValue is 1
booleanValue (after using boolalpha) is true

switch booleanValue and use noboolalpha

booleanValue is 0
booleanValue (after using boolalpha) is false
```

شکل ۲۰-۱۵ | دستکاری کننده‌های استریم boolalpha و noboolalpha.

۸-۷-۱۵ تنظیم و تنظیم مجدد وضعیت قالب‌بندی از طریق تابع عضو flags

در سرتاسر بخش ۷-۱۵ از انواع دستکاری کننده‌های استریم به منظور تغییر در صفات قالب‌بندی خروجی استفاده کردیم. اکنون بحث خود را متوجه نحوه باز گرداندن فرمت استریم خروجی به حالت یا وضعیت پیش‌فرض خود پس از اعمال دستکاری کننده‌ها می‌کنیم. تابع عضو **flags** بدون آرگومان، تنظیمات فرمت جاری را از نوع داده **fmtflags** برگشت می‌دهد (از کلاس **iso_base**)، که نشاندهنده وضعیت فرمت یا قالب‌بندی است. تابع عضو **flags** به همراه آرگومان **fmtflags** مبادرت به تنظیم فرمت وضعیت با توجه به آرگومان کرده و تنظیمات سابق را برمی‌گرداند. ممکن است مقدار تنظیم اولیه برگشتی توسط **flags** در سیستم‌های مختلف با هم تفاوت داشته باشند. در برنامه شکل ۲۱-۱۵ از تابع عضو **flags** برای ذخیره وضعیت فرمت اولیه استریم (خط 22)، سپس بازیابی فرمت اصلی استفاده شده است (خط 30).

```
1  // Fig. 15.21: Fig15_21.cpp
2  // Demonstrating the flags member function.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::ios_base;
7  using std::oct;
8  using std::scientific;
```

```
9  using std::showbase;
10
11 int main()
12 {
13    int integerValue = 1000;
14    double doubleValue = 0.0947628;
15
16    // display flags value, int and double values (original format)
17    cout << "The value of the flags variable is: " << cout.flags()
18       << "\nPrint int and double in original format:\n"
19       << integerValue << '\t' << doubleValue << endl << endl;
20
21    // use cout flags function to save original format
22    ios_base::fmtflags originalFormat = cout.flags();
23    cout << showbase << oct << scientific; // change format
24
25    // display flags value, int and double values (new format)
26    cout << "The value of the flags variable is: " << cout.flags()
27       << "\nPrint int and double in a new format:\n"
28       << integerValue << '\t' << doubleValue << endl << endl;
29
30    cout.flags( originalFormat ); // restore format
31
32    // display flags value, int and double values (original format)
33    cout << "The restored value of the flags variable is: "
34       << cout.flags()
35       << "\nPrint values in original format again:\n"
36       << integerValue << '\t' << doubleValue << endl;
37    return 0;
38 } // end main
```

```
The value of the flags variable is: 513
Print int and double in original format:
    1000    0.0947628

The value of the flags variable is: 012011
Print int and double in a new format:
    1750    9.476280e-002

The restored value of the flags variable is: 513
Print values in original format again:
1000    0.0947628
```

شکل ۲۱-۱۵ تابع عضو flags.

۸-۱۵ وضعیت خطا در استریم

می‌توان وضعیت یک استریم را با تست بیت‌های موجود در کلاس **ios_base** مشخص کرد. در برنامه شکل ۲۲-۱۵ نحوه تست این بیت‌ها نشان داده شده است.

بیت **eofbit** برای تنظیم یک استریم ورودی پس از مواجه شدن با انتهای فایل بکار گرفته می‌شود. برنامه می‌تواند با استفاده از تابع **eof** مشخص کند که آیا به انتهای فایل در استریم رسیده است یا خیر. فراخوانی

```
cin.eof()
```

اگر به انتهای فایل رسیده باشد مقدار **true** و در غیر اینصورت **false** برگشت می‌دهد.

بیت **failbit** برای یک استریم زمانیکه تنظیم می‌شود که یک خطای فرمت در استریم رخ داده باشد، مانند زمانیکه برنامه در حال وارد کردن مقادیر صحیح باشد و یک کاراکتر غیر عددی در استریم دیده شود. زمانیکه چنین خطای رخ دهد، کاراکترها مفقود نمی‌شوند. تابع عضو **fail** گزارشی در ارتباط با شکست یا عدم شکست عملیات تهیه می‌کند، بازیابی از چنین خطاهای امکان‌پذیر است.

بیت **badbit** زمانیکه خطای رخ دهد که باعث از دست رفتن داده شود، تنظیم می‌گردد (برای آن استریم). تابع عضو **bad** گزارشی از شکست یا عدم شکست عملیات تهیه می‌کند. معمولاً چنین شکست‌های غیرقابل بازیابی و جبران است.

```
1  // Fig. 15.22: Fig15_22.cpp
2  // Testing error states.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    int integerValue;
11
12    // display results of cin functions
13    cout << "Before a bad input operation:"
14       << "\ncin.rdstate(): " << cin.rdstate()
15       << "\n    cin.eof(): " << cin.eof()
16       << "\n   cin.fail(): " << cin.fail()
17       << "\n    cin.bad(): " << cin.bad()
18       << "\n   cin.good(): " << cin.good()
19       << "\n\nExpects an integer, but enter a character: ";
20
21    cin >> integerValue; // enter character value
22    cout << endl;
23
24    // display results of cin functions after bad input
25    cout << "After a bad input operation:"
26       << "\ncin.rdstate(): " << cin.rdstate()
27       << "\n    cin.eof(): " << cin.eof()
28       << "\n   cin.fail(): " << cin.fail()
29       << "\n    cin.bad(): " << cin.bad()
30       << "\n   cin.good(): " << cin.good() << endl << endl;
31
32    cin.clear(); // clear stream
33
34    // display results of cin functions after clearing cin
35    cout << "After cin.clear()" << "\ncin.fail(): " << cin.fail()
36       << "\ncin.good(): " << cin.good() << endl;
37    return 0;
38 } // end main
```

```
Before a bad input operation:
cin.rdstate(): 0
    cin.eof(): 0
   cin.fail(): 0
    cin.bad(): 0
   cin.good(): 1

Expects an integer, but enter a character: A
cin.rdstate(): 2
    cin.eof(): 0
   cin.fail(): 1
    cin.bad(): 0
   cin.good(): 0
After cin.clear()
cin.fail(): 0
cin.good(): 1
```

شکل ۲۲-۱۵ | تست وضعیت خطا.

اگر هیچ یک از بیت‌های **eofbit**، **failbit** یا **badbit** برای استریمی تنظیم نشده باشند، بیت **goodbit** تنظیم خواهد شد. تابع عضو **good** در صورتیکه توابع **bad**، **fail** و **eof** تماماً **false** باز گردانند، مبادرت به برگشت **true** می‌کند. عملیات I/O باید فقط با استریم‌های **good** کار کند.

تابع عضو **rdstate** مبادرت به بازگرداندن وضعیت خطا از استریم می‌کند. برای مثال با فراخوانی **cout.rdstate** وضعیت استریم برگشت داده خواهد شد، که سپس می‌تواند توسط یک عبارت **switch** به منظور بررسی **eofbit**، **badbit**، **failbit** و **goodbit** بکار گرفته شود.

تابع عضو **clear** برای اعاده کردن وضعیت استریم به حالت مناسب "good" بکار گرفته می‌شود. از اینرو است که I/O می‌تواند بر روی استریم ادامه یابد. آرگومان پیش‌فرض برای **clear** گزینه **goodbit** است، از اینرو عبارت

```
cin.clear();
```

مبادرت به ترخیص **cin** و تنظیم **goodbit** می‌کند. عبارت

```
cin.clear( iso::failbit );
```

مبادرت به تنظیم **failbit** می‌کند. امکان دارد برنامه‌نویس اینکار را به هنگام انجام عملیات ورودی بر روی **cin** با نوع داده تعریف شده از سوی کاربر و در مواجه شدن با یک مشکل انجام دهد. ممکن است نام **clear** چندان مناسب کاری که انجام می‌دهد نباشد، اما چنین است.

در برنامه شکل ۲۲-۱۵ به بررسی توابع عضو **rdstate**، **eof**، **fail**، **bad**، **good** و **clear** پرداخته شده است. تابع عضو **operator!** از کلاس **basic_ios** در صورتیکه **badbit**، **failbit** یا هر دو تنظیم شده باشند، مقدار **true** برگشت می‌دهد. تابع عضو **operator void *** در صورتیکه **badbit**، **failbit** یا هر دو تنظیم شده باشد مقدار **false** برگشت می‌دهد. این توابع به هنگام پردازش فایل سودمند هستند.

۹-۱۵ پیوند استریم خروجی با استریم ورودی

معمولاً برنامه‌های تعاملی درگیر یک **istream** برای ورودی و یک **ostream** برای خروجی هستند. زمانیکه یک پیغام بر روی صفحه ظاهر می‌شود، کاربر با وارد کردن داده مناسب به آن پاسخ می‌دهد. بدیهی است که این پیغام باید قبل از عملیات ورودی ظاهر شود. با بافر کردن خروجی، خروجی فقط زمانیکه بافر پر شود، یا بصورت صریح توسط برنامه یا بطور اتوماتیک در انتهای برنامه خالی شود، ظاهر خواهد شد. زبان C++ تابع عضو **tie** را برای همزمان کردن (یعنی با یکدیگر پیوند داشتن) عملیات یک **istream** و یک **ostream** در اختیار گذاشته تا مطمئن شویم که خروجی‌ها قبل از ورودی‌های متعاقب آنها ظاهر خواهند شد. در فراخوانی

```
cin.tie( &cout );
```

cout (یک **ostream**) به **cin** (یک **istream**) پیوند زده می‌شود. در واقع این نوع از فراخوانی اضافی است، چرا که C++ این نوع عملیات را بصورت اتوماتیک برای ایجاد یک محیط استاندارد I/O برای

کاربر فراهم می‌آورد. با این همه، کاربر می‌تواند سایر نوع‌های **isteam/ostream** را بصورت صریح به یکدیگر پیوند دهد. برای گشودن یک استریم ورودی، **inputStream**، از یک استریم خروجی، از فراخوانی زیر استفاده می‌شود:

```
inputStream.tie( 0 );
```