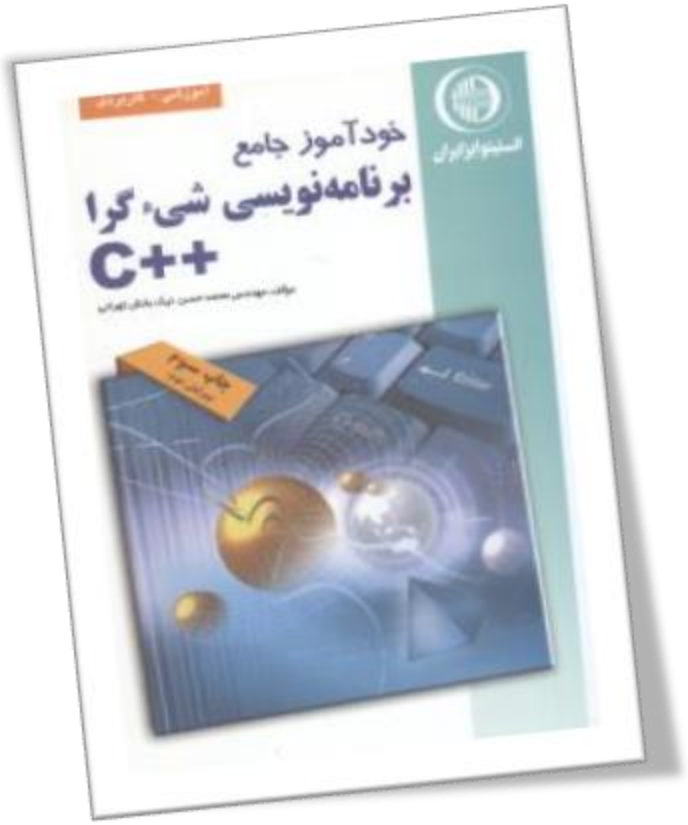

خودآموز جامع برنامه نویسی شی‌گرا C++

مؤلف : مهندس محمد حسن نیک‌بخش تهرانی

نوبت چاپ : سوم – زمستان ۱۳۸۶

انتشارات : انستیتوایزایران

تلخیص : مسعود حسینی

بهار ۱۳۹۲

Object Oriented Programming C++

```
Template < class T > Class myclass : private base{  // generic class
        T i, j; // private blocks
Protected :
        Char ch[10]; // protected blocks
Public :
        Myclass (int a, b) : base(a);  // constructor with 2 arg
        Myclass (int c);  // constructor with 1 arg ,  function overloading
        ~Myclass ();  // destructor
        Myclass (const &myclass); // copy constructor
        Friend int func (myclass & obj); //  friend function
        Myclass operator+ (myclass ob);   // operator overloading
        T show (){ return i ; } // member function
        Virtual char* get () = 0; // pure virtual function
} obj[100], ob;
```

Constructor : تابعی هم‌نام کلاس که هنگام ایجاد یک obj از کلاس فراخوانی می‌شود . برای global obj ها این تابع فقط زمان شروع برنامه فراخوانی می‌شود ولی برای local obj ها هر بار که کلاس جهت معرفی یک obj فراخوانی می‌شود ، این تابع نیز فراخوانی می‌شود .

Destructor : تابعی هم‌نام کلاس که قبل از آن علامت (~) قرار می‌گیرد و هنگام از بین رفتن یک obj فراخوانی می‌شود .

constructor می‌تواند پارامتر ورودی داشته باشد ولی destructor خیر .

نکته بسیار مهم : هنگام مقداردهی اولیه در constructor نمی‌توان هم آرگومان prototype و هم آرگومان جایی که تابع تعریف می‌شود را مقداردهی اولیه کرد ، بلکه فقط یکی از آن‌ها باید مقداردهی شود و همچنین اگر یک متغیر مقداردهی اولیه شد ، سایر متغیرهای بعد از آن نیز باید مقداردهی شوند .

به عنوان مثال این تکه‌برنامه syntax دارد .

```
Class myclass{
        Myclass(int s=0);
/* other blocks */ };
Myclass :: myclass(int s=0)
{ /* blocks */ }
```

اصلاح شده این تکه برنامه به این صورت است :

```
Class myclass{
        Myclass(int s);    // دقت شود
        /* other blocks */ };
Myclass :: myclass(int s=0) // دقت شود
        { /* blocks */ }
```

```
Class myclass{
        Myclass(int s=0);    // دقت شود
        /* other blocks */ };
Myclass :: myclass(int s) // دقت شود
        { /* blocks */ }
```

Arrow operator (->) : دسترسی به اعضای اشاره‌گر به یک شی .

Address operator (&) : به دست آوردن آدرس یک شی .

```
Myclass *p, obj(10);        p = &obj;        p->func();
```

Copy constructor : هنگامی که یک کلاس به عنوان ورودی یا خروجی یک تابع قرار می گیرد و یا اینکه یک شی کپی شود این تابع اجرا می‌شود .

```
Class_name(const class_name&){ /* blocks */ }
```

Friend function : گاهی اوقات لازم است تابع بدون اینکه عضو یک کلاس باشد ، به اعضای آن دست پیدا کند ، این توابع به صورت دوست تعریف می شوند . برای این منظور prototype تابع را به صورت زیر درون کلاسی که می‌خواهیم تابع به آن دسترسی داشته‌باشد ، تعریف می کنیم و خود تابع را به صورت معمول بیرون از کلاس تعریف می‌کنیم :

```
friend return_type func_name ();
```

در نگاه اول تابع دوست کاربرد چندانی ندارد ، چون تابع را به جای friend می توان به عنوان member کلاس تعریف کرد ، ولی نگاه دقیق تر این است که اگر بخواهیم یک تابع به اعضای private دو یا چند کلاس دسترسی داشته باشد ، دیگر نمی توانیم آن را به عنوان member تعریف کنیم بلکه باید به صورت friend تمامی کلاس‌ها تعریف شود . برای دسترسی به متغیر private در تابع friend نمی‌توان مستقیماً نام آن را آورد بلکه باید به صورت myclass.variable فراخوانی شود .

اشاره‌گر this : هر بار که یک تابع member فراخوانی می‌شود یک اشاره‌گر به شی‌ای که آن را فراخوانده است، ارسال می‌گردد که همان اشاره گر this است و به طور اتوماتیک به وجود می‌آید .

فقط توابع member اشاره گر this را ایجاد و به شی ارسال می کنند ، توابع دیگر مانند friendها فاقد این اشاره‌گر هستند .

Allocate operator (new) : این اپراتور جهت تخصیص حافظه استفاده می‌شود که نحوه استفاده از آن به صورت زیر می‌باشد :

```
P_var = new type [size] {arguments};
```

به عنوان مثال تکه‌برنامه زیر یک آرایه ۵ تایی دینامیکی از کلاس myclass ساخته و آدرس شروع آن را در pointer p قرار می‌دهد و constructor مربوط به هر کدام را با عدد داخل {} اجرا می‌کند :

```
P = new myclass[5]{1, 2, 3, 4, 5};
```

Free operator (delete) : این اپراتور جهت آزاد کردن حافظه تخصیص داده شده مورد استفاده قرار می‌گیرد . اگر حافظه‌ای که قصد آزاد کردن آن را داریم به صورت یک متغیر باشد ، این گونه (delete p_var) و اگر به صورت آرایه‌ای باشد بدین گونه (delete [] p_var) مورد استفاده قرار می‌گیرد .

Reference : یک اشاره‌گر ضمنی است که در تمام مواقع می‌تواند به عنوان اسم دوم یک متغیر مورد استفاده قرار گیرد و سه کاربرد دارد :

۱) استفاده به عنوان آرگومان یک تابع که هر گونه تغییر در این آرگومان به متغیر اصلی منتقل می‌شود.

۲) استفاده به عنوان مقدار برگشتی یک تابع که اگر مقدار برگشتی محلی باشد ، بیهوده است .

۳) ایجاد یک اشاره گر مرجع مستقل که اگر مقدار آن تغییر کند ، مقدار متغیری که به آن اشاره می‌کند نیز تغییر می‌کند .

```
myclass & func(myclass &s){ // ارسال آرگومان به تابع
    // blocks
    Return s; } // آرگومان برگشتی از تابع
Int main()
{   Int x;
    Int &ref = x;
    Ref = 10;  // x نیز برابر ۱۰ قرار می‌گیرد
    Myclass p;
    Func(p)=100; // متغیر درون شی برابر ۱۰۰ قرار می گیرد
    Return 0;}
```

Fuction overloading : گاهی اوقات ورودی توابع مشخص نیست یعنی بسته به نیاز کاربر می‌تواند int ، char و ... و یا اینکه تعداد پارامترهای ورودی آن متفاوت باشد ، برای این منظور می‌توان یک تابع را برای ورودی‌های متعدد overload کرد .

```
R_type f_name (arg1);        R_type f_name (arg2);        …
```

ابهامات overloading : گاهی اوقات توابعی را overload می‌کنیم که خود توابع مبهم نیستند ولی استفاده از آن‌ها توسط کاربر می‌تواند باعث ابهام شود ، نمونه‌ای از آن‌ها عبارتند از :

۱) تبدیلات اتوماتیک ؛ به عنوان مثال یک تابع برای دو نوع float و double ، overload شده است و کاربر یک عدد صحیح وارد می‌کند ، عدد صحیح هم می‌تواند به float تبدیل شود و هم به double و این عمل کامپایلر را دچار سردرگمی می‌کند .

۲) استفاده از پارامتر مرجع ؛ هرگاه تابعی را overload کنیم که یکی از آن ها دارای پارامتر مرجع و دیگری دارای پارامتر عادی است ، زمانی که یک متغیر به تابع ارسال می‌شود کامپایلر دچار ابهام می‌شود که value آن را به تابع اول ارسال کند یا reference آن را به تابع دوم .

۳) مقدار پیش‌فرض؛ فرض می‌کنیم که تابعی یکبار بدون پارامتر و بار دیگر با یک پارامتر و مقدار پیش‌فرض overload شده است، هرگاه کاربر مقداری را به تابع ارسال نمی‌کند ، کامپایلر دچار عدم تشخیص تابع می‌شود .

Pointer to function : گاهی اوقات به جای استفاده از تابع از آدرس آن استفاده می‌شود که اشاره‌گر به آدرس به صورت زیر تعریف می‌شود :

```
Return_type ( * Pointer_name ) (arguments);
```

Operator overloading : یک اپراتور را به دو روش member و friend می‌توان overload کرد .

اپراتورهایی که قابلیت overload کردن ندارند عبارتند از : :: ، : ? ، * ، . .

ابتدا به روش member می‌پردازیم که فرم کلی آن به صورت زیر است :

```
Return_type class_name :: operator# (arguments)
{ /* operation to be performed */ }
```

در این تکه‌برنامه اغلب مقدار بازگشتی تابع از نوع کلاسی است که اپراتور را برای آن overload کرده‌ایم و و اپراتور مورد نظر جایگزین # می‌شود . نکته قابل توجه این است که تعداد اپرندهای یک اپراتور قابل تغییر نیست و تقدم و تأخر اپراتورها نیز تغییر نمی کند و همچنین این اپراتورها نمی‌توانند دارای مقادیر پیش‌فرض باشند .

برای overload کردن اپراتورهای باینری (اپراتورهایی که دارای دو اپرند هستند) ، تابع اپراتوری فقط دارای یک پارامتر است . این پارامتر دریافت کننده شی‌ای است که به عنوان اپرند سمت راست مورد استفاده قرار خواهد گرفت و شی سمت چپ ، شی‌ای است که موجب فراخوانی تابع اپراتوری می‌شود . بهترا ست برای پارامتر تابع اپراتوری از اشاره‌گر مرجع (reference) استفاده شود تا destructor و copy constructor مشکلی ایجاد نکند .

برای overload کردن اپراتورهای منطقی نباید مقدار برگشتی را از نوع کلاس قرار دهیم بلکه باید نشان‌دهنده درست یا غلط باشند.

Overload کردن اپراتورهای یگانی نیز همانند باینری است با این تفاوت که تابع اپراتوری آن هیچ پارامتری ندارد ، چون این اپراتور فقط دارای یک اپرند است که خود موجب فراخوانی تابع می‌شود .

هرگاه بخواهیم مقدار برگشتی شی‌ای باشد که اپراتور روی آن اعمال شده است ، مقدار this* را برگشت می‌دهیم .

حال به روش friend می‌پردازیم که شکل کلی آن به صورت زیر است :

```
Friend return_type operator# (arguments) ;          // prototype درون کلاس
```

Return_type operator# (arguments) // تعریف تابع در بیرون از کلاس

{ /* operation to be performed */ }

اپراتور = را به کمک تابع friend نمی‌توان overload کرد.

چون در تابع friend اشاره‌گر this وجود ندارد ، پس تمامی توابع اپراتوری به تعداد اپرندهای اپراتور ، پارامتر دارند که پارامتر اول (اپرند سمت چپ) صدا کننده و پارامتر دوم (اپرند سمت راست) به عنوان دریافت کننده استفاده می‌شود ، البته این مورد فقط برای اپراتورهای باینری است زیرا اپراتورهای یگانی دارای یک پارامتر هستند که همان صداکننده است .

Inheritance : به ارث رسیدن ویژگی‌های یک کلاس به کلاس دیگر است . تعریف آن به صورت زیر است :

Class derived_class : access_specifier base_class1 : access_specifier base_class2 : …

{ /* body of class */ }

Access specifier : عمومی (public) ، خصوصی (private) ، محافظت شده (Protected)

Access specifier	Member of the base class		
	Public	Private	Protected
Public	Public for derived class	Private for base class	Protected for derived class
Private	Private for derived class	Private for base class	Private for derived class
protected	Protected for derived class	Private for base class	Protected for derived class

ترتیب اجرای constructor و destructor در توارث به این صورت است که ابتدا constructor کلاس‌های base به ترتیب از چپ به راست و سپس کلاس derived اجرا می‌شود و destructor برعکس عمل می‌کند .

Class a : public b : private c : public d

{ /* body of class a */ }

ترتیب اجرای destructor $a \rightarrow d \rightarrow c \rightarrow b$ / ترتیب اجرای constructor $b \rightarrow c \rightarrow d \rightarrow a$

برای ارسال پارامتر به تابع constructor ، از کلاس derived به کلاس base ، برای تعریف constructor کلاس derived به صورت زیر عمل می کنیم :

Derived_class_name(arguments) : base_class_constructor(arguments passed)

{ /* body of derived class constructor */ }

Virtual class : هنگامی که کلاس a به کلاس b و c و کلاس‌های b و c به کلاس d ارث برسند ، مشخصات کلاس a دوبار به کلاس d به ارث می‌رسد که باعث ابهام شده و کامپایلر syntax می‌دهد ، برای جلوگیری از این مشکل قبل از access specifier کلمه virtual را می‌آوریم که به کلاس مجازی معروف است .

Class derived_class : virtual access_specifier base_class { // body of derived class }

Virtual function : این توابع به روش معمول تعریف می‌شوند با این تفاوت که هنگام تعریف قبل از return_type کلمه virtual قرار می گیرد .

هنگامی که یک اشاره‌گر از کلاس base داشته باشیم می‌توانیم به کمک این اشاره‌گر به کلاس derived اشاره کنیم ، لازم به تذکر است که فقط توابعی توسط این اشاره گر فراخوانی می‌شود که در کلاس base نیز وجود داشته باشند .

تابع virtual یک خاصیت polymorphism است به طوری که اشاره‌گر به هر شی‌ای که اشاره کند ، تابع مربوط به آن فراخوانی می‌شود . این تابع فقط از طریق اشاره‌گر قابل دسترسی است و هر کلاسی که تابعی با این نام نداشته باشد ، تابع کلاس baseبرای آن اجرا خواهد شد .

در حقیقت تابع virtual یک نوع override است نه function overloading ، زیرا در overloading تعداد پارامترها می‌تواند تغییر کند ولی در virtual function این گونه نیست ، به همین منظور به آن override گفته می‌شود .

Pure virtual function : هرگاه تابع مجازی درون کلاس base هیچ بدنه‌ای نداشته باشد و برابر صفر قرار گیرد ، کلاس derived موظف است که این تابع را override کند . به این تابع ، تابع مجازی خالص گفته می‌شود .

Formatted I/O : تغییرات بر روی ورودی و خروجی به دلخواه کاربر .

Format flags	Explanation
Skipws	این علامت معمولاً برای خواندن مورد استفاده قرار می‌گیرد و با set شدن آن از کاراکترهای whitespace (space ، newline ، tab) صرف نظر می‌شود .
Right	این علامت برای خروجی مورد استفاده قرار می‌گیرد و با set شدن آن خروجی به سمت راست تغییر مکان می‌دهد .
Left	این علامت برای خروجی مورد استفاده قرارمی‌گیرد وبا set شدن آن خروجی به سمت چپ تغییرمکان می‌دهد.
Internal	این علامت برای خروجی مورد استفاده قرار می‌گیرد و با set شدن آن خروجی به سمت راست تغییر مکان می‌دهد . تفاوت آن با right در این است که اگر right ، set شود در اعداد علاوه بر عدد علامت آن نیز به سمت راست منتقل می‌شود ولی اگر internal ، set شود ، علامت عدد در سمت چپ باقی مانده و خود عدد به سمت راست منتقل می‌شود.

Oct	این علامت معمولاً برای خروجی مورد استفاده قرار می گیرد و عدد را به مبنای ۸ تغییر می‌دهد .
Hex	این علامت معمولاً برای خروجی مورد استفاده قرار می گیرد و عدد را به مبنای ۱۶ تغییر می‌دهد .
Dec	این علامت معمولاً برای خروجی مورد استفاده قرار می گیرد و عدد را به مبنای ۱۰ تغییر می‌دهد .
Showbase	این علامت برای خروجی مورد استفاده قرار می‌گیرد و با set شدن آن مبنای اعداد نیز در کنار آن‌ها چاپ می‌شود .
Unitbuf	Set شدن این علامت موجب بهبود کارایی سیستم ورودی / خروجی می‌شود . (خارج از محدوده بحث)
Showpoint	این علامت برای خروجی مورد استفاده قرار می‌گیرد و با set شدن آن صفرهای بی‌ارزش بعد از ممیز نیز بر چاپ می‌شود .
Uppercase	این علامت برای خروجی مورد استفاده قرار می گیرد و با set شدن آن حروف کوچک e در نماد علمی و x و سایر حروف در مبنای ۱۶ به حروف بزرگ تبدیل می‌شوند.
Scientific	این علامت برای خروجی مورد استفاده قرار می گیرد و با set شدن آن اعداد به صورت نماد علمی نوشته می‌شوند .
Fixed	این علامت برای خروجی مورد استفاده قرار می‌گیرد و با set شدن آن اعداد ممیز شناور با دقت ۶ رقم بعد از اعشار نمایش داده می‌شوند .
Stdio	این علامت بعد از هر بار خروجی stream ، flush می‌شود .
Showpos	این علامت برای خروجی مورد استفاده قرار می گیرد و با set شدن آن علامت + در کنار اعداد مثبت dec قرار می‌گیرد .

برای set کردن یکی از علائم فوق به شکل زیر عمل می‌کنیم :

```
Stream.setf(ios::format_flag1 | ios::format_flag2 | …);
```

برای برگشت به تنظیمات اولیه از تابع unsetf به شکل زیر استفاده می‌کنیم :

```
Stream.unsetf(ios::format_flag1 | ios::format_flag2 | …);
```

در برخی موارد به جای تغییر حالت علائم ، حالت فعلی آن ها مدنظر است که برای این منظور از توابع set و unset نمی‌توان استفاده کرد . برای رفع مشکل از تابع flags استفاده می کنیم .

Member function	Explanation
Width	این تابع حداقل پهنای فیلد مورد نظر را به دلخواه کاربر تنظیم می‌کند .
Precision	ممیز شناور اعداد اعشاری به طور پیش‌فرض ۶ رقم بعد از اعشار است که به کمک این تابع می‌توان آن را به دلخواه تغییر داد .
Fill	هنگامی که از تابع width استفاده می کنیم ، به طور پیش‌فرض در جاهای خالی space چاپ می‌شود ولی اگر بخواهیم این کاراکتر را تغییر دهیم ، از تابع fill استفاده می‌کنیم .

توابع width برای هر بار استفاده باید set شود ولی توابع fill و precision با هربار set شدن تا اینکه دوباره تغییر نکند ، همان مقدار قبلی می‌ماند .

I/O manipulator : توابع مخصوص فرمت ورودی / خروجی که می توانند به جای اینکه جدا از دستورالعمل I/O و مانند استفاده از توابع عضو IOS عمل کنند ، درون دستورالعمل I/O قرار گیرند . به عنوان مثال :

Cout << oct << 100 << hex << 25 << setw(10) << "abc" << endl;

Manipulator	Explanation	I / O
Dec	این تابع مبنای اعداد را به ۱۰ تغییر می‌دهد .	Output
Endl	این تابع یک کاراکتر newline چاپ کرده و stream را flush می کند .	Output
Ends	این تابع یک کاراکتر null چاپ می کند .	Output
Flush	این تابع stream را flush می کند .	Output
Hex	این تابع مبنای اعداد را به ۱۶ تغییر می‌دهد .	Output
Oct	این تابع مبنای اعداد را به ۸ تغییر می‌دهد .	Output
Resetiosflags	این تابع برای off کردن علائم ورودی به کار می‌رود .	Output / Input
Setbase	این تابع مبنای اعداد را به پارامتر ورودی تغییر می‌دهد .	Output
Setfill	این تابع جای‌های خالی (fill) را با کاراکتر ورودی پر می کند .	Output
Setiosflags	این تابع برای on کردن علائم ورودی به کار می‌رود .	Output / Input
Setprecision	این تابع برای تغییردقت در ممیز شناور اعداد اعشاری به کار می‌رود .	Output
Setw	این تابع برای افزایش پهنای field به کار می‌رود .	Output
Ws	این تابع از کاراکتر های whitespace صرف‌نظر می کند .	Input

نکته قابل توجه این است که manipulatorها فقط روی streamهایی اثر می‌گذارند که خود بخشی از آن stream هستند .

Inserter : overload کردن اپراتور >> به دلخواه خود که حتماً باید به صورت friend تعریف شود . فرم کلی آن به شکل زیر است :

```
Ostream &operator<< (ostream &stream, class_name obj)
{ // body of inserter
    Stream << … ;
    Return stream; }
```

Extractor : overload کردن اپراتور << به دلخواه خود که حتماً باید به صورت friend تعریف شود . فرم کلی آن به شکل زیر است :

```
Istream &operator>> (istream &stream, class_name obj)
{ // body of extractor
    Stream >> … ;
    Return stream; }
```

Custom manipulator : ایجاد manipulator های بدون پارامتر به دلخواه کاربر ، این manipulator ها به دو دسته ورودی و خروجی تقسیم می‌شوند .

فرم کلی manipulator های خروجی به صورت زیر است :

```
Ostream &manipulator_name (ostream &stream)
{ // body of manipulator
    Return stream; }
```

Manipulator های ورودی به صورت زیر تعریف می‌شوند :

```
Istream &manipulator_name (istream &stream)
{ // body of manipulator
    Return stream; }
```

علی‌رغم وجود یک پارامتر برای تعریف manipulatorها ، در استفاده از آن ها هیچ پارامتری قرار نمی‌دهیم ، حتی از علامت () هم نباید در جلوی آن ها استفاده کنیم .

File I/O basics : برای انجام عملیات ورودی/خروجی روی فایل ها ابتدا header ، fstream را include می‌کنیم و سپس نوع فایل را با استفاده از ifstream (فقط ورودی) ، ofstream (فقط خروجی) و fstream (ورودی و خروجی) تعیین می‌کنیم .

برای باز کردن یک فایل از تابع open به روش زیر استفاده می کنیم :

```
Stream_name.open(char* file_address, int mode, int access);
```

به یاد داشته باشیم که در نوشتن آدرس فایل به جای \ باید از \\ استفاده شود .

جای mode می توان از مقادیر جدول زیر با ترکیب | استفاده کرد .

Mode	Explanation
Ios::app	این مقدار تنها روی فایل‌های خروجی کار می کند و باعث می‌شود تمام خروجی‌ها به انتهای فایل اضافه شوند .
Ios::ate	این مقدار هم بر روی فایل‌های خروجی کار می کند و هم فایل‌های ورودی و باعث می‌شود focus برای خواندن یا نوشتن در انتهای فایل قرار گیرد .
Ios::in	تخصیص این مقدار مشخص می‌کند که فایل از نوع ورودی است و فقط روی فایل های ورودی/خروجی کار می کند .
Ios::out	تخصیص این مقدار مشخص می‌کند که فایل از نوع خروجی است و فقط روی فایل‌های ورودی/خروجی کار می کند .
Ios::nocreat	این مقدار باعث می‌شود تا اگر در هنگام فراخوانی فایل ، فایل وجود نداشته باشد ، فراخوانی با خطا مواجه شود.
Ios::noreplace	این مقدار باعث می‌شود تا اگر فایل با این نام موجود باشد ، فراخوانی دچار مشکل شود .
Ios::trunk	این مقدار باعث می‌شود تا اگرفایلی با این نام موجود باشد، محتویات آن از بین رفته و اندازه آن برابر صفر شود .

Access نحوه‌ی دسترسی به فایل را مشخص می کند و مقدار آن یکی از اعداد جدول زیر است :

Attributed	Meaning
0	Normal file_open access
1	Read_only file
2	Hidden file
4	System file
8	Archive bit set

البته این مقادیر ممکن است در ویندوز عمل نکند و فقط در سیستم‌عامل dos اجرا شود ، برای سایر سیستم عامل ها باید به help ویندوز مراجعه کرد .

Close	بعد از انجام عملیات روی فایل با این تابع آن را می‌بندیم .
Eof	این تابع کمک می کند تا بفهمیم به انتهای فایل رسیده ایم یا نه .
<<	این اپراتور عمل نوشتن را بر روی فایل انجام می دهد .
>>	این اپراتور عمل خواندن را از روی فایل انجام می‌دهد .

Binary file I/O :

Function	Explanation
Get(char &ch)	این تابع برای خواندن یک بایت از فایل استفاده می‌شود و آن را در ch قرار می‌دهد .

Put(char ch)	این تابع برای نوشتن یک بایت روی فایل استفاده می‌شود .
Read(unsignedhcar *buf, int num)	این تابع به اندازه num بایت از فایل می‌خواند و در buf قرار می‌دهد .
Write(unsignedhcar *buf, int num)	این تابع به اندازه num بایت روی فایل می نویسد .
Int gcount()	این تابع مشخص می کند ، چند بایت خوانده شده است .

چون c++ به طور اتوماتیک اشاره‌گر را از یک نوع به نوع دیگر تغییر نمی‌دهد ، در تابع read باید از type casting استفاده شود .

تابع get، overload نیز شده است که فرم آن به صورت زیر است :

Istream &get (char *buf, int num, char delim='character')

این تابع تا زمانی که به num بایت یا قبل از آن به کاراکتری که برابر delim قرارگرفته است ، نرسد ، عمل خواندن را انجام می‌دهد .

تابع getline دقیقاً مشابه تابع get است با این تفاوت که در get کاراکتر delim نیز خوانده شده و focus بعد از آن قرار می گیرد ولی در getline، focus قبل از این کاراکتر قرار می گیرد .

تابع peek کاراکتر بعدی را خوانده و کد اسکی آن را نشان می‌دهد .

به کمک تابع putback(ch) نیز می توان آخرین کاراکتر خوانده شده از یک جریان ورودی را برگرداند .

تابع flush باعث flush شدن بافر می‌شود .

Random access : برای این منظور از دوتابع seekg و seekp استفاده می‌شود .

Seekg(offset, seek_dir);

Seekp(offset, seek_dir);

در ابن دو تابع offset یک عدد int و seek_dir یکی از مقادیر ios::beg (شروع فایل) ، ios::cur (مکان فعلی) و ios::end (انتهای فایل) می‌باشد . وظیفه این دو تابع تغییر مکان focus به اندازه offset بایت از seek_dir می‌باشد .

تفاوت seekp و seekg در این است که seekp ، focus را برای تابع put جابه‌جا کرده ولی seekg آن را برای تابع get.

توابع tellg و tellp نیز به ترتیب مکان فعلی focus را برای توابع get و put مشخص می کنند .

Template : مجموعه ای از کلاس‌ها که فی‌نفسه کار مشابهی را انجام می‌دهند ، می توان آن‌ها را در یک قالب کلی (template) معرفی و مورد استفاده قرار داد .

اگر بخواهیم کلاس را به صورت template تعریف کنیم، به شکل زیر عمل می کنیم :

```
Template <class T> class class_name     // T یک نوع داده است
{ /* body of class */ }
```

برای تعریف توابع member یک generic class به فرم زیر عمل می کنیم :

```
Template <class T> return_type class_name < T > :: function_name (arguments)
{ /* body of function */ }
```

برای تعریف object از یک generic class همانند زیر عمل می کنیم :

```
Class_name < type /* for example int*/ > object_name;
```

دقت کنیم اگر برای class_name اپراتوری overload شده باشد ، باید آن اپراتور برای کلاس T نیز overload شود .

در تعریف generic class اگر بخواهیم از مقدار پیش‌فرض استفاده کنیم ، باید در <> به جای class از نوع پیش‌فرض استفاده کرد .

Template function : این توابع همانند generic class ها به صورت زیر تعریف می‌شوند :

```
Template <class T> return_type function_name (arguments)
{ /* body of function */ }
```

اگر T در آرگومان های تابع مورد استفاده قرار گرفته شده باشد ، مقدار T برابر آن type ورودی می‌شود و نیاز به مشخص کردن T نیست ولی اگر T در آرگومان ها استفاده نشده باشد ، هنگام استفاده باید در جلوی نام تابع و قبل از آرگومان‌ها نوع T را به صورت <type /*for example int */> مشخص کرد .

```
Template <class T> void func1 (T a)
{    int b=3;
     Return a*a; }
Func1 (2); // correct
Func1 <int> (2); // correct
```

```
Template <class T> func2(int a)
{   T b=3;
    Return a*b; }
func2 (2); // incorrect
func2 <int> (2); // correct
```

Exception : در اجرای برنامه کامپایلر با دو رویداد مواجه می‌شود ، اول آن‌هایی که در برخورد با آن‌ها آمادگی دارد و دوم آن‌هایی که کامپایلر آن‌ها را دوست ندارد ، زیرا آمادگی مواجه شدن با آن ها را ندارد .

Exception عبارت است از خطا یا مشکل قابل پیش‌بینی در برنامه که توسط سیستم عامل به‌وجود نیامده بلکه توسط برنامه شما رخ داده است .

Exception handling : توانایی رفع خطای احتمالی در برنامه . این عمل با سه کلمه کلیدی try ، throw ، catch انجام می‌شود که به توضیح هر کدام می‌پردازیم :

Try : این کلمه روال عادی برنامه را در قالب یک بلوک پی می‌گیرد و به کامپایلر می‌گوید که در این قسمت از برنامه آمادگی برخورد با exception وجود دارد .

Catch : این کلمه اعمالی که در برخورد با مشکل باید انجام شود را نشان می‌دهد و بلافاصله بعد از try می‌آید . این کلمه دارای یک آرگومان است که اگر هیچ آرگومانی نداشته باشد ، جای آرگومان آن (...) قرار می‌دهیم .

Throw : این کلمه در هنگام برخورد با مشکل ، با پرتاب کردن یک مقدار اجرای برنامه را به بلوک catch هدایت می کند و اعمال بعد از throw انجام نمی‌شود .

```
Try
{ /* try the program flow */ }
Catch (argument1)
{ /* catch the exception1 */ }
Catch (argument2)
{ /* catch the exception2 */ }
Catch (…)
{ /* catch another exception */ }
```

یک try را می‌توان در بدنه یک try دیگر استفاده کرد که به آن استثنا تودرتو یا nesting exception می‌گویند ، لازم به یادآوری است که هر Try باید catch مورد نیاز خود را دارا باشد .

به عنوان مثال تکه‌برنامه تقسیم به صورت زیر است :

```
Try
{    Cin >> a >> b;
     If (b==0)
          Throw "division by zero not allowed";
      Cout << a/b ; }
Catch (char *str)
{ cout << str; }
Catch(…)
{ /* no action */ }
```

کلمه throw در تمام توابعی که تعریف می‌شوند و در بلوک try از آن ها استفاده می‌شود ، نیز می‌تواند به کار رود ، البته در prototype تابع بعد از آرگومان‌ها باید کلمه throw() آورده شود تا مشخص شود که در این تابع از throw استفاده شده است . اگر درون پرانتز مقادیر خاصی قرار دهیم ، این استثنا به آن مقادیر محدود می‌شود .

```
Return_type function_name (arguments) throw(/*for example */ int, float)
{ /* body of function */ }
```

Main arguments : تابع main می تواند دارای دو پارامتر باشد که به صورت زیر می‌باشد :

```
Int main ( int argc, char * argv[] )
{ /* body of main */ }
```

در این تکه‌برنامه argc تعداد پارامترهای main را نگه می‌دارد (نام برنامه نیز جزء پارامترها محسوب می‌شود) و argv پارامترها را ذخیره می‌کند .

در argv[0] آدرس برنامه ، در argv[1] اولین پارامتر ، در argv[2] دومین پارامتر و به همین ترتیب تا انتها پارامترها درون argv ذخیره می‌شوند .

به عنوان مثال تکه برنامه زیر آدرس فایل و پارامترهای ورودی را چاپ می‌کند :

```
Int main (int argc, char* argv[])     // project 1   →   name project is test.exe
{
        Cout << setw(10) << left << argc ;
        For(int i=0; i<argc; i++)
                Cout << setw(15) << left << argv[i] ;
        Return 0;
}


Int main ()   // project 2
{
        System("c:\\test.exe  1 2 3 masoud 25");
        Return 0;
}
```

خروجی تکه‌برنامه فوق به صورت زیر است :

6 C:\\test.exe 1 2 3 Masoud 25

۱۰کاراکتر ۱۵کاراکتر ۱۵کاراکتر ...

پایان

مسعود حسینی