In The Name of Allah

شی گرایی در زبان C++

فهرست مطالب

- مباحث
  - اعلان کلاس‌ها
  - سازنده‌ها
  - فهرست مقداردهي در سازنده‌ها
  - توابع دستيابي
  - توابع عضو خصوصي
  - سازندهٔ کپي
  - نابود کننده
  - اشياي ثابت
  - اشاره‌گر به اشيا
  - اعضاي داده‌اي ايستا
  - توابع عضو ايستا

آشنایی با کلاس‌ها

مقدمه

«*شی‌گرایی*» رهیافت جدیدی بود که برای پاره ای از مشکلات برنامه نویسی راه حل داشت. این مضمون از دنیای فلسفه به جهان برنامه‌نویسی آمد و کمک کرد تا معضلات تولید و پشتیبانی نرم‌افزار کم‌تر شود.

اشیا را می‌توان با توجه به مشخصات ورفتار آن‌ها دسته بندی کرد.

در بحث شی‌گرایی به دسته‌ها «*کلاس*» می‌گویند و به نمونه‌های هر کلاس «*شی*» گفته می‌شود.

مشخصات هر شی را «*صفت*» می‌نامند و به رفتارهای هر شی «*متد*» می‌گویند.

برنامه‌نویسی شی گرا بر سه ستون استوار است:

الف. بسته‌بندی: یعنی این که داده‌های مرتبط، با هم ترکیب شوند و جزییات پیاده‌سازی مخفی شود.

ب. وراثت: در دنیای واقعی، وراثت به این معناست که یک شی وقتی متولد می‌شود، خصوصیات و ویژگی‌هایی را از والد خود به همراه دارد .

ج. چند ریختی: که به آن چندشکلی هم می‌گویند به معنای یک چیز بودن و چند شکل داشتن است. چندریختی بیشتر در وراثت معنا پیدا می‌کند.

اعلان کلاس‌ها

کد زیر اعلان یک کلاس را نشان می‌دهد.

```
class Ratio
{  public:
      void assign(int, int);
      viod print();
   private:
      int num, den;
};
```

اعلان اعضای کلاس درون یک بلوک انجام می‌شود و سرانجام یک سمیکولن بعد از بلوک نشان می‌دهد که اعلان کلاس پایان یافته است.

عبارت public و عبارت private . هر عضوی که ذیل عبارت public اعلان شود، یک «عضو عمومی» محسوب می‌شود و هر عضوی که ذیل عبارت private اعلان شود، یک «عضو خصوصی» محسوب می‌شود.

سازنده‌ها

وظیفهٔ تابع سازنده این است که حافظهٔ لازم را برای شیء جدید تخصیص داده و آن را مقداردهی نماید و با اجرای وظایفی که در تابع سازنده منظور شده، شیء جدید را برای استفاده آماده کند.

هر کلاس می‌تواند چندین سازنده داشته باشد. در حقیقت تابع سازنده می‌تواند چندشکلی داشته باشد.

سازنده‌ها، از طریق فهرست پارامترهای متفاوت از یکدیگر تفکیک می‌شوند

```
class Ratio
{  public:
      Ratio() { num = 0; den = 1; }
      Ratio(int n) { num = n; den = 1; }
      Ratio(int n, int d) { num = n; den = d; }
      void print() { cout << num << '/' << den; }
   private:
      int num, den;
};
```

یک کلاس می‌تواند سازنده‌های مختلفی داشته باشد. ساده‌ترین آن‌ها، سازنده‌ای است که هیچ پارامتری ندارد. به این سازنده *سازندهٔ پیش‌فرض* می‌گویند.

اگر در یک کلاس، سازندهٔ پیش‌فرض ذکر نشود، کامپایلر به طور خودکار آن را برای کلاس مذکور ایجاد می‌کند.

فهرست مقداردهی در سازنده‌ها

سازنده‌ها اغلب به غير از مقداردهي داده‌هاي عضو يک شي، کار ديگري انجام نمي‌دهند. به همين دليل در C++ يک واحد دستوري مخصوص پيش‌بيني شده که توليد سازنده را تسهيل مي‌نمايد. اين واحد دستوري *فهرست مقداردهي* نام دارد.

```
class Ratio
{  public:
      Ratio() : num(0) , den(1) { }
      Ratio(int n) : num(n) , den(1) { }
      Ratio(int n, int d) : num(n), den(d) { }
   private:
      int num, den;
};
```

توابع دستیابی

داده‌های عضو یک کلاس معمولا به صورت خصوصی (private) اعلان می‌شوند تا دستیابی به آن‌ها محدود باشد اما همین امر باعث می‌شود که نتوانیم در مواقع لزوم به این داده‌ها دسترسی داشته باشیم. برای حل این مشکل از توابعی با عنوان ***توابع دستیابی*** استفاده می‌کنیم.

تابع دستیابی یک تابع عمومی عضو کلاس است و به همین دلیل اجازهٔ دسترسی به اعضای داده‌ای خصوصی را دارد.

با استفاده از توابع دستیابی فقط می‌توان اعضای داده‌ای خصوصی را خواند ولی نمی‌توان آن‌ها را دست‌کاری کرد.

افزودن توابع دستیابی به کلاس Ratio

```
class Ratio
{  public:
      Ratio(int n=0, int d=1) : num(n) ,
den(d) { }
      int numerator() { return num; }
      int denomerator() { return den; }
   private:
      int num, den;
};
```

توابع عضو خصوصی

توابع عضو را گاهی می‌توانیم به شکل یک عضو خصوصی کلاس معرفی کنیم. واضح است که چنین تابعی از داخل برنامهٔ اصلی به هیچ عنوان قابل دستیابی نیست. این تابع فقط می‌تواند توسط سایر توابع عضو کلاس دستیابی شود. به چنین تابعی یک *تابع سودمند محلی* می‌گوییم.

```
class Ratio
{  public:
      Ratio(int n=0, int d=1) : num(n), den(d) { }
      void print() { cout << num << '/' << den <<
endl; }
      void printconv() { cout << toFloat() <<
endl; }
   private:
      int num, den;
      double toFloat();
};
```

این تابع فقط درون بدنهٔ تابع عضو printconv() استفاده شده و به انجام وظیفهٔ آن کمک می‌نماید و هیچ نقشی در برنامهٔ اصلی ندارد.

سازندهٔ کپی

می‌دانیم که به دو شیوه می‌توانیم متغیر جدیدی تعریف نماییم:

```
int x;
int x=k;
```

در روش اول متغیری به نام x از نوع int ایجاد می‌شود. در روش دوم هم همین کار انجام می‌گیرد با این تفاوت که پس از ایجاد x مقدار موجود در متغیر k که از قبل وجود داشته درون x کپی می‌شود. اصطلاحا x یک کپی از k است.

Ratio y(x);

کد بالا یک شی به نام y از نوع Ratio ایجاد می‌کند و تمام مشخصات شیء x را درون آن قرار می‌دهد. اگر در تعریف کلاس، سازندهٔ کپی ذکر نشود (مثل همهٔ کلاس‌های قبلی) به طور خودکار یک سازندهٔ کپی پیش‌فرض به کلاس افزوده خواهد شد.

افزودن یک سازندهٔ کپی به کلاس Ratio

```
{  public:
     Ratio(int n=0, int d=1) : num(n), den(d) { }
   Ratio(const Ratio& r) : num(r.num), den(r.den) { }
     void print() { cout << num << '/' << den; }
   private:
     int num, den;
};
```

سازندۀ کپی در سه وضعیت فرا خوانده می‌شود:

۱ – وقتی که یک شی هنگام اعلان از روی شیء دیگر کپی شود.

۲ – وقتی که یک شی به وسیلۀ مقدار به یک تابع ارسال شود.

۳ – وقتی که یک شی به وسیلۀ مقدار از یک تابع بازگشت داده شود .

نابود کننده

وقتی که شی به پایان زندگی‌اش برسد، تابع عضو دیگری به طور خودکار فراخوانی می‌شود تا نابودکردن آن شی را مدیریت کند. این تابع عضو، ***نابودکننده*** نامیده می‌شود .

سازنده وظیفه دارد تا منابع لازم را برای شی تخصیص دهد و نابودکننده وظیفه دارد آن منابع را آزاد کند.

هر کلاس فقط یک نابودکننده دارد.

افزودن یک نابودکننده به کلاس Ratio

```
class Ratio
{  public:
     Ratio() { cout << "OBJECT IS BORN.\n"; }
     ~Ratio() { cout << "OBJECT DIES.\n"; }
   private:
     int num, den;
};
```

اشیای ثابت

اشیا را نیز می‌توان با استفاده از عبارت const به صورت یک شیء ثابت اعلان کرد:

```
const Ratio PI(22,7);
```

اشاره‌گر به اشیا

می‌توانیم اشاره‌گر به اشیای کلاس نیز داشته باشیم. از آن‌جا که یک کلاس می‌تواند اشیای داده‌ای متنوع و متفاوتی داشته باشد، اشاره‌گر به اشیا بسیار سودمند و مفید است.

اشاره‌گر به اشیا برای ساختن فهرست‌های پیوندی و درخت‌های داده‌ای به کار می‌رود.

استفاده از اشاره‌گر به اشیا

```
class X
{  public:
      int data;
};
main()
{  X* p = new X;
   (*p).data = 22;       // equivalent to: p->data = 22;
   cout << "(*p).data = " << (*p).data << " = " << p->data << endl;
   p->data = 44;
   cout << " p->data = " << (*p).data << " = " << p->data << endl;
}
```

در این مثال، p اشاره‌گری به شیء x است. پس *p یک شیء x است و (*p).data دادهٔ عضو آن شی را دستیابی می‌کند.

حتما باید هنگام استفاده از *p آن را درون پرانتز قرار دهید زیرا عملگر انتخاب عضو (.) تقدم بالاتری نسبت به عملگر مقداریابی (*) دارد.

اگر پرانتزها قید نشوند و فقط *p.data نوشته شود، کامپایلر این خط را به صورت *(p.data) تفسیر خواهد کرد که این باعث خطا می‌شود.

مثال بعدی اهمیت بیشتری دارد و کاربرد اشاره‌گر به اشیا را بهتر نشان می‌دهد.

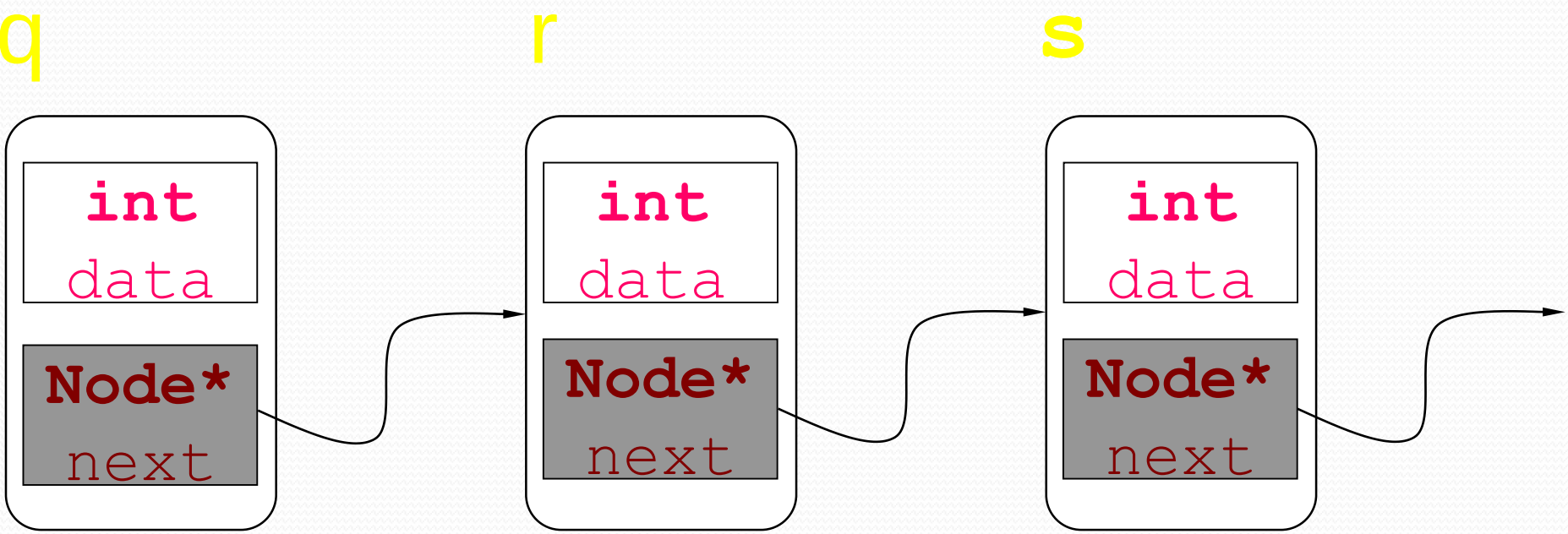
فهرست‌های پیوندی با استفاده از کلاس **Node**

به کلاسی که در زیر اعلان شده دقت کنید:

```
class Node
{  public:
     Node(int d, Node* p=0) : data(d),
  next(p) { }
     int data;
     Node* next;
};
```

عبارت بالا کلاسی به نام Node تعریف می‌کند که اشیای این کلاس دارای دو عضو داده‌ای هستند که یکی متغیری از نوع int است و دیگری یک اشاره‌گر از نوع همین کلاس. این کار واقعا ممکن است و باعث می‌شود بتوانیم یک شی را با استفاده از همین اشاره‌گر به شیء دیگر پیوند دهیم و یک زنجیره بسازیم.

اگر اشیای q و r و s از نوع Node باشند، می‌توانیم پیوند این سه شی را به صورت زیر مجسم کنیم:

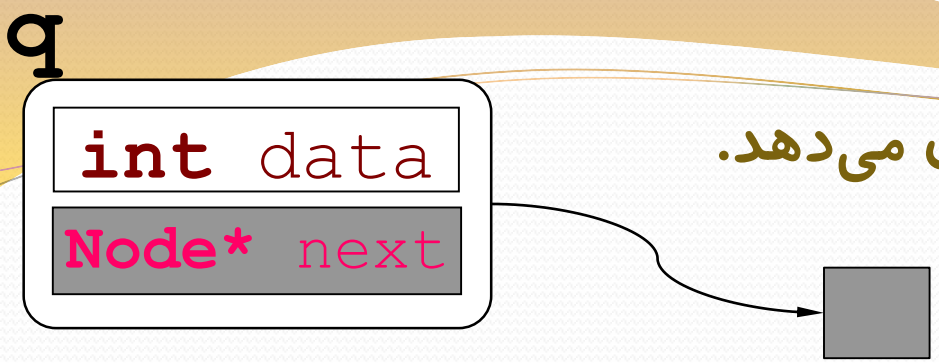


به تابع سازنده نیز دقت کنید که چطور هر دو عضو داده‌ای شیء جدید را مقداردهی می‌کند.

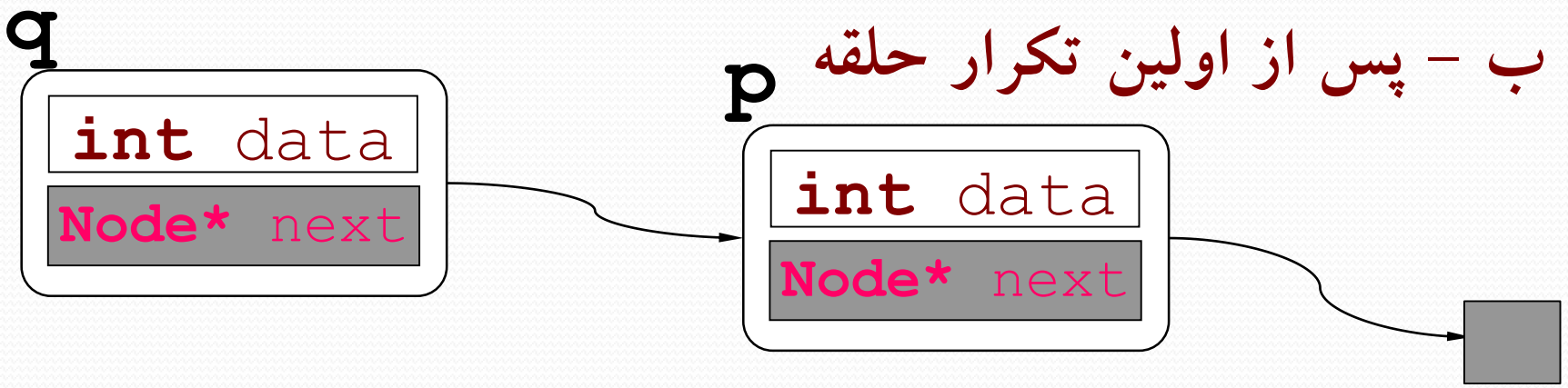
```
int main()
{  int n;
   Node* p;
   Node* q=0;
   while (cin >> n)
   {  p = new Node(n, q);
      q = p;
   }
   for ( ; p->next; p = p->next)
      cout << p->data << " -> ";
   cout << "*\n";
}
```

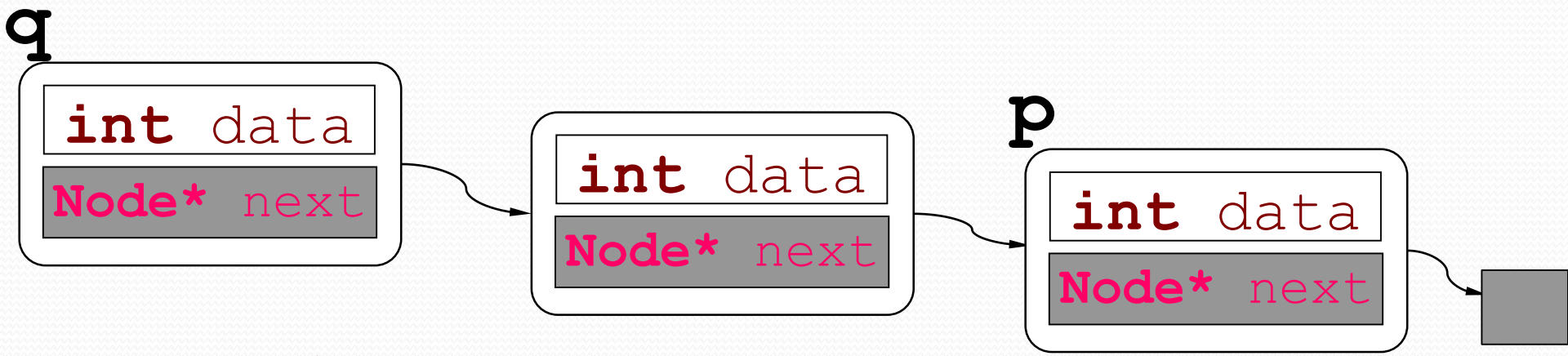
شکل زیر روند اجرای برنامه را نشان می‌دهد.

الف – قبل از شروع حلقه



ب – پس از اولین تکرار حلقه





ج – پس از دومین تکرار حلقه

اعضای داده‌ای ایستا

هر وقت که شیئی از روی یک کلاس ساخته می‌شود، آن شی مستقل از اشیای دیگر، داده‌های عضو خاص خودش را دارد. گاهی لازم است که مقدار یک عضو داده‌ای در همهٔ اشیا یکسان باشد. اگر این عضو مفروض در همهٔ اشیا تکرار شود، هم از کارایی برنامه می‌کاهد و هم حافظه را تلف می‌کند. در چنین مواقعی بهتر است آن عضو را به عنوان یک عضو ایستا اعلان کنیم .

عضو ایستا عضوی است که فقط یک نمونه از آن ایجاد می‌شود و همه اشیا از همان نمونهٔ مشترک استفاده می‌کنند. با استفاده از کلمهٔ کلیدی static در شروع اعلان متغیر، می‌توانیم آن متغیر را به صورت ایستا اعلان نماییم. یک متغیر ایستا را فقط باید به طور مستقیم و مستقل از اشیا مقداردهی نمود. کد زیر نحوهٔ اعلان و مقداردهی یک عضو داده‌ای ایستا را بیان می‌کند:

```
class X
{  public:
   static int n;       // declaration of n as a static data member
};
int X::n = 0;          // definition of n
```

خط آخر نشان می‌دهد که متغیرهای ایستا را باید به طور مستقیم و مستقل از اشیا مقداردهی کرد.

متغیرهای ایستا به طور پیش‌فرض با صفر مقداردهی اولیه می‌شوند. بنابراین مقداردهی صریح به این گونه متغیرها ضروری نیست مگر این که بخواهید یک مقدار اولیهٔ غیر صفر داشته باشید.

یک عضو داده‌ای ایستا

کد زیر، کلاسی به نام widget اعلان می‌کند که این کلاس یک عضو داده‌ای ایستا به نام count دارد. این عضو، تعداد اشیای widget که موجود هستند را نگه می‌دارد. هر وقت که یک شیء widget ساخته می‌شود، از طریق سازنده مقدار count یک واحد افزایش می‌یابد و هر زمان که یک شیء widget نابود می‌شود، از طریق نابودکننده مقدار count یک واحد کاهش می‌یابد:

```
class Widget
{  public:
      Widget() { ++count; }
      ~Widget() { --count; }
      static int count;
};
int Widget::count = 0;
main()
{  Widget w, x;
   cout << "Now there are " << w.count << " widgets.\n";
   {  Widget w, x, y, z;
      cout << "Now there are " << w.count << " widgets.\n";
   }
   cout << "Now there are " << w.count << " widgets.\n";
   Widget y;
   cout << "Now there are " << w.count << " widgets.\n";
}
```

```
Now there are 2 widgets.
Now there are 6 widgets.
Now there are 2 widgets.
Now there are 3 widgets.
```

توجه کنید که چگونه چهار شیء widget درون بلوک داخلی ایجاد شده است. هنگامی که اجرای برنامه از آن بلوک خارج می‌شود، این اشیا نابود می‌شوند و لذا تعداد کل widgetها از 6 به 2 تقلیل می‌یابد.

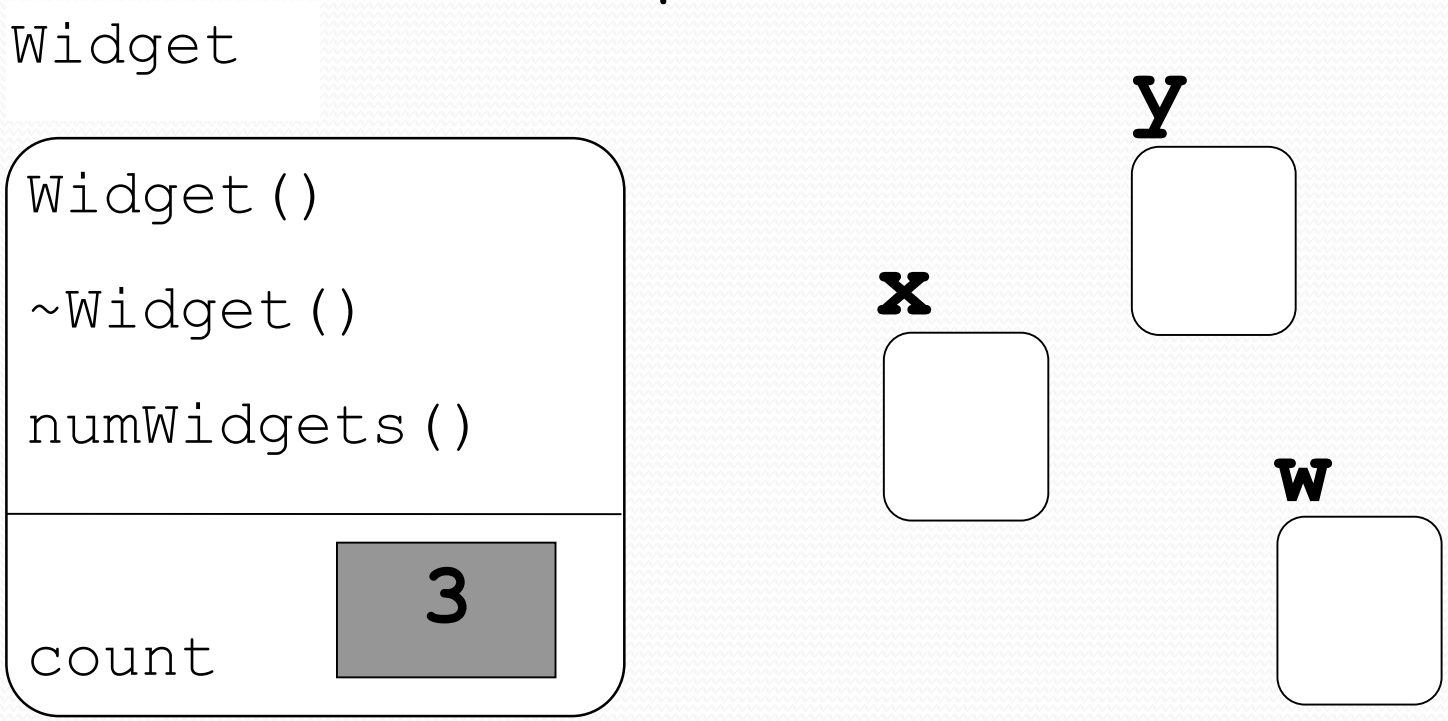
یک عضو داده‌ای ایستا مثل یک متغیر معمولی است: فقط یک نمونه از آن موجود است بدون توجه به این که چه تعداد شی از آن کلاس موجود باشد. از آن‌جا که عضو داده‌ای ایستا عضوی از کلاس است، می‌توانیم آن را به شکل یک عضو خصوصی نیز اعلان کنیم.

یک عضو داده‌ای ایستا و خصوصی

```
class Widget
{  public:
      Widget() { ++count; }
      ~Widget() { --count; }
      int numWidgets() { return count; }
   private:
      static int count;
};
```

```
int Widget::count = 0;

main()

{  Widget w, x;

   cout << "Now there are " << w.numWidgets() << " widgets.\n";

   {  Widget w, x, y, z;

      cout << "Now there are " << w.numWidgets() << " widgets.\n";

   }

   cout << "Now there are " << w.numWidgets() << " widgets.\n";

   Widget y;

   cout << "Now there are " << w.numWidgets() << " widgets.\n";

}
```

این برنامه مانند مثال قبل کار می‌کند با این تفاوت که متغیر ایستای **count** به شکل یک عضو خصوصی اعلان شده و به همین دلیل به تابع دستیابی **numWidgets()** نیاز داریم تا بتوانیم درون برنامهٔ اصلی به متغیر **count** دسترسی داشته باشیم. می‌توانیم کلاس **Widget** و اشیای **x** و **y** و **w** را مانند مقابل تصور کنیم:



توابع عضو ایستا

با دقت در مثال قبلی به دو ایراد بر می‌خوریم: اول این که گرچه متغیر count یک عضو ایستا است ولی برای خواندن آن حتما باید از یک شیء موجود استفاده کنیم. در مثال قبلی از شیء W برای خواندن آن استفاده کرده‌ایم. این باعث می‌شود که مجبور شویم همیشه مواظب باشیم عضو ایستای مفروض از طریق یک شی که الان موجود است فراخوانی شود.

یک تابع عضو ایستا

کد زیر همان کد مثال قبلی است با این فرق که در این کد، تابع دستیابی کننده نیز به شکل ایستا اعلان شده است:

```
class Widget
{  public:
      Widget() { ++count; }
      ~Widget() { --count; }
      static int num() { return count; }
   private:
      static int count;
};
```

```
int Widget::count = 0;
int main()
{  cout << "Now there are " << Widget::num() << " widgets.\n";
   Widget w, x;
   cout << "Now there are " << Widget::num() << " widgets.\n";
   {  Widget w, x, y, z;
      cout << "Now there are " << Widget::num() << " widgets.\n";
   }
   cout << "Now there are " << Widget::num() << " widgets.\n";
   Widget y;
   cout << "Now there are " << Widget::num() << " widgets.\n";
}
```

وقتی تابع num() به صورت ایستا تعریف شود، از اشیای کلاس مستقل می‌شود و برای فراخوانی آن نیازی به یک شیء موجود نیست و می‌توان با کد Widget::num() به شکل مستقیم آن را فراخوانی کرد.