In The Name of Allah

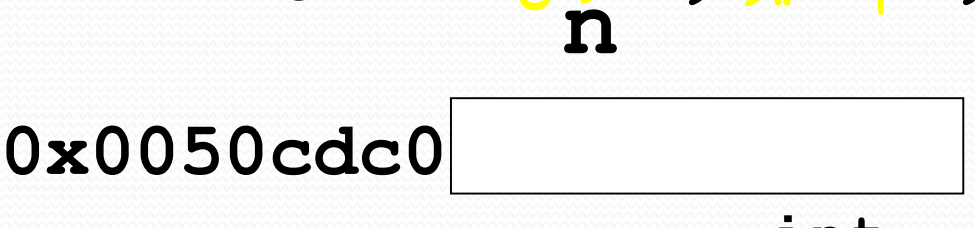

اشاره گرها در زبان C++

فهرست مطالب

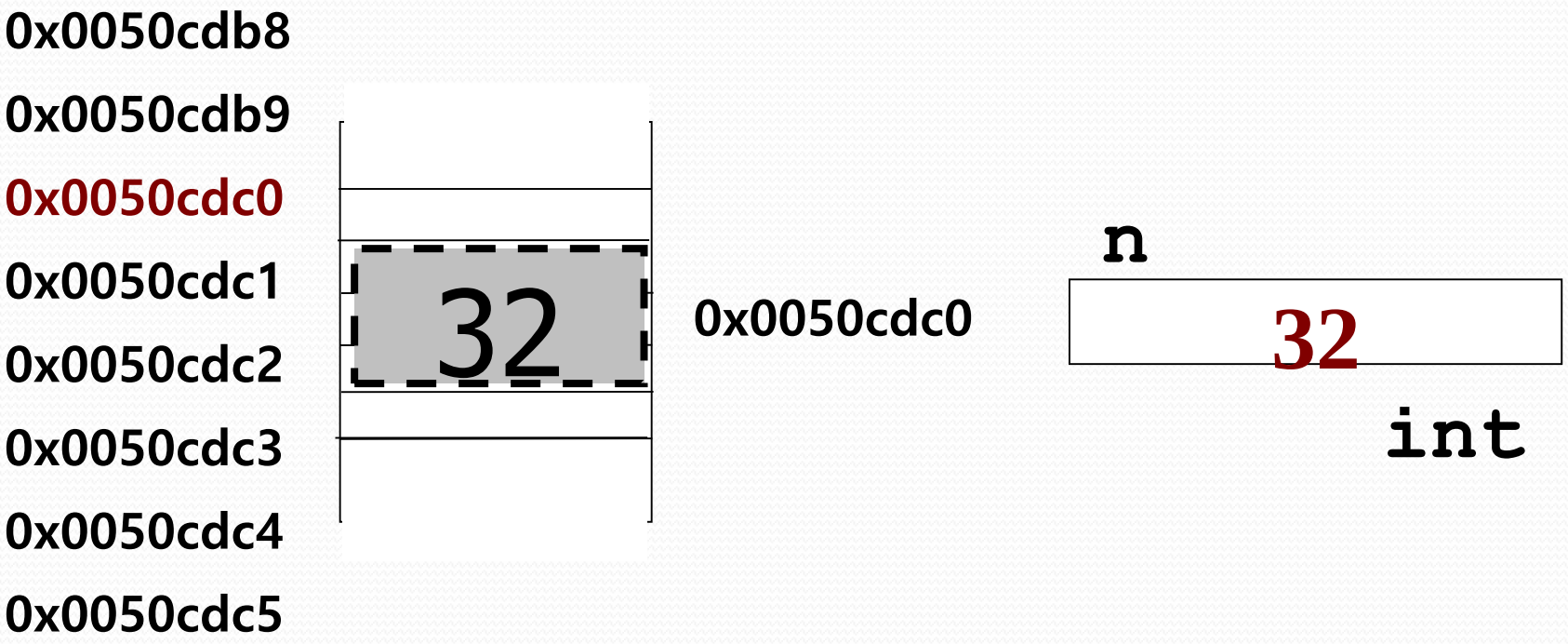
- مباحث
  - عملگر ارجاع
  - ارجاع ها
  - اشاره گرها
  - مقداریابی
  - بازگشت از نوع ارجاع
  - آرایه ها و اشاره گرها
  - عملگر new و delete
  - آرایه های پویا
  - اشاره گر ثابت
  - آرایه ای از اشاره گرها
  - اشاره گر به اشاره گر
  - اشاره گر به توابع
  - Null

مقدمه

- حافظهٔ رایانه را می‌توان به صورت یک آرایهٔ بزرگ در نظر گرفت. برای مثال رایانه‌ای با ۲۵۶ مگابایت RAM در حقیقت حاوی آرایه‌ای به اندازهٔ ۲۶۸،۴۳۵،۴۵۶ خانه است که اندازهٔ هر خانه یک بایت است.
- این خانه‌ها دارای ایندکس صفر تا ۲۶۸،۴۳۵،۴۵۵ هستند. به ایندکس هر بایت، آدرس حافظهٔ آن می‌گویند.
- آدرس‌های حافظه را با اعداد شانزده‌دهی نشان می‌دهند. پس رایانهٔ مذکور دارای محدوده آدرس 0x00000000 تا 0x0fffffff می‌باشد.
- هر وقت که متغیری را اعلان می‌کنیم، سه ویژگی اساسی به آن متغیر نسبت داده می‌شود: «نوع متغیر» و «نام متغیر» و «آدرس حافظه» آن.



اگر متغیر فوق به شکل int n=32; مقداردهی اولیه شود، آنگاه بلوک حافظه به شکل زیر خواهد بود. مقدار 32 در چهار بایتی که برای آن متغیر منظور شده ذخیره می‌شود.



عملگر ارجاع

در C++ برای بدست آوردن آدرس یک متغیر می‌توان از *عملگر ارجاع* ا **&** استفاده نمود. به این عملگر «علمگر آدرس» نیز می‌گویند. عبارت &n آدرس متغیر n را به دست می‌دهد.

```
int main()
{  int n=44;
   cout << " n = " << n << endl;
 cout << "&n = " << &n << endl;
}
```

ارجاع‌ها

یک «ارجاع» یک اسم مستعار یا واژهٔ مترادف برای متغیر دیگر است.
نحو اعلان یک ارجاع به شکل زیر است:

type& ref_name = *var_name*;

type **نوع متغیر است، ref_name نام مستعار است و *var_name* نام متغیری است که می‌خواهیم برای آن نام مستعار بسازیم. برای مثال در اعلان :**

```
int& rn=n;    // rn is a synonym for n
```

rn یک ارجاع یا نام مستعار برای n است. البته n باید قبلا اعلان شده باشد.

در برنامهٔ زیر **rn** به عنوان یک ارجاع به **n** اعلان می‌شود:

```
int main()
{  int n=44;
   int& rn=n;      // rn is a synonym for n
   cout << "n = " << n << ", rn = " << rn << endl;
   --n;
   cout << "n = " << n << ", rn = " << rn << endl;
   rn *= 2;
   cout << "n = " << n << ", rn = " << rn << endl;
}
```

```
n = 44, rn = 44
n = 43, rn = 43
n = 86, rn = 86
```

همانند ثابت‌ها، ارجاع‌ها باید هنگام اعلان مقداردهی اولیه شوند با این تفاوت که مقدار اولیهٔ یک ارجاع، یک متغیر است نه یک لیترال. بنابراین کد زیر اشتباه است:

```
int& rn=44; // ERROR: 44 is not a variable;
```

گرچه برخی از کامپایلرها ممکن است دستور بالا را مجاز بدانند ولی با نشان دادن یک هشدار اعلام می‌کنند که یک متغیر موقتی ایجاد شده تا rn به حافظهٔ آن متغیر، ارجاع داشته باشد.

درست است که ارجاع با یک متغیر مقداردهی می‌شود، اما ارجاع به خودی خود یک متغیر نیست.

یک متغیر، فضای ذخیره‌سازی و نشانی مستقل دارد، حال آن که ارجاع از فضای ذخیره‌سازی و نشانی متغیر دیگری بهره می‌برد.

```
int main()
{  int n=44;
   int& rn=n;          // rn is a synonym for n
   cout << "  &n = " << &n << ",  &rn = " << &rn << endl;
   int& rn2=n;         // rn2 is another synonym for n
   int& rn3=rn;        // rn3 is another synonym for n
   cout << "&rn2 = " << &rn2 << ", &rn3 = "
<< &rn3 << endl;
}
```

اشاره‌گرها

می‌دانیم که اعداد صحیح را باید در متغیری از نوع int نگهداری کنیم و اعداد اعشاری را در متغیرهایی از نوع float.

به همین ترتیب کاراکترها را باید در متغیرهایی از نوع char نگهداریم و مقدارهای منطقی را در متغیرهایی از نوع bool.

اما آدرس حافظه را در چه نوع متغیری باید قرار دهیم؟

عملگر ارجاع & آدرس حافظهٔ یک متغیر موجود را به دست می‌دهد. می‌توان این آدرس را در متغیر دیگری ذخیره نمود.

متغیری که یک آدرس در آن ذخیره می‌شود *اشاره‌گر* نامیده می‌شود.

برای این که یک اشاره‌گر اعلان کنیم، ابتدا باید مشخص کنیم که آدرس چه نوع داده‌ای قرار است در آن ذخیره شود. سپس از عملگر اشاره * استفاده می‌کنیم تا اشاره‌گر را اعلان کنیم.

برای مثال دستور :

```
float* px;
```

اشاره‌گری به نام px اعلان می‌کند که این اشاره‌گر، آدرس متغیرهایی از نوع float را نگهداری می‌نماید. به طور کلی برای اعلان یک اشاره‌گر از نحو زیر استفاده می‌کنیم:

```
type* pointername;
```

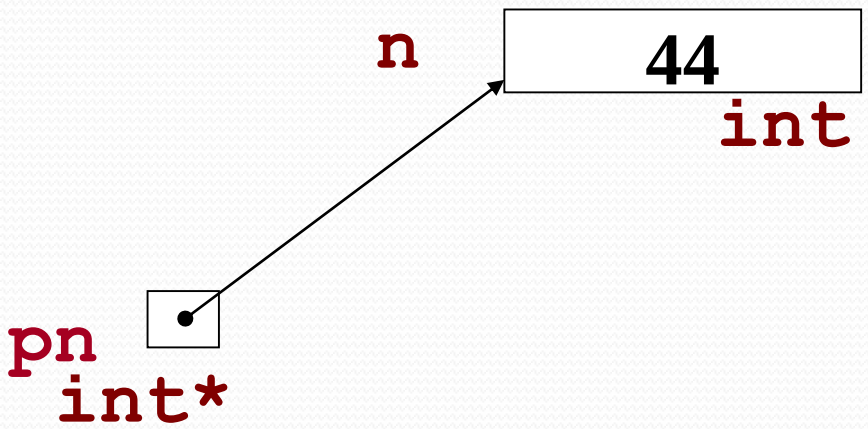
که *type* نوع متغیرهایی است که این اشاره‌گر آدرس آن‌ها را نگهداری می‌کند و pointername نام اشاره‌گر است.

برنامهٔ زیر یک متغیر از نوع int به نام n و یک اشاره‌گر از نوع int* به نام pn را اعلان می‌کند:

```
int main()
{  int n=44;
   cout << "n = " << n << ", &n = " << &n
<< endl;
   int* pn=&n;   // pn holds the address of n
   cout << "       pn = " << pn << endl;
   cout << "&pn = " << &pn << endl;}
```

اما pn یک متغیر مستقل است و آدرس مستقلی دارد. &pn آدرس pn را به دست می‌دهد. خط سوم خروجی ثابت می‌کند که متغیر pn مستقل از متغیر n است. تصویر زیر به درک بهتر این موضوع کمک می‌کند. در این تصویر ویژگی‌های مهم n و pn نشان داده شده. pn یک اشاره‌گر به n است و n مقدار 44 دارد.

وقتی می‌گوییم «pn به n اشاره می‌کند» یعنی درون pn آدرس n قرار دارد.



مقداریابی

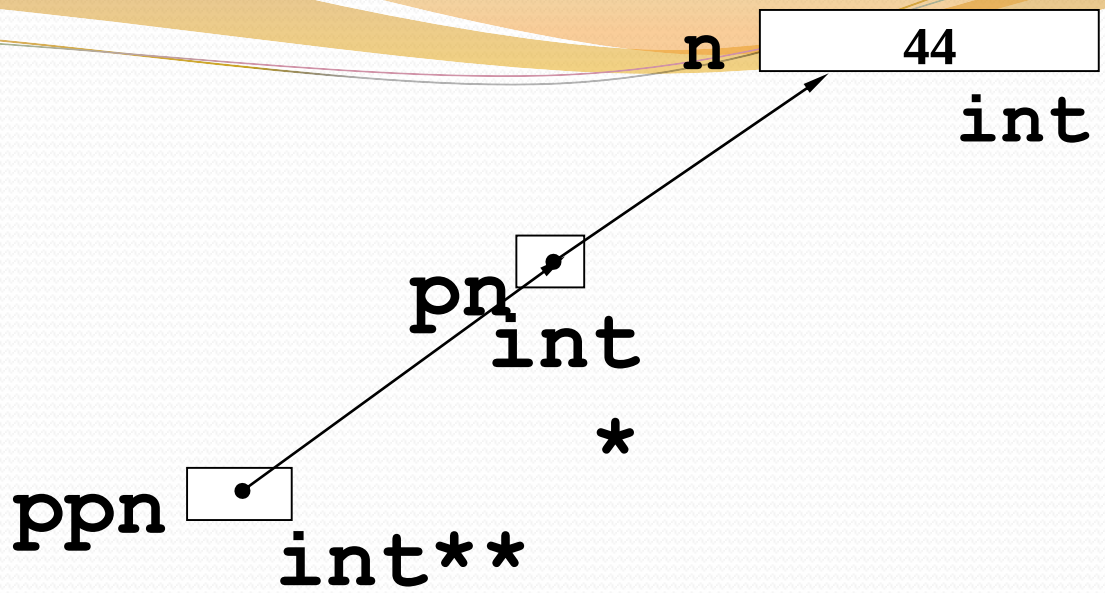
فرض کنید n دارای مقدار 22 باشد و pn اشاره‌گری به n باشد. با این حساب باید بتوان از طریق pn به مقدار 22 رسید. با استفاده از * می‌توان مقداری که اشاره‌گر به آن اشاره دارد را به دست آورد.

به این کار *مقداریابی اشاره‌گر* می‌گوییم.

```
int main()
{  int n=44;
   cout << "n = " << n << ", &n = " << &n << endl;
   int* pn=&n;     // pn holds the address of n
   cout << "         pn = " << pn << endl;
   cout << "&pn = " << &pn << endl;
   cout << "*pn = " << *pn << endl;
}
```

یک اشاره‌گر به هر چیزی می‌تواند اشاره کند، حتی به یک اشاره‌گر دیگر. به مثال زیر دقت کنید.

```
{  int n=44;
   cout << "    n = " << n << endl;
   cout << "   &n = " << &n << endl;
   int* pn=&n;      // pn holds the address of n
   cout << "   pn = " << pn << endl;
   cout << "  &pn = " << &pn << endl;
   cout << "  *pn = " << *pn << endl;
   int** ppn=&pn;   // ppn holds the address of pn
   cout << "  ppn = " << ppn << endl;
   cout << " &ppn = " << &ppn << endl;
   cout << " *ppn = " << *ppn << endl;
   cout << "**ppn = " << **ppn << endl;
}
```



در برنامهٔ بالا متغیر n از نوع int تعریف شده. pn اشاره‌گری است که به n اشاره دارد. پس نوع pn باید int* باشد. ppn اشاره‌گری است که به pn اشاره می‌کند. پس نوع ppn باید int** باشد. همچنین چون ppn به pn اشاره دارد، پس *ppn مقدار pn را نشان می‌دهد و چون pn به n اشاره دارد، پس *pn مقدار n را می‌دهد.

چپ مقدارها، راست مقدارها

یک دستور جایگزینی دو بخش دارد: بخشی که در سمت چپ علامت جایگزینی قرار می‌گیرد و بخشی که در سمت راست علامت جایگزینی قرار می‌گیرد. مثلا دستور **n = 55;** متغیر **n** در سمت چپ قرار گرفته و مقدار **55** در سمت راست. این دستور را نمی‌توان به شکل **55 = n;** نوشت زیرا مقدار **55** یک ثابت است و نمی‌تواند مقدار بگیرد. پس هنگام استفاده از عملگر جایگزینی باید دقت کنیم که چه چیزی را در سمت چپ قرار بدهیم و چه چیزی را در سمت راست.

چیزهایی که می‌توانند در سمت چپ جایگزینی قرار بگیرند «چپ‌مقدار» خوانده می‌شوند و چیزهایی که می‌توانند در سمت راست جایگزینی قرار بگیرند «راست‌مقدار» نامیده می‌شوند. متغیرها (و به طور کلی اشیا) چپ‌مقدار هستند و لیترال‌ها (مثل **15** و **"ABC"**) راست مقدار هستند.

یک ثابت در ابتدا به شکل یک چپ‌مقدار نمایان می‌شود:

```
const int MAX = 65535;      // MAX is an lvalue
```

اما از آن پس دیگر نمی‌توان به عنوان چپ مقدار از آن‌ها استفاده کرد:

```
MAX = 21024;     // ERROR: MAX is constant
```

به این گونه چپ‌مقدارها، چپ‌مقدارهای «تغییر ناپذیر» گفته می‌شود. مثل آرایه‌ها:

```
int a[] = {1,2,3};    // O.K
a[] = {1,2,3};        // ERROR
```

مابقی چپ‌مقدارها که می‌توان آن‌ها را تغییر داد، چپ‌مقدارهای «تغییر پذیر» نامیده می‌شوند. هنگام اعلان یک ارجاع به یک چپ‌مقدار نیاز داریم:

```
int& r = n;          // O.K. n is an lvalue
```

اما اعلان‌های زیر غیرمعتبرند زیرا هیچ کدام چپ‌مقدار نیستند:

```
int& r = 44;         // ERROR: 44 is not an lvalue

int& r = n++;        // ERROR: n++ is not an lvalue

int& r = cube(n);    // ERROR: cube(n) is not an lvalue
```

1 – L_values 2- R_values

یک تابع، چپ‌مقدار نیست اما اگر نوع بازگشتی آن یک ارجاع باشد، می‌توان تابع را به یک چپ‌مقدار تبدیل کرد.

بازگشت از نوع ارجاع

در بحث توابع، ارسال از طریق مقدار و ارسال از طریق ارجاع را دیدیم. این دو شیوهٔ تبادل در مورد بازگشت از تابع نیز صدق می‌کند:

بازگشت از طریق مقدار و بازگشت از طریق ارجاع. توابعی که تاکنون دیدیم بازگشت به طریق مقدار داشتند. یعنی همیشه یک مقدار به فراخواننده برمی‌گشت. می‌توانیم تابع را طوری تعریف کنیم که به جای مقدار، یک ارجاع را بازگشت دهد. مثلا به جای این که مقدار m را بازگشت دهد، یک ارجاع به m را بازگشت دهد.

وقتی بازگشت به طریق مقدار باشد، تابع یک راست‌مقدار خواهد بود زیرا مقدارها لیترال هستند و لیترال‌ها راست‌مقدارند. به این ترتیب تابع را فقط در سمت راست یک جایگزینی می‌توان به کار برد مثل:

```
m = f();
```

وقتی بازگشت به طریق ارجاع باشد، تابع یک چپ‌مقدار خواهد بود زیرا ارجاع‌ها چپ‌مقدار هستند. در این حالت تابع را می‌توان در سمت چپ یک جایگزینی قرار داد مثل :

```
f() = m;
```

برای این که نوع بازگشتی تابع را به ارجاع تبدیل کنیم کافی است عملگر ارجاع را به عنوان پسوند نوع بازگشتی درج کنیم.

```
int& max(int& m, int& n)
{  return (m > n ? m : n);}
int main()
{  int m = 44, n = 22;
   cout << m << ", " << n << ", " << max(m,n) << endl;
   max(m,n) = 55;
cout << m << ", " << n << ", " << max(m,n) << endl;
}
```

```
44, 22, 44
55, 22, 55
```

تابع max() از بین m و n مقدار بزرگ‌تر را پیدا کرده و سپس ارجاعی به آن را باز می‌گرداند.

بنابراین اگر m از n بزرگ‌تر باشد، تابع max(m,n) آدرس m را برمی‌گرداند.

پس وقتی می‌نویسیم max(m,n) = 55; مقدار 55 در حقیقت درون متغیر m قرار می‌گیرد (اگر m>n باشد).

به بیانی ساده، فراخوانی max(m,n) خود m را بر می‌گرداند نه مقدار آن را.

اخطار:

وقتی یک تابع پایان می‌یابد، متغیرهای محلی آن نابود می‌شوند. پس هیچ وقت ارجاعی به یک متغیر محلی بازگشت ندهید زیرا وقتی کار تابع تمام شد، آدرس متغیرهای محلی‌اش غیر معتبر می‌شود و ارجاع بازگشت داده شده ممکن است به یک مقدار غیر معتبر اشاره داشته باشد. تابع max() در مثال بالا یک ارجاع به m یا n را بر می‌گرداند. چون m و n خودشان به طریق ارجاع ارسال شده‌اند، پس محلی نیستند و بازگرداندن ارجاعی به آن‌ها خللی در برنامه وارد نمی‌کند.

```
float& component(float* v, int k)
{  return v[k-1];}
int main()
{  float v[4];
   for (int k = 1; k <= 4; k++)
      component(v,k) = 1.0/k;
   for (int i = 0; i < 4; i++)
      cout << "v[" << i << "] = " << v[i] << endl;
}
```

```
v[0] = 1
v[1] = 0.5
v[2] = 0.333333
v[3] = 0.25
```

آرایه‌ها و اشاره‌گرها

گرچه اشاره‌گرها از انواع عددی صحیح نیستند اما بعضی از اعمال حسابی را می‌توان روی اشاره‌گرها انجام داد. حاصل این می‌شود که اشاره‌گر به خانهٔ دیگری از حافظه اشاره می‌کند. اشاره‌گرها را می‌توان مثل اعداد صحیح افزایش و یا کاهش داد و می‌توان یک عدد صحیح را به آن‌ها اضافه نمود یا از آن کم کرد. البته میزان افزایش یا کاهش اشاره‌گر بستگی به نوع داده‌ای دارد که اشاره‌گر به آن اشاره دارد.

```
int main()
{  const int SIZE = 3;
   short a[SIZE] = {22, 33, 44};
   cout << "a = " << a << endl;
   cout << "sizeof(short) = " << sizeof(short) << endl;
   short* end = a + SIZE; // converts SIZE to offset 6
   short sum = 0;
   for (short* p = a; p < end; p++)
   {  sum += *p;
      cout << "\t p = " << p;
      cout << "\t *p = " << *p;
      cout << "\t sum = " << sum << endl;
   }
   cout << "end = " << end << endl;
}
```

این مثال نشان می‌دهد که هر گاه یک اشاره‌گر افزایش یابد، مقدار آن به اندازهٔ تعداد بایت‌های شیئی که به آن اشاره می‌کند، افزایش می‌یابد. مثلا اگر p اشاره‌گری به double باشد و sizeof(double) برابر با هشت بایت باشد، هر گاه که p یک واحد افزایش یابد، اشاره‌گر p هشت بایت به پیش می‌رود.

مثلا کد زیر :

```
float a[8];

float* p = a;    // p points to a[0]

++p; // increases the value of p by sizeof(float)
```

اگر floatها ۴ بایت را اشغال کنند آنگاه p++ مقدار درون p را ۴ بایت افزایش می‌دهد و p += 5; مقدار درون p را ۲۰ بایت افزایش می‌دهد. با استفاده از خاصیت مذکور می‌توان آرایه را پیمایش نمود: یک اشاره‌گر را با آدرس اولین عنصر آرایه مقداردهی کنید، سپس اشاره‌گر را پی در پی افزایش دهید. هر افزایش سبب می‌شود که اشاره‌گر به عنصر بعدی آرایه اشاره کند. یعنی اشاره‌گری که به این نحو به کار گرفته شود مثل ایندکس آرایه عمل می‌کند.

همچنین با استفاده از اشاره‌گر می‌توانیم مستقیما به عنصر مورد نظر در آرایه دستیابی کنیم:

```
float* p = a;        // p points to a[0]
p += 5;              // now p points to a[5]
```

یک نکتهٔ ظریف در ارتباط با آرایه‌ها و اشاره‌گرها وجود دارد: اگر اشاره‌گر را بیش از ایندکس آرایه افزایش دهیم، ممکن است به بخش‌هایی از حافظه برویم که هنوز تخصیص داده نشده‌اند یا برای کارهای دیگر تخصیص یافته‌اند. تغییر دادن مقدار این بخش‌ها باعث بروز خطا در برنامه و کل سیستم می‌شود. همیشه باید مراقب این خطر باشید.

کد زیر نشان می‌دهد که چطور این اتفاق رخ می‌دهد.

```
float a[8];
float* p = a[7]; // points to last element in the array
++p; //now p points to memory past last element!
*p = 22.2;           // TROUBLE!
```

مثال بعدی نشان می‌دهد که ارتباط تنگاتنگی بین آرایه‌ها و اشاره‌گرها وجود دارد. نام آرایه در حقیقت یک اشاره‌گر ثابت (const) به اولین عنصر آرایه است. همچنین خواهیم دید که اشاره‌گرها را مانند هر متغیر دیگری می‌توان با هم مقایسه نمود.

پیمایش عناصر آرایه از طریق آدرس

```
int main()
{  short a[] = {22, 33, 44, 55, 66};
   cout << "a = " << a << ", *a = " << *a << endl;
   for (short* p = a; p < a +5; p++)
    cout << "p = " << p << ", *p = " << *p << endl;
}
```

در نگاه اول، **a** و **p** مانند هم هستند: هر دو به نوع **short** اشاره می‌کنند و هر دو دارای مقدار **0x3fffd08** هستند. اما **a** یک اشاره‌گر ثابت است و نمی‌تواند افزایش یابد تا آرایه پیمایش شود. پس به جای آن **p** را افزایش می‌دهیم تا آرایه را پیمایش کنیم. شرط **(p < a+5)** حلقه را خاتمه می‌دهد. **a+5** به شکل زیر ارزیابی می‌شود:

0x3fffd08 + 5*sizeof(short) = 0x3fffd08 + 5*2 = 0x3fffd08 + 0xa = 0x3fffd12

پس حلقه تا زمانی که **p < 0x3fffd12** باشد ادامه می‌یابد.

عملگر زیرنویس [] مثل عملگر مقداریابی * رفتار می‌کند. هر دوی این‌ها می‌توانند به عناصر آرایه دسترسی مستقیم داشته باشند.

```
a[0] == *a
a[1] == *(a + 1)
a[2] == *(a + 2)
    ...
    ...
```

پس با استفاده از کد زیر نیز می‌توان آرایه را پیمایش نمود:

```
for (int i = 0; i < 8; i++)
     cout << *(a + i) << endl;
```

در این مثال، تابع **loc()** در میان **n1** عنصر اول آرایهٔ **a1** به دنبال **n2** عنصر اول آرایهٔ **a2** می‌گردد. اگر پیدا شد، یک اشاره‌گر به درون **a1** برمی‌گرداند که **a2** از آن‌جا شروع می‌شود وگرنه اشاره‌گر **NULL** را برمی‌گرداند.

```
short* loc(short* a1, short* a2, int n1, int n2)
{  short* end1 = a1 + n1;
   for (short* p1 = a1; p1 <end1; p1++)
      if (*p1 == *a2)
      {  for (int j = 0; j < n2; j++)
         if (p1[j] != a2[j]) break;
         if (j == n2) return p1;
      }
   return 0;
```

```
int main()
{  short a1[9] = {11, 11, 11, 11, 11, 22, 33, 44, 55};
   short a2[5] = {11, 11, 11, 22, 33};
   cout << "Array a1 begins at location\t" << a1 << endl;
   cout << "Array a2 begins at location\t" << a2 << endl;
   short* p = loc(a1, a2, 9, 5);
   if (p)
   {  cout << "Array a2 found at location\t" << p << endl;
      for (int i = 0; i < 5; i++)
         cout << "\t" << &p[i] << ": " << p[i] << "\t"
              << &a2[i] << ": " << a2[i] << endl; }
   else cout << "Not found.\n";}
```

عملگر new

وقتی یک اشاره‌گر شبیه این اعلان شود:

```
float* p;    // p is a pointer to a float
```

یک فضای چهاربایتی به p تخصیص داده می‌شود (معمولا sizeof(float) چهار بایت است). حالا p ایجاد شده است اما به هیچ جایی اشاره نمی‌کند زیرا هنوز آدرسی درون آن قرار نگرفته. به چنین اشاره‌گری *اشاره‌گر سرگردان* می‌گویند. اگر سعی کنیم یک اشاره‌گر سرگردان را مقداریابی یا ارجاع کنیم با خطا مواجه می‌شویم.

مثلا دستور:

*p = 3.14159; // **ERROR:** no storage has been allocated for *P

خطاست. زیرا p به هیچ آدرسی اشاره نمی‌کند و سیستم عامل نمی‌داند که مقدار 3.14159 را کجا ذخیره کند. برای رفع این مشکل می‌توان اشاره‌گرها را هنگام اعلان، مقداردهی کرد:

```
float x = 0;    // x cintains the value 0

float* p = &x   // now p points to x

*p = 3.14159;   // O.K. assigns this value to address that p points to
```

می توان این کار با استفاده از عملگر new انجام داد:

```
float* p;
p = new float; // allocates storage for 1 float
*p = 3.14159; // O.K. assigns this value to that storage
```

دقت کنید که عملگر new فقط خود p را مقداردهی می‌کند نه آدرسی که p به آن اشاره می‌کند. می‌توانیم سه خط فوق را با هم ترکیب کرده و به شکل یک دستور بنویسیم:

```
float* p = new float(3.141459);
```

با این دستور، اشاره‌گر p از نوع float* تعریف می‌شود و سپس یک بلوک خالی از نوع float منظور شده و آدرس آن به p تخصیص می‌یابد و همچنین مقدار 3.14159 در آن آدرس قرار می‌گیرد. اگر عملگر new نتواند خانهٔ خالی در حافظه پیدا کند، مقدار صفر را برمی‌گرداند. اشاره‌گری که این چنین باشد، «اشاره‌گر تهی» یا NULL می‌نامند.

با استفاده از کد هوشمند زیر می‌توانیم مراقب باشیم که اشاره‌گر تهی ایجاد نشود:

```
double* p = new double;
if (p == 0) abort();      // allocator failed: insufficent memory
else *p = 3.141592658979324;
```

در این قطعه کد، هرگاه اشاره‌گری تهی ایجاد شد، تابع **abort()** فراخوانی شده و این دستور لغو می‌شود.

تاکنون دانستیم که به دو طریق می‌توان یک متغیر را ایجاد و مقداردهی کرد. روش اول:

```
float x = 3.14159;              // allocates named memory
```

و روش دوم:

```
float* p = new float(3.14159);  // allocates unnamed memory
```

در حالت اول، حافظهٔ مورد نیاز برای **x** هنگام کامپایل تخصیص می‌یابد. در حالت دوم حافظهٔ مورد نیاز در زمان اجرا و به یک شیء بی‌نام تخصیص می‌یابد که با استفاده از ***p** قابل دستیابی است.

عملگر delete

عملگر **delete** عملی برخلاف عملگر **new** دارد. کارش این است که حافظهٔ اشغال شده را آزاد کند. وقتی حافظه‌ای آزاد شود، سیستم عامل می‌تواند از آن برای کارهای دیگر یا حتی تخصیص‌های جدید استفاده کند. عملگر **delete** را تنها روی اشاره‌گرهایی می‌توان به کار برد که با دستور **new** ایجاد شده‌اند. وقتی حافظهٔ یک اشاره‌گر آزاد شد، دیگر نمی‌توان به آن دستیابی نمود مگر این که دوباره این حافظه تخصیص یابد:

```
float* p = new float(3.14159);
delete p;           // deallocates q
*p = 2.71828; // ERROR: q has been deallocated
```

وقتی اشاره گر p در کد بالا آزاد شود، حافظه‌ای که توسط new به آن تخصیص یافته بود، آزاد شده و به میزان sizeof(float) به حافظهٔ آزاد اضافه می‌شود. وقتی اشاره‌گری آزاد شد، به هیچ چیزی اشاره نمی‌کند؛ مثل متغیری که مقداردهی نشده. به این اشاره‌گر، اشاره‌گر سرگردان می‌گویند.

اشاره‌گر به یک شیء ثابت را نمی‌توان آزاد کرد:

```
const int* p = new int;
delete p; // ERROR: cannot delete pointer to const objects
```

علت این است که «ثابت‌ها نمی‌توانند تغییر کنند».

اگر متغیری را صریحا اعلان کرده‌اید و سپس اشاره‌گری به آن نسبت داده‌اید، از عملگر **delete** استفاده نکنید. این کار باعث اشتباه غیر عمدی زیر می‌شود:

```
float x =3.14159; // x contains the value 3.14159
float* p = &x; // p contains the address of x
delete p; // WARNING: this will make x free
```

کد بالا باعث می‌شود که حافظهٔ تخصیص‌یافته برای **x** آزاد شود. این اشتباه را به سختی می‌توان تشخیص داد و اشکال‌زدایی کرد.

آرایه‌های پویا

نام آرایه در حقیقت یک اشاره‌گر ثابت است که در زمان کامپایل، ایجاد و تخصیص داده می‌شود:

```
float a[20]; //a is a const pointer to a block of 20 floats

float* const p = new float[20];    // so is p
```

هم a و هم p اشاره‌گرهای ثابتی هستند که به بلوکی حاوی ۲۰ متغیر float اشاره دارند. به اعلان a *بسته‌بندی ایستا* می‌گویند زیرا این کد باعث می‌شود که حافظهٔ مورد نیاز برای a در زمان کامپایل تخصیص داده شود. وقی برنامه اجرا شود، به هر حال حافظهٔ مربوطه تخصیص خواهد یافت حتی اگر از آن هیچ استفاده‌ای نشود.

می‌توانیم با استفاده از اشاره‌گر، آرایهٔ فوق را طوری تعریف کنیم که حافظه مورد نیاز آن فقط در زمان اجرا تخصیص یابد:

```
float* p = new float[20];
```

دستور بالا، ۲۰ خانهٔ خالی حافظه از نوع float را در اختیار گذاشته و اشاره‌گر p را به خانهٔ اول آن نسبت می‌دهد. به این آرایه، «آرایهٔ پویا» می‌گویند. به این طرز ایجاد اشیا *بسته‌بندی پویا* یا «بسته‌بندی زمان جرا» می‌گویند.

آرایهٔ ایستای **a** و آرایهٔ پویای **p** را با یکدیگر مقایسه کنید. آرایهٔ ایستای **a** در زمان کامپایل ایجاد می‌شود و تا پایان اجرای برنامه، حافظهٔ تخصیصی به آن مشغول می‌ماند.

ولی آرایهٔ پویای **p** در زمان اجرا و هر جا که لازم شد ایجاد می‌شود و پس از اتمام کار نیز می‌توان با عملگر **delete** حافظهٔ تخصیصی به آن را آزاد کرد:

```
delete [] p;
```

برای آزاد کردن آرایهٔ پویای **p** براکت‌ها **[]** قبل از نام **p** باید حتما قید شوند زیرا **p** به یک آرایه اشاره دارد.

مثال ۱۵-۷ استفاده از آرایه‌های پویا

تابع **get()** در برنامهٔ زیر یک آرایهٔ پویا ایجاد می‌کند:

```
void get(double*& a, int& n)
{  cout << "Enter number of items: ";  cin >> n;
   a = new double[n];
   cout << "Enter " << n << " items, one per line:\n";
   for (int i = 0; i < n; i++)
   {  cout << "\t" << i+1 << ": ";
 cin >> a[i];
   }}
void print(double* a, int n)
{  for (int i = 0; i < n; i++)
   cout << a[i] << " " ;
   cout << endl;
}
```

```
int main()
{ double* a;// a is simply an unallocated pointer
   int n;
   get(a,n);       // now a is an array of n doubles  print(a,n);
   delete [] a;// now a is simply an unallocated pointer again
   get(a,n); // now a is an array of n doubles   print(a,n);
}
```

Enter number of items: **4**

Enter 4 items, one per line:

1: **44.4**

2: **77.7**

3: **22.2**

4: **88.8**

44.4 77.7 22.2 88.8

Enter number of items: **2**

Enter 2 items, one per line:

1: **3.33**

2: **9.99**

3.33 9.99

اشاره‌گر ثابت

«اشاره‌گر به یک ثابت» با «اشاره‌گر ثابت» تفاوت دارد. این تفاوت در قالب مثال زیر نشان داده شده است.

در این کد چهار اشاره‌گر اعلان شده. اشاره‌گر p، اشاره‌گر ثابت cp، اشاره به یک ثابت pc، اشاره‌گر ثابت به یک ثابت cpc :

```
int n = 44;                  // an int
int* p = &n;                  // a pointer to an int
++(*p);                      // OK: increments int *p
++p;                        // OK: increments pointer p
int* const cp = &n;           // a const pointer to an int
++(*cp);                     // OK: increments int *cp
++cp;                       // illegal: pointer cp is const
const int k = 88;             // a const int
const int * pc = &k;          // a pointer to a const int
++(*pc);                     // illegal: int *pc is const
++pc;                       // OK: increments pointer pc
const int* const cpc = &k;   // a const pointer to a const int
++(*cpc);                    // illegal: int *pc is const
++cpc;                      // illegal: pointer cpc is const
```

اشاره‌گر p اشاره‌گری به متغیر n است. هم خود p قابل افزایش است (p++) و هم مقداری که p به آن اشاره می‌کند قابل افزایش است ((P*)++). اشاره گر cp یک اشاره‌گر ثابت است. یعنی آدرسی که در cp است قابل تغییر نیست ولی مقداری که در آن آدرس است را می‌توان دست‌کاری کرد. اشاره‌گر pc اشاره‌گری است که به آدرس یک ثابت اشاره دارد. خود pc را می‌توان تغییر داد ولی مقداری که pc به آن اشاره دارد قابل تغییر نیست. در آخر هم cpc یک اشاره‌گر ثابت به یک شیء ثابت است. نه مقدار cpc قابل تغییر است و نه مقداری که آدرس آن در cpc است.

آرایه‌ای از اشاره‌گرها

می‌توانیم آرایه‌ای تعریف کنیم که اعضای آن از نوع اشاره‌گر باشند. مثلا دستور:

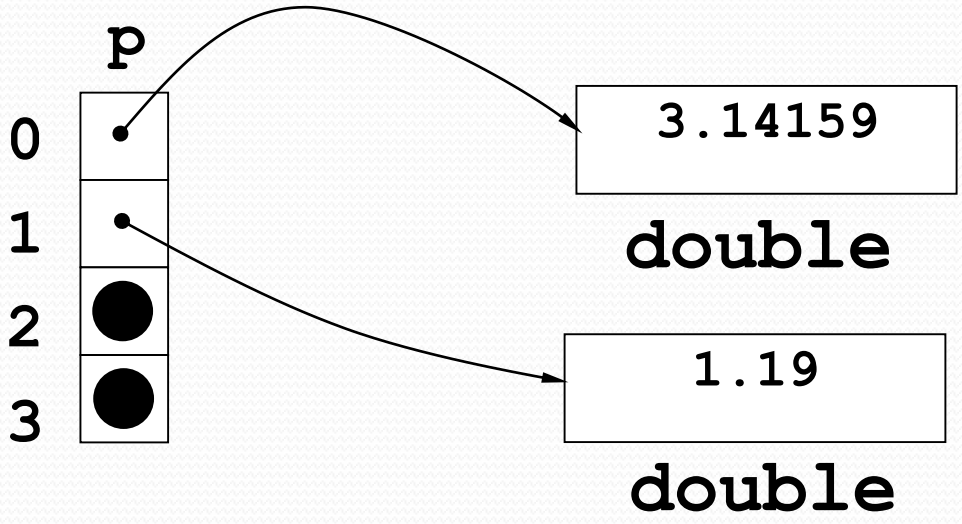
```
float* p[4];
```

آرایهٔ **p** را با چهار عنصر از نوع ***float** (یعنی اشاره‌گری به **float**) اعلان می‌کند. عناصر این آرایه را مثل اشاره‌گرهای معمولی می‌توان مقداردهی کرد:

```
p[0] = new float(3.14159);
p[1] = new float(1.19);
```

این آرایه را می توانیم شبیه شکل مقابل مجسم کنیم:

مثال بعد نشان می‌دهد که آرایه‌ای از اشاره‌گرها به چه دردی می‌خورد. از این آرایه می‌توان برای مرتب‌کردن یک فهرست نامرتب به روش حبابی استفاده کرد. به جای این که خود عناصر جابجا شوند، اشاره‌گرهای آن‌ها جابجا می‌شوند.



مرتب‌سازی حبابی غیرمستقیم

```
void sort(float* p[], int n)
{  float* temp;
   for (int i = 1; i < n; i++)
      for (int j = 0; j < n-i; j++)
         if (*p[j] > *p[j+1])
         {  temp = p[j];
            p[j] = p[j+1];
            p[j+1] = temp;
         }
}
```

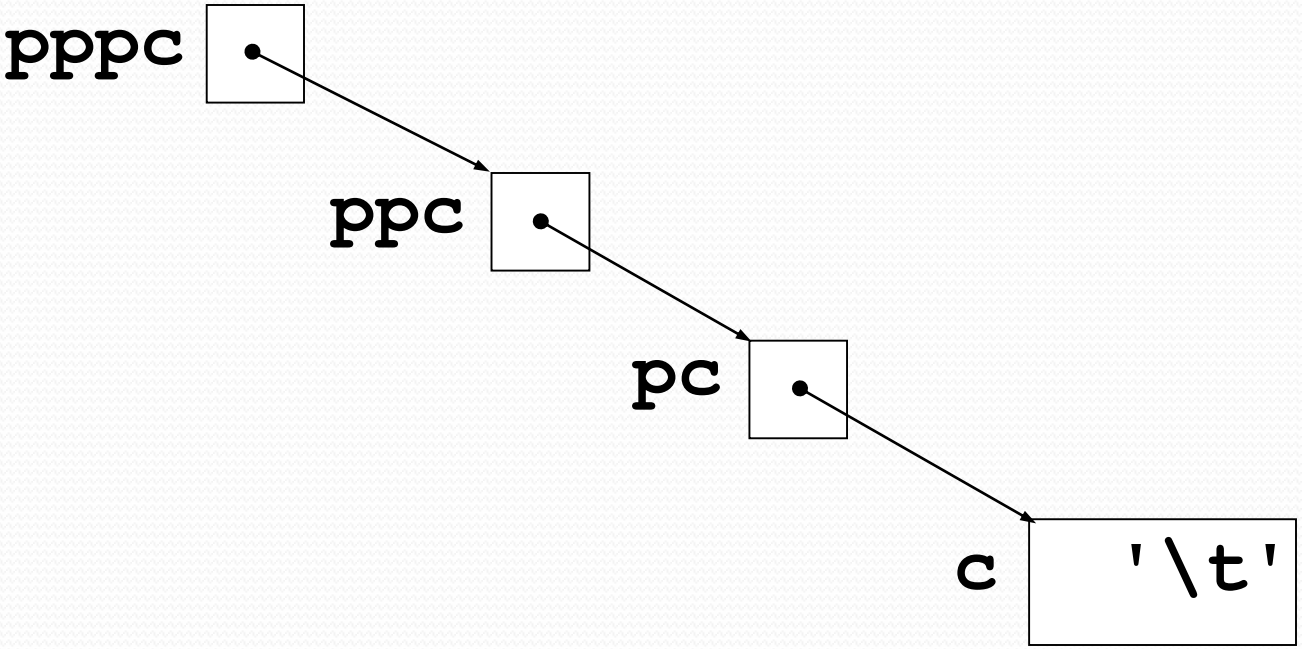
تابع **sort()** آرایه‌ای از اشاره‌گرها را می‌گیرد. سپس درون حلقه‌های تودرتوی **for** بررسی می‌کند که آیا مقادیری که اشاره‌گرهای مجاور به آن‌ها اشاره دارند، مرتب هستند یا نه. اگر مرتب نبودند، جای اشاره‌گرهای آن‌ها را با هم عوض می‌کند. در پایان به جای این که یک فهرست مرتب داشته باشیم، آرایه‌ای داریم که اشاره‌گرهای درون آن به ترتیب قرار گرفته اند.

اشاره‌گری به اشاره‌گر دیگر

یک اشاره‌گر می‌تواند به اشاره‌گر دیگری اشاره کند. مثلا:

```
char c = 't';
char* pc = &c;
char** ppc = &pc;
char*** pppc = &ppc;
***pppc = 'w';   // changes value of c to 'w'
```

حالا pc اشاره‌گری به متغیر کاراکتری c است. ppc اشاره‌گری به اشاره‌گر pc است و اشاره‌گر pppc هم به اشاره‌گر ppc اشاره دارد. مثل شکل مقابل:



با این وجود می‌توان با اشاره‌گر **pppc** مستقیما به متغیر **c** رسید.

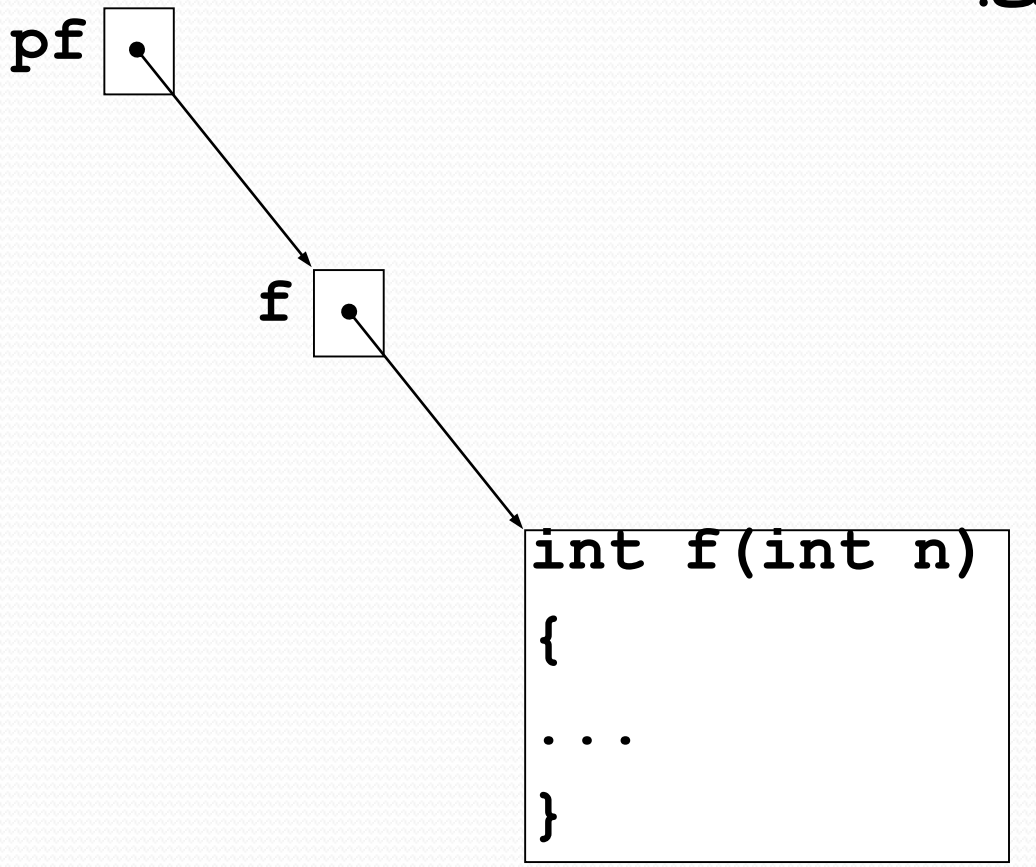
اشاره‌گر به توابع

این بخش ممکن است کمی عجیب به نظر برسد. حقیقت این است که نام یک تابع مثل نام یک آرایه، یک اشاره‌گر ثابت است. نام تابع، آدرسی از حافظه را نشان می‌دهد که کدهای درون تابع در آن قسمت جای گرفته‌اند. پس بنابر قسمت قبل اگر اشاره‌گری به تابع اعلان کنیم، در اصل اشاره‌گری به اشاره‌گر دیگر تعریف کرده‌ایم. اما این تعریف، نحو متفاوتی دارد:

```
int f(int);       // declares function f
int (*pf)(int);  // declares function pointer pf
pf = &f;          // assigns address of f to pf
```

اشاره‌گر pf همراه با * درون پرانتز قرار گرفته، یعنی این که pf اشاره‌گری به یک تابع است. بعد از آن یک int هم درون پرانتز آمده است، به این معنی که تابعی که pf به آن اشاره می‌نماید، پارامتری از نوع int دارد. اشاره‌گر pf را می‌توانیم به شکل زیر تصور کنیم:

فایدهٔ اشاره‌گر به توابع این است که به این طریق می‌توانیم توابع مرکب بسازیم. یعنی می‌توانیم یک تابع را به عنوان آرگومان به تابع دیگر ارسال کنیم! این کار با استفاده از اشاره‌گر به تابع امکان پذیر است.



تابع مرکب جمع

تابع **sum()** در این مثال دو پارامتر دارد: اشاره‌گر تابع **pf** و عدد صحیح **n** :

```
int sum(int (*)(int), int);
int square(int);
int cube(int);
int main()
{  cout << sum(square,4) << endl; // 1 + 4 + 9 + 16
   cout << sum(cube,4) << endl; //1 + 8 + 27 + 64
}
```

تابع sum() یک پارامتر غیر معمول دارد. نام تابع دیگری به عنوان آرگومان به آن ارسال شده. هنگامی که sum(square,4) فراخوانی شود، مقدار square(1)+square(2)+square(3)+square(4) بازگشت داده می‌شود. چون square(k) مقدار k*k را برمی‌گرداند، فراخوانی sum(square,4) مقدار 1+4+9+16=30 را محاسبه نموده و بازمی‌گرداند. تعریف توابع و خروجی آزمایشی به شکل زیر است:

```
int sum(int (*pf)(int k), int n)
{  // returns the sum f(0) + f(1) + f(2) + ... + f(n-1):
   int s = 0;
   for (int i = 1; i <= n; i++)
      s += (*pf)(i);
   return s;
}
int square(int k)
{  return k*k;
}
int cube(int k)
{  return k*k*k;
}
```

```
30
100
```

pf در فهرست پارامترهای تابع **sum()** یک اشاره‌گر به تابع است. اشاره‌گر به تابعی که آن تابع پارامتری از نوع **int** دارد و مقداری از نوع **int** را برمی‌گرداند. **k** در تابع **sum** اصلا استفاده نشده اما حتما باید قید شود تا کامپایلر بفهمد که **pf** به تابعی اشاره دارد که پارامتری از نوع **int** دارد. عبارت **(*pf)(i)** معادل با **square(i)** یا **cube(i)** خواهد بود، بسته به این که کدام یک از این دو تابع به عنوان آرگومان به **sum()** ارسال شوند.

نام تابع، آدرس شروع تابع را دارد. پس **square** آدرس شروع تابع **square()** را دارد. بنابراین وقتی تابع **sum()** به شکل **sum(square,4)** فراخوانی شود، آدرسی که درون **square** است به اشاره‌گر **pf** فرستاده می‌شود. با استفاده از عبارت **(*pf)(i)** مقدار **i** به آرگومان تابعی فرستاده می‌شود که **pf** به آن اشاره دارد.

NUL و NULL

ثابت صفر (**0**) از نوع **int** است اما این مقدار را به هر نوع بنیادی دیگر می‌توان تخصیص داد:

```
char c = 0;         // initializes c to the char '\0'
short d = 0;        // initializes d to the short int 0
int n = 0;          // initializes n to the int 0
unsigned u = 0; // initializes u to the unsigned int 0
float x = 0;        // initializes x to the float 0.0
double z = 0;       // initializes z to the double 0.0
```

مقدار صفر معناهای گوناگونی دارد. وقتی برای اشیای عددی به کار رود، به معنای عدد صفر است. وقتی برای اشیای کاراکتری به کار رود، به معنای *کاراکتر تهی* یا NUL است. NUL معادل کاراکتر '\0' نیز هست. وقتی مقدار صفر برای اشاره‌گرها به کار رود، به معنای «هیچ چیز» یا NULL است. NULL یک کلمهٔ کلیدی است و کامپایلر آن را می‌شناسد. هنگامی که مقدار NULL یا صفر در یک اشاره‌گر قرار می‌گیرد، آن اشاره‌گر به خانه 0x0 در حافظه اشاره دارد. این خانهٔ حافظه، یک خانهٔ استثنایی است که قابل پردازش نیست. نه می‌توان آن خانه را مقداریابی کرد و نه می‌توان مقداری را درون آن قرار داد. به همین دلیل به NULL «هیچ چیز» می‌گویند.

وقتی اشاره‌گری را بدون استفاده از **new** اعلان می‌کنیم، خوب است که ابتدا آن را **NULL** کنیم تا مقدار زبالهٔ آن پاک شود. اما همیشه باید به خاطر داشته باشیم که اشاره‌گر **NULL** را نباید مقداریابی نماییم:

```
int* p = 0;     // p points to NULL
*p = 22;        // ERROR: cannot dereference the NULL pointer
```

پس خوب است هنگام مقداریابی اشاره‌گرها، احتیاط کرده و بررسی کنیم که آن اشاره‌گر **NULL** نباشد:

```
if (p) *p = 22;   // O.K.
```

حالا دستور ***p=22;*** وقتی اجرا می‌شود که **p** صفر نباشد. می‌دانید که شرط بالا معادل شرط زیر است:

```
if (p != NULL) *p = 22;
```

اشاره‌گرها را نمی‌توان نادیده گرفت.

آن‌ها سرعت پردازش را زیاد می‌کنند و کدنویسی را کم.

با استفاده از اشاره‌گرها می‌توان به بهترین شکل از حافظه استفاده کرد.

با به کارگیری اشاره‌گرها می‌توان اشیایی پیچیده‌تر و کارآمدتر ساخت.