درس «مبانی کامپیوتر و برنامه‌سازی»

اشاره‌گر

صادق علی‌اکبری

سرفصل مطالب

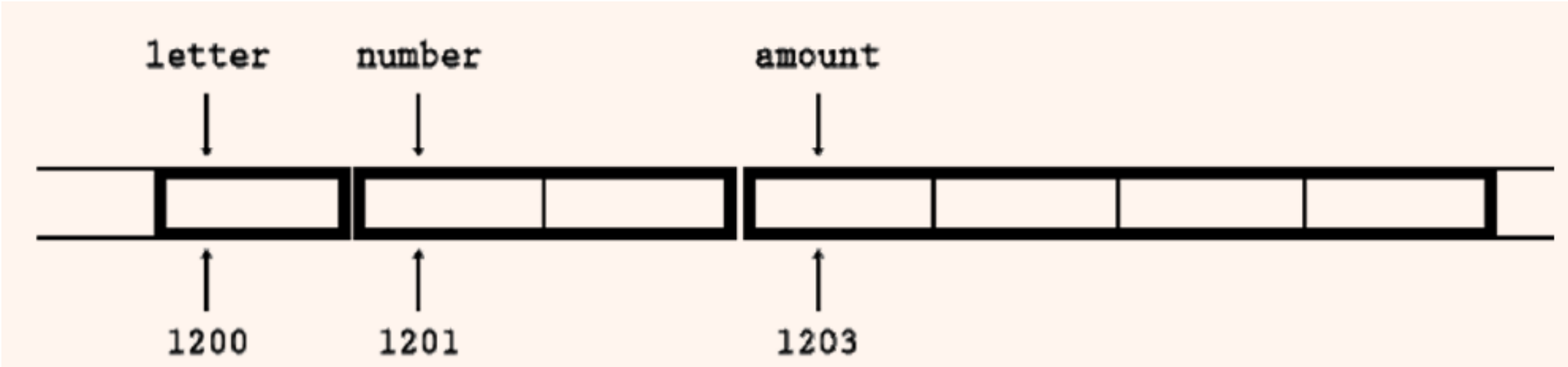
- مفهوم اشاره‌گر
- آرایه و اشاره‌گر
- ارسال پارامتر با کمک اشاره‌گر
- اشاره‌گر به تابع
- تخصیص حافظه پویا

آدرس حافظه

- هر بایت از حافظه دارای آدرسی منحصر به فرد است
  - برای کار با حافظه، باید آدرس خانه موردنظر مشخص شود
  - مثال: CPU به RAM دستور می‌دهد در بایتی با **آدرس** 0xFFA05B مقدار 8 را بنویس
- هر متغیر بخشی از حافظه را در زمان اجرا اشغال خواهد کرد
- آدرس هر متغیر، آدرس اولین بایتی از حافظه است که به آن متغیر اختصاص داده شده است

مثال

```
char letter = 'A';
short int number = 12;
float amount = 12.5;
```



دریافت آدرس یک متغیر

- با کمک عملگر **&** می‌توانیم به آدرس یک متغیر دسترسی پیدا کنیم
- یک آدرس، یک عدد است
- مثال:

```
int number = 12;
float amount = 12.5;

cout<<number<<endl;
cout<<&number<<endl;

cout<<amount<<endl;
cout<<&amount<<endl;
```

```
12
0x28ff2c
12.5
0x28ff28
```

- روی محل و آدرس یک متغیر نمی‌توانیم حساب کنیم
  - مگر در مواردی خاص
  - (مثلاً در آرایه‌ها می‌دانیم a[1] بلافاصله بعد از a[0] ذخیره می‌شود)

مفهوم اشاره‌گر (Pointer)

- یک آدرس حافظه، یک عدد است
  - این عدد را می‌توانیم در یک متغیر مستقل نگهداری کنیم
- به عنوان مثال، آدرس حافظه متغیر x را می‌توانیم در متغیر y نگهداری کنیم
- در این صورت می‌گوییم:
  - متغیر y ، آدرس حافظه متغیر x را نگه می‌دارد
  - متغیر y ، به آدرس حافظه متغیر x **اشاره** می‌کند
  - متغیر y ، به متغیر x **اشاره** می‌کند
- در مثال فوق، y یک **اشاره‌گر** (pointer) است

تعریف اشاره‌گر

- برای تعریف متغیری که به عنوان اشاره‌گر استفاده می‌شود، از * استفاده می‌کنیم

- مثال:

```
int number = 12;
float amount = 12.5;
int* pnum = &number;
float* pamount = &amount;
```

- در مثال فوق:
  - pnum یک اشاره‌گر است که قرار است به یک متغیر int اشاره کند
  - pamount یک اشاره‌گر است که قرار است به یک متغیر float اشاره کند

```
pamount = & number;//syntax error
pnum = & amount;//syntax error
```

دسترسی به یک متغیر از طریق اشاره‌گر

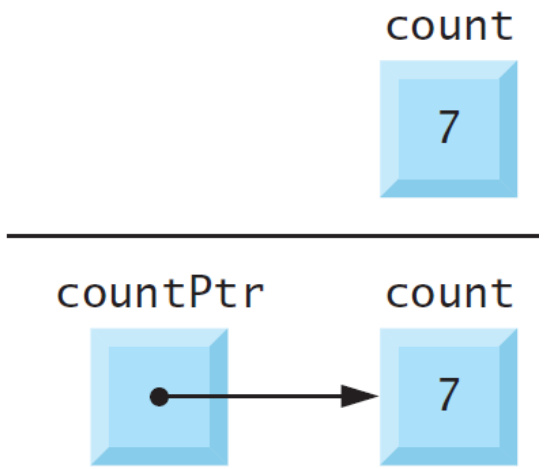
- با داشتن یک اشاره‌گر، به محتوای محلی که به آن اشاره می‌شود، دسترسی داریم
- برای این کار، از عملگر * استفاده می‌شود
- عملگر * به صورت معکوس عملگر & کار می‌کند

```
cout <<number<<endl;
cout <<&number<<endl;
cout <<*(&number)<<endl;
```

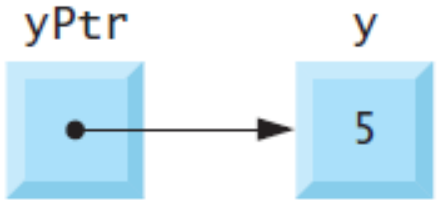
```
12
0x28ff1c
12
```

مثال

- int count = 7;
- int *countPtr = &count;



- int y = 5;

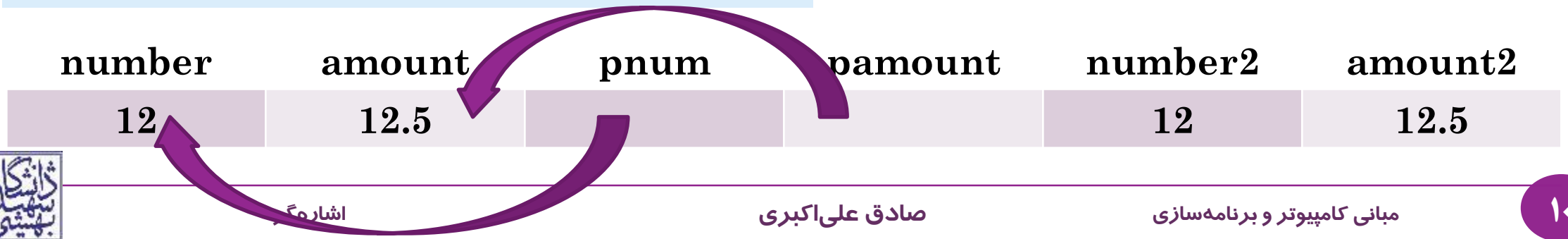


- int *yPtr ;
- yPtr = &y;

مثال

```
int number = 12;
float amount = 12.5;
int* pnum = &number;
float* pamount = &amount;
int number2 = *pnum;
float amount2 = *pamount;

cout <<pnum <<endl;
cout <<*pnum <<endl;
cout <<number2 <<endl;

cout <<pamount<<endl;
cout <<*pamount<<endl;
cout <<amount2<<endl;
```

```
0x28ff1c
12
12

0x28ff18
12.5
12.5
```



نکته (۱)

- اشاره‌گرها خود متغیر هستند و در حافظه جای می‌گیرند
  - بنابراین محلی در حافظه دارند
- آدرس یک اشاره‌گر را می‌توانیم در یک اشاره‌گر دیگر نگهداری کنیم

```
int* pnum = &number;
int** pointer2pointer = &pnum;
```

  - متغیر **pointer2pointer** یک اشاره‌گر به یک اشاره‌گر دیگر است
  - متغیر **pointer2pointer** یک اشاره‌گر به `int*` است، پس `int**` است

نکته (۲)

- یک اشاره‌گر، به اولین بایت از حافظه یک متغیر اشاره میکند
  - یعنی آدرس اولین بایت را نگهداری می‌کند
- طول حافظه اشاره‌گر به نوع اشاره‌گر وابسته نیست
- یک int* همان‌قدر حافظه می‌گیرد که یک double* می‌گیرد (مثلاً چهار بایت)

مثال

عملگر sizeof طول حافظه یک متغیر (یا نوع داده) را برمی‌گرداند

```
short int number = 12;
double amount = 12.5;

short int* pnum = &number;
short int** pointer2pointer = &pnum;
double* pamount = &amount;

cout << sizeof(number)<<endl;
cout << sizeof(amount)<<endl;
cout << sizeof(pnum)<<endl;
cout << sizeof(pamount)<<endl;
cout << sizeof(pointer2pointer)<<endl;

cout << sizeof(double)<<endl;
cout << sizeof(double*)<<endl;
cout << sizeof(double**)<<endl;
```

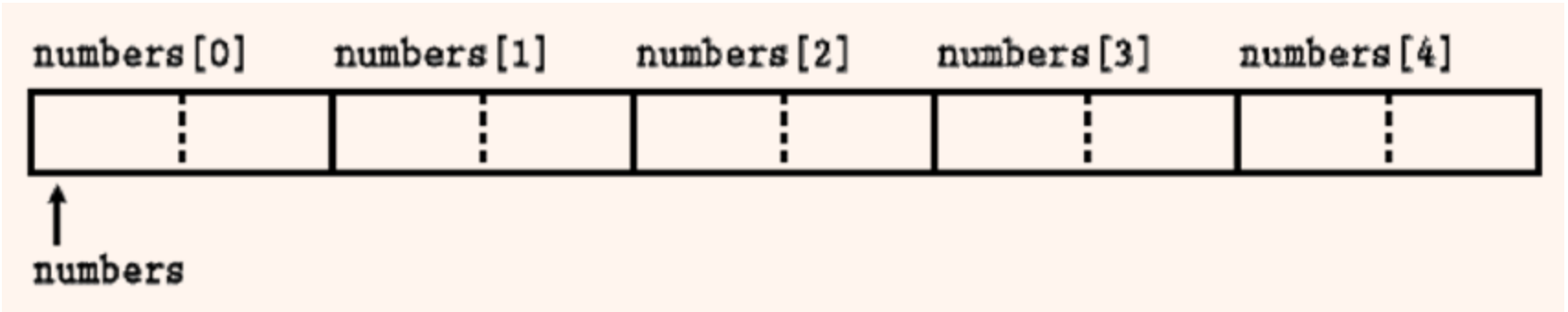
```
2
8
4
4
4

8
4
4
```

اشاره‌گر و آرایه

- نام آرایه، مانند یک **اشاره‌گر ثابت** به اولین عنصر آرایه عمل می‌کند
- از نام آرایه، مانند یک اشاره‌گر می‌توانیم استفاده کنیم
- مثال:

```
double a[]= {2.7, 3.14, 10};
cout << *a <<endl;
double d = *a;
cout << d <<endl;
double* p = a;
cout << p <<endl;
cout << *p <<endl;
```

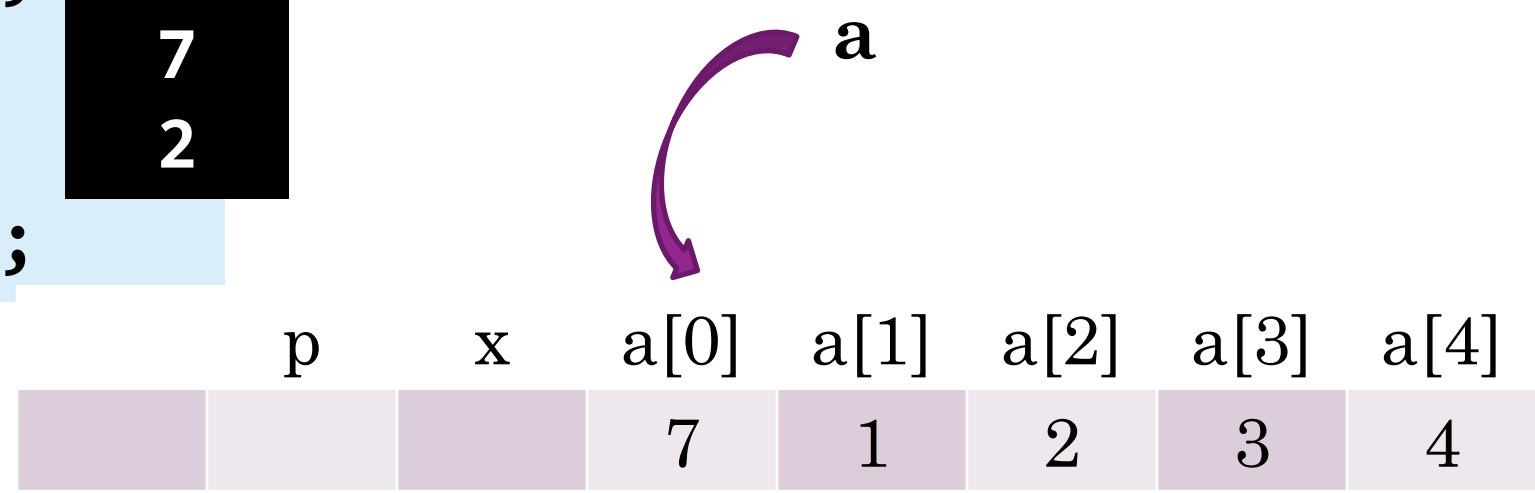
```
2.7
2.7
0x28ff08
2.7
```



مثال

```
int a[] = {7,1,2,3,4};
int * p;
p = a;
int x = *p;
cout << x << endl;
p = &a[0];
x = *p;
cout << x << endl;
p = &a[2];
x = *p;
cout << x << endl;
```

```
7
7
2
```



نکته

- مقدار اشاره‌گر قابل تغییر است، ولی مقدار آرایه قابل تغییر نیست
- آرایه همانند یک «**اشاره‌گر ثابت**» است

```
double a[]= {2.7, 3.14, 10};
double* p ;
double PI = 3.14;

p = a;
p = &PI;
```

```
a = p;
a = &PI;
```

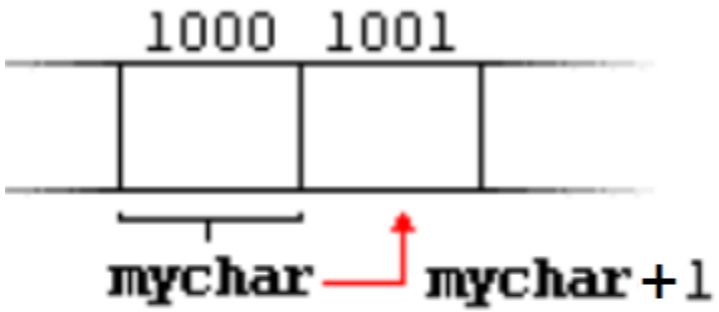
خطای کامپایل

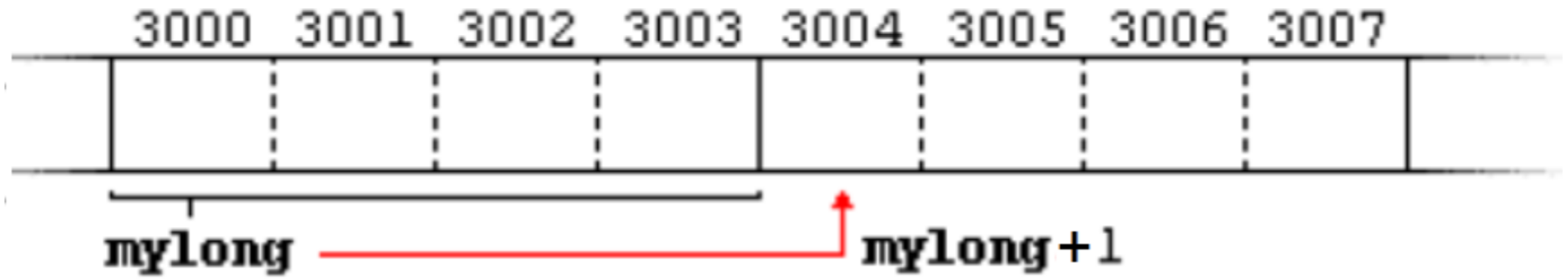
جمع و تفریق اشاره‌گرها

- یک اشاره‌گر، مقداری مانند یک عدد صحیح نگهداری می‌کند
- اما فقط عملگرهای جمع و تفریق برای اشاره‌گرها قابل استفاده هستند
  - مثلاً ضرب و تقسیم برای اشاره‌گر تعریف نشده است
- جمع و تفریق برای اشاره‌گرها به نحوه خاصی اجرا می‌شوند
  - جمع و تفریق معمولی نیست
- طول نوع متغیری که به آن اشاره می‌شود، در عملیات جمع و تفریق مؤثر است
- مثلاً اگر p یک double* باشد، p+1 یعنی اشاره‌گر به double بعدی
  - p+1 به معنی اشاره‌گر به بایت بعدی نیست، اشاره‌گر به ۸ بایت بعد از p است

جمع و تفریق اشاره‌گرها (ادامه)

- فرض کنید mychar از نوع char و mylong از نوع long باشد
- همچنین فرض کنید sizeof(char)=1 و sizeof(long)=4
- اگر عدد N با اشاره‌گری از نوع X جمع شود، در واقع N*sizeof(X) با عدد اشاره‌گر جمع می‌شود





مثال

```
double a[3];
a[0] = 1.5;
a[1] = 2.5;
a[2] = 3.5;
double* p = a;
cout << p << endl;
cout << p + 1 << endl;
cout << p + 2 << endl;
cout << *p << endl;
cout << *(p + 1) << endl;
cout << *(p + 2) << endl;

int index = 2;
cout<<a[index]<<endl;
cout<<*(a+index)<<endl;
cout<<*(p+index)<<endl;
```

```
0x28ff10
0x28ff18
0x28ff20
1.5
2.5
3.5

3.5
3.5
3.5
```

- بنابراین موارد زیر معادل هم هستند:

```
a[index]
*(a+index)
*(p+index)
```

مثال

```
long a[] = {1,2,3,4};
long* p = a;
cout<<*p<<endl;
cout<<*(p+1)<<endl;
cout<<p[2]<<endl;
p++;
cout<<*p<<endl;
cout<<p[2]<<endl;
p=a;
for(int i=0;i<4;i++, p++)
    cout<<*p<<endl;
```

```
1
2
3
2
4
1
2
3
4
```

- خروجی قطعه‌برنامه زیر چیست؟

- نکته: از اشاره‌گر هم مانند یک آرایه می‌توان استفاده کرد

توابعی با نوع برگشتی اشاره‌گر

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;

#define SIZE 10
int * random_array() {
    static int r[SIZE];
    int i;
    srand(time(0));
    for ( i = 0; i < SIZE; ++i)
        r[i] = rand();
    return r;
}
int main() {
    int* array = random_array();
    for(int i=0;i<SIZE;i++)
        cout<<array[i]<<endl;
}
```

- خروجی (مقدار برگشتی) یک تابع، ممکن است از نوع اشاره‌گر باشد
- مثال:
- در این مثال، اگر r استاتیک نبود، چه می‌شد؟
- کار خوبی نیست که اشاره‌گر به یک متغیر محلی غیراستاتیک را برگردانیم

اشاره‌گر وحشی و اشاره‌گر سرگردان و اشاره‌گر NULL

- **اشاره‌گر وحشی (wild pointer)**
  - اشاره‌گری که به عنوان یک متغیر محلی، هنوز مقدار نگرفته است (مقداردهی نشده است)
  - مقدار آن مشخص نیست، معلوم نیست به کجا اشاره می‌کند
  - به یک آدرس از حافظه اشاره می‌کند، ولی شماره آدرس قابل پیش‌بینی نیست
  - استفاده از محتوای یک اشاره‌گر وحشی: (احتمالاً) خطا در زمان اجرا
- **اشاره‌گر سرگردان (dangling pointer)**
  - چنین اشاره‌گری، به آدرسی اشاره می‌کند که قبلاً متعلق به یک متغیر بوده است، ولی الان متغیر موردنظر از آن آدرس حذف شده است
- **اشاره‌گر NULL** : به هیچ آدرس مشخصی از حافظه اشاره نمی‌کند
  - مقدار عددی داخل این متغیر، صفر است

مثال

- خروجی؟

```
int *global;                                   NULL Pointer
int * f() {
   int x = 7;
   return &x;
}
void g() {int t[] = {1,2,3,4,5};}
int main() {
   int *ptr = f();                             Dangling Pointer
   g();//called to overwrite local variables of f()
   cout<< *ptr <<endl;
   int* p2;                                    Wild Pointer
   cout << p2 << endl;

   int* np = NULL;                             NULL Pointer
   cout << global << endl;
   cout << np << endl;
   if(np!=NULL){...}
}
```

```
5
0x28ff68
0
0
```

مفهوم ارجاع

- در بحث ارسال پارامتر با ارجاع (call by reference)
  با مفهوم «ارجاع» (reference) آشنا شدیم
- در حالت عادی (نه فقط به صورت پارامتر) هم می‌توانیم ارجاع تعریف کنیم
- مثال: در کد روبرو، y اسم مستعاری برای x است
  - هر کاری با y بکنیم، انگار با x کرده‌ایم
  - علامت & در این‌جا به معنی «آدرس» نیست
    (همان‌طور که * در int* a; به معنای «محتوای آدرس» نیست)

```
int x = 5;
int&y = x;
y = 3;
cout<<x<<endl;
cout<<y<<endl;
```

```
3
3
```

- نکته مهم: در زبان C مفهوم ارجاع (reference) وجود ندارد
  - این مفهوم مخصوص C++ است

مفهوم ارجاع در مقایسه اشاره‌گر

```
int a = 2;
int&r = a;
int*p = &a;
```

- ارجاع مانند یک اشاره‌گر محدود و ساده است
  - کامپایلرها هم معمولاً مفهوم ارجاع را با کمک مفهوم اشاره‌گر پیاده می‌کنند
- تفاوت‌ها:
  - عملگرهای جمع و تفریق برای اشاره‌گر تعریف شده، ولی برای ارجاع تعریف نشده است.
  - برای استفاده از r احتیاج به & و * نیست
  - برای این که p به a اشاره کند، باید از &a استفاده کنیم
    و برای دسترسی به محتوای p ، از *p استفاده کنیم
  - مفهوم اشاره‌گر به اشاره‌گر داریم، ولی مفهوم ارجاع به ارجاع نداریم
  - اشاره‌گر می‌تواند NULL باشد، ولی ارجاع نمی‌تواند.
  - اشاره‌گر می‌تواند وحشی باشد، ولی ارجاع نمی‌تواند (مثلاً int*a; صحیح است ولی int&a; غلط)

خلاصه:
ارجاع امکانی امن‌تر اما
محدودتر از اشاره‌گر است

نکته

- «عملگر & و یک متغیر» نمی‌توانند سمت چپ انتساب قرار گیرند
- اما «عملگر * و یک متغیر» می‌توانند سمت چپ انتساب قرار گیرند
- مثال:

```
long a = 2;
long* p = &a;

//&p = 2; ➔ Syntax Error

*p = 5;
cout<<a;
```

5

ارسال با اشاره‌گر

- یادآوری: سه روش ارسال پارامتر به تابع وجود دارد
  - ارسال با مقدار (call by value)
  - ارسال با ارجاع (call by reference)
  - ارسال با اشاره‌گر (call by pointer)
- با دوتای اول قبلاً آشنا شدیم

- ارسال با اشاره‌گر (call by pointer) یعنی آدرس یک متغیر را به تابع پاس کنیم
  - به این ترتیب امکان تغییر محتوای این متغیر (آرگومان) را هم خواهیم داشت

مثال

```
void swap(int*a, int*b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}
int main() {
   int x = 1, y = 2;
   swap(&x, &y);
   cout << x << endl;
   cout << y << endl;
}
```

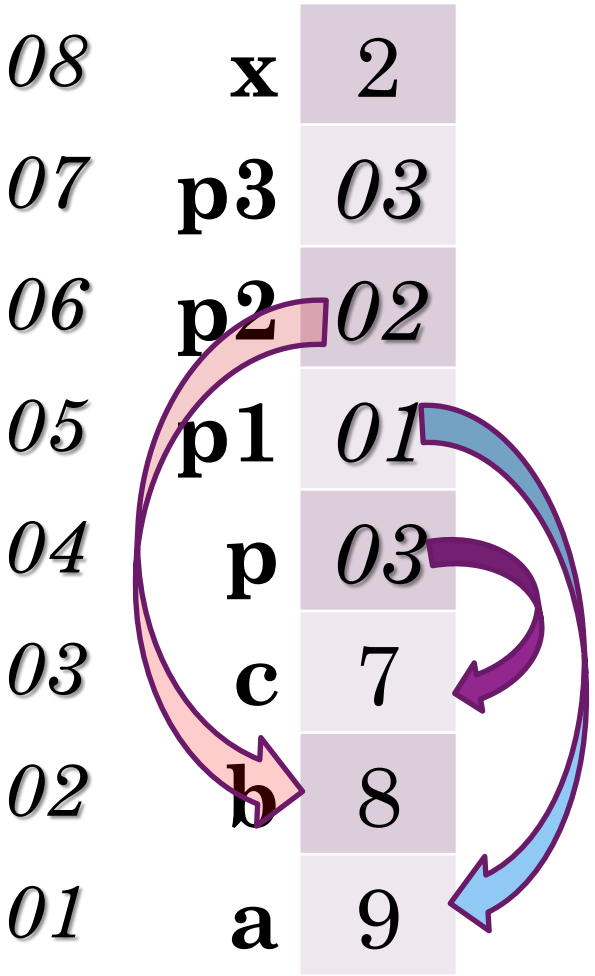
```
2
1
```

مثال

```
void fun(int byValue, int* byPointer, int& byRef){
   byValue = 1;
   *byPointer = 2;
   byRef = 3;
}

int main() {
   int a=9, b=8, c=7;
   fun(a, &b, c);
   cout<<a<<endl;
   cout<<b<<endl;
   cout<<c<<endl;
}
```

```
9
2
3
```

مثال




```
void fun(int* p1, int* p2, int* p3){
    *p1 = 1;
    int x =2;
    p2 = &x;
    p3 = &x;
}

int main() {
    int a=9, b=8, c=7;
    int*p = &c;
    fun(&a, &b, p);
    cout<<a<<endl;
    cout<<b<<endl;
    cout<<c<<endl;
}
```

```
1
8
7
```

ارسال با اشاره‌گر یا ارسال با ارجاع؟ کدام بهتر است؟

- ارسال با اشاره‌گر و ارسال با ارجاع، کاربرد مشابهی دارند
- در هر دو حالت، امکان تغییر متغیر آرگومان وجود دارد
- معمولاً پیاده‌سازی با کمک ارجاع ساده‌تر و امن‌تر است
- اگر هدف، فقط تغییر مقدار آرگومان است، ارسال با ارجاع بهتر است
- برخی اهداف پیچیده‌تر، با اشاره‌گر ممکن می‌شود
  - مثلاً تغییر مقدار حافظه‌ای که بعد از متغیر موردنظر قرار گرفته است
- در زبان C ، اصلاً ارجاع (و ارسال با ارجاع) وجود ندارد

کوییز

- خروجی؟

```
int* fun(int a, int*b, int*c, int& d){
   static int array[]={1,2,3,4};
   *b = 5;
   c= &a;
   *c = 6;
   d=7;
   b=&array[3];
   *b=8;
   int* result = array;
   return result;
}
```

```
int main() {
   int* p;
   int w = 9, x=10, y=11, z=12;
   p = fun(w,&x,&y,z);
   cout<<w<<endl;
   cout<<x<<endl;
   cout<<y<<endl;
   cout<<z<<endl;
   cout<<p[0]<<endl;
   cout<<p[3]<<endl;
}
```

```
9
5
11
7
1
8
```

اشاره‌گر به نوع نامعلوم (void*)

- یک اشاره‌گر را می‌توانیم از نوع void* تعریف کنیم
- با کمک چنین اشاره‌گری می‌توانیم به هر نوع داده‌ای اشاره کنیم
  - مثلاً int ، float و ...
- اما برای استفاده از محتوای محلی که به آن اشاره می‌شود، به نوع داده احتیاج داریم
  - راه‌حل: استفاده از type casting

```
int a =2;
double b =1.5;
bool c = true;
void* ptr;
ptr = &a;
int* x = (int*)ptr;
ptr = &b;
double* y = (double*)ptr;
ptr = &c;
bool* z = (bool*)ptr;

//ptr++; ➔ Syntax Error
//*ptr; ➔ Syntax Error
```

حافظه پویا

- برنامه‌هایی که تا به حال می‌نوشتیم، حافظه‌ای مشخص و معلوم ایجاد می‌کنند
  - در زمان کامپایل معلوم بود که چه متغیرهایی در زمان اجرا ایجاد می‌شوند
  - و اندازه (طول) هر متغیر چقدر است
  - حتی طول آرایه‌ها، در زمان کامپایل معلوم بود
    - `int a[10];` آرایه‌ای به طول ۱۰ می‌سازد:

      طول آرایه در زمان کامپایل و قبل از اجرای برنامه مشخص است
    - `int a[N]` به شرطی صحیح است که N یک ثابت و در زمان کامپایل مشخص باشد
    - اگر قرار باشد N یک متغیر معمولی باشد که مقدار آن در زمان اجرا مشخص می‌شود، چه؟
    - ← به تخصیص حافظه پویا (یعنی در زمان اجرا) نیازمندیم

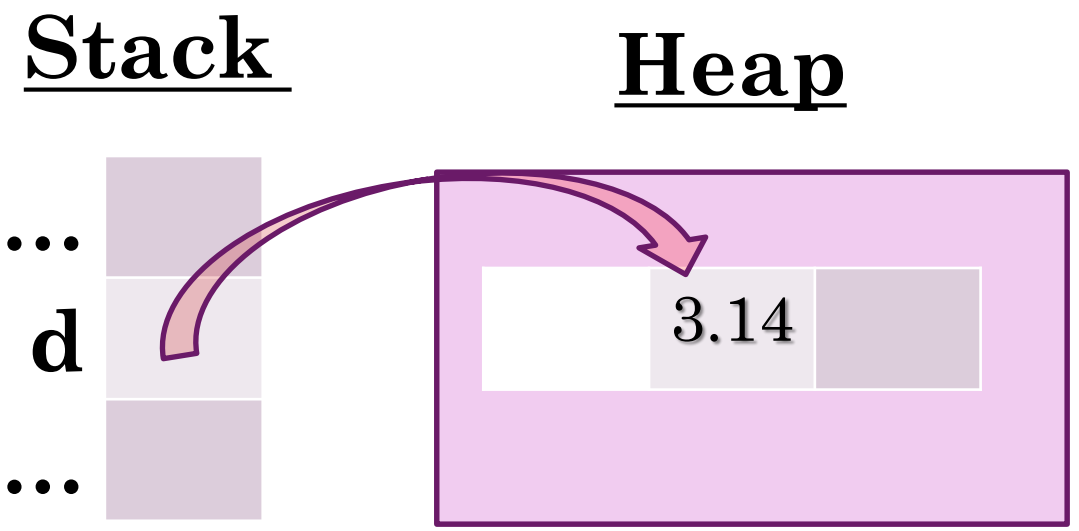
تخصیص حافظه پویا (Dynamic memory allocation)

- گاهی لازم است حافظه‌ای در زمان اجرا مشخص شود و تخصیص یابد
- مثلاً اگر بخواهیم آدرس یک فایل را از کاربر بگیریم و محتوای فایل را در یک آرایه بریزیم (طول فایل و در نتیجه اندازه آرایه در زمان اجرا مشخص می‌شود)
- تخصیص حافظه پویا در زبان C++ با کمک اشاره‌گرها و عملگر new انجام می‌شود
- ایجاد یک متغیر: pointer = new type ;
- ایجاد آرایه‌ای از متغیرها: pointer = new type [size] ;
- مثال:

```
double* d;
d = new double;
*d = 3.14;
```

```
int size;
cin >> size;
double* a;
a = new double[size];
a[2] = 3.14;
```

مثال




```
double* d;
d = new double;
*d = 3.14;
cout << *d << endl;
```

3.14

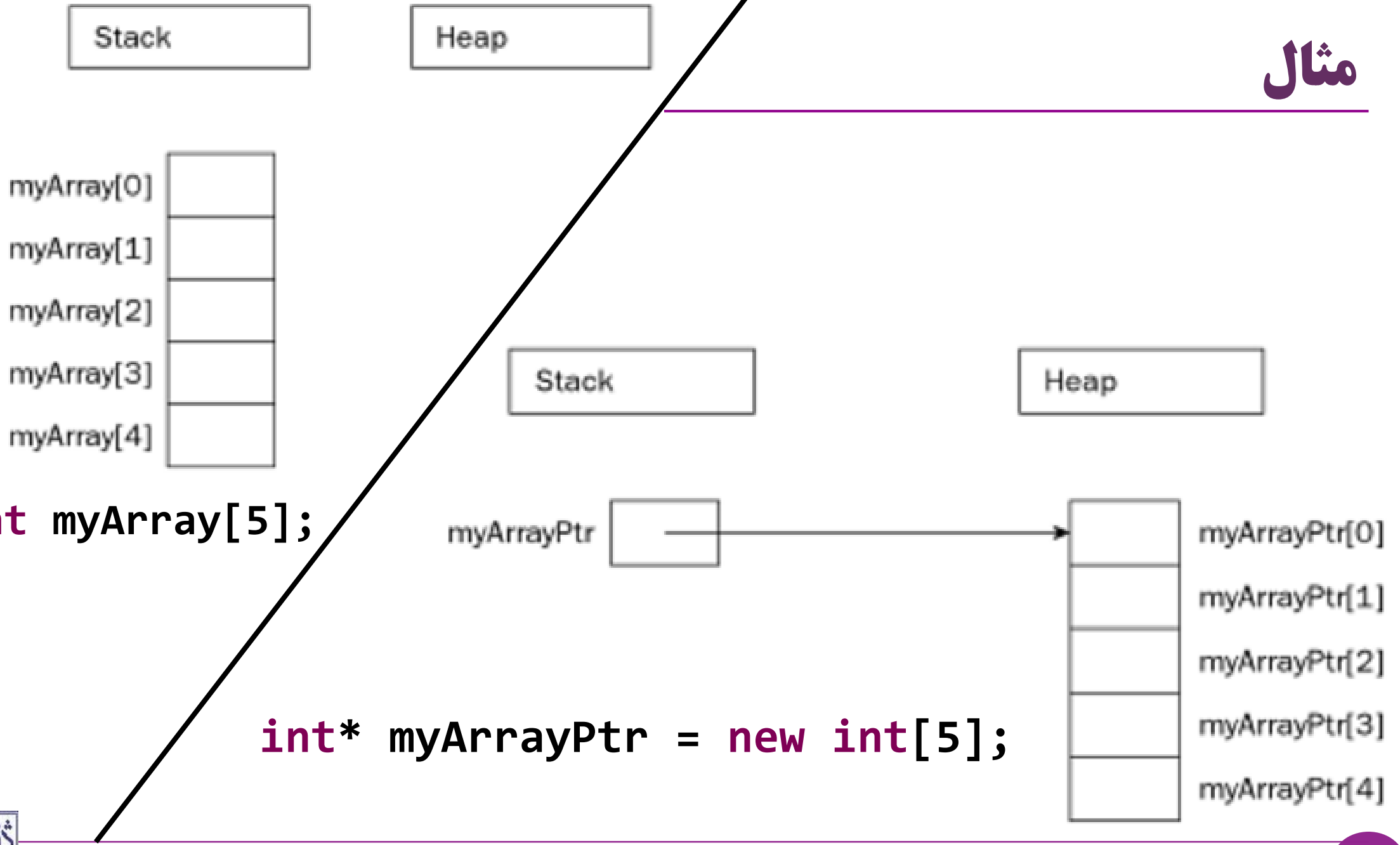
- نحوه عملکرد عملگر new :
  - یک بخش در حافظه اختصاص می‌دهد
  - اشاره‌گر به محل اختصاص یافته را برمی‌گرداند
  - حافظه اختصاص یافته در پشته (stack) نیست، بلکه در فضایی به نام Heap است
  - Stack و Heap هر دو در حافظه اصلی (RAM) هستند

مثال

```
void bubble_sort(double a[], int size) {
    for (int i = 0; i < size; i++)
        for (int j = 0; j < size - i - 1; j++)
            if (a[j + 1] < a[j])
                swap(a[j], a[j + 1]);
}
int main() {
   int size;
   cout<<"Enter array size:";
   cin >> size;
   double* a;
   a = new double[size];
   for (int i = 0; i < size; ++i){
      cout<<"Enter array["<<i<<"] : ";
      cin >> a[i];
   }
   bubble_sort(a, size);
   for (int i = 0; i < size; ++i)
      cout << a[i] << endl;
}
```

```
Enter array size:5
Enter array[0] : 1
Enter array[1] : 9
Enter array[2] : 3.14
Enter array[3] : 2.71
Enter array[4] : 99
1
2.71
3.14
9
99
```

مثال


Stack
Heap
myArray[0]
myArray[1]
myArray[2]
myArray[3]
myArray[4]
int myArray[5];
Stack
Heap
myArrayPtr
myArrayPtr[0]
myArrayPtr[1]
myArrayPtr[2]
myArrayPtr[3]
myArrayPtr[4]
int* myArrayPtr = new int[5];

آزادسازی حافظه پویا

- تخصیص حافظه پویا، توسط برنامه‌نویس تعیین می‌شود
  - مثلاً برنامه‌نویس تعیین می‌کند که در زمان اجرا باید آرایه‌ای با طول متغیر n ایجاد شود
- آزادسازی حافظه تخصیص‌یافته پویا هم باید توسط برنامه‌نویس انجام شود
- آزادسازی حافظه پویا، توسط **عملگر delete** انجام می‌شود
- با کمک این دستور، حافظه اختصاص یافته پویا آزاد می‌شود
  و فضا برای ذخیره حافظه جدید (با کمک new) باز می‌شود
- اگر بعد از اتمام نیاز به حافظه، آن را delete نکنیم، بعد از مدتی حافظه (Heap) پر می‌شود
- حذف (آزادسازی) یک متغیر: `delete pointer ;`
- حذف آرایه‌ای از متغیرها: `delete [ ] pointer ;`

مثال

- فرض کنید این برنامه را اجرا کنیم
  - و حافظه مصرفی آن را رصد کنیم
  - مثلاً با دستور top در لینوکس
    یا با کمک taskmgr در ویندوز

```
long* ptr = new long;
*ptr = 1625367736L;
cout << *ptr << endl;
delete ptr;
```

حافظه مصرفی برنامه تا این‌جا: حدود ۴۰۰ کیلوبایت

```
int n = 100000000;//100 milion
ptr = new long[n];
```

حافظه مصرفی برنامه تا این‌جا: حدود ۴۰۰ مگابایت

```
for (int i = 0; i < n; i++)
   ptr[i] = i;
cout << ptr[n/2] << endl;
delete[] ptr;
```

حافظه مصرفی برنامه تا این‌جا: حدود ۴۰۰ کیلوبایت

تخصیص حافظه ایستا

- روش عادی اختصاص حافظه، توسط کامپایلر تعیین می‌شود
- این روش static memory allocation خوانده می‌شود
- اختصاص حافظه و آزادسازی حافظه، خودکار است
- برای این کارها، نیازی به دستور برنامه‌نویس نیست
  - هرگاه در یک محدوده (scope) یک متغیر تعریف شود، حافظه برای آن در نظر گرفته می‌شود
  - هرگاه اجرای برنامه از یک محدوده خارج می‌شود، همه متغیرهای محلی آن محدوده آزاد می‌شوند
- مثال:

```
void f(int p){double d; long a[100];}
int main() {
   f(5);
}
```

  - با فراخوانی تابع f حافظه لازم برای سه متغیر محلی ایجاد می‌شود
  - با پایان اجرای این تابع، حافظه تخصیص‌یافته (در stack) آزاد می‌شود

تخصیص حافظه پویا

- تعیین میزان حافظه موردنیاز در زمان اجرا، توسط برنامه (برنامه‌نویس) تعیین می‌شود
- این روش dynamic memory allocation خوانده می‌شود
- اختصاص حافظه و آزادسازی حافظه، خودکار نیست
- برنامه‌نویس باید مشخص کند که چه زمانی حافظه گرفته شود، و چه زمانی آزاد شود
- کامپایلر نمی‌تواند بفهمد از چه زمانی دیگر نیازی به حافظه تخصیص‌یافته نیست
- نکته:

با پایان محدوده (scope) یک حافظه پویا، لزوماً زمان استفاده از آن پایان نیافته است

مثال

```
int* random_array(int size) {
    int* a = new int[size];
    for (int i = 0; i < size; ++i)
        a[i] = rand();
    return a;
}
void bubble_sort(int a[], int size) {
    for (int i = 0; i < size; i++)
        for (int j = 0; j < size - i - 1; j++)
            if (a[j + 1] < a[j])
                swap(a[j], a[j + 1]);
}
int main() {
    srand(time(0));
    int n;
    cin >> n;
    int* randoms = random_array(n);
    bubble_sort(randoms, n);
    cout << randoms[0] << endl;
    cout << randoms[n - 1] << endl;
    delete[] randoms;
}
```

- تابع random_array یک آرایه پویا ایجاد می‌کند و برمی‌گرداند
- اما با پایان اجرای این تابع ، نباید حافظه مربوط به این آرایه حذف (آزاد) شود
- بنابراین زمان آزادسازی حافظه پویا باید توسط برنامه‌نویس مشخص شود

نشت حافظه

- اگر حافظه گرفته شده را آزاد نکنیم، حافظه هدر می‌رود
  - و فضا برای استفاده در ادامه برنامه، توسط سایر متغیرها، فراهم نمی‌شود
- به این شرایط **نشت حافظه** (**memory leak**) گفته می‌شود
  - وقتی رخ می‌دهد که متناظر با هر new از delete استفاده نکنیم
- مثال:

```
void memLeak( )
{
  int *data = new int;
  *data = 15;
}
```

```
void memLeak(){
  int *data = new int[100];
  *data = 15;
  data[1] = 16;
  data = new int;
  delete data;
}
```

وضع اشاره‌گر بعد از عملگر delete

- بعد از اجرای delete روی یک اشاره‌گر، تبدیل به یک اشاره‌گر سرگردان می‌شود
- Dangling pointer
- مقدار اشاره‌گر براثر delete تغییر نمی‌کند

```
int* p = new int;
*p = 1;
cout << p << endl;
cout << *p << endl;
delete p;
//p is now a dangling pointer
...
```

- در انتهای کد فوق، p به جایی از حافظه اشاره می‌کند که دیگر حذف (آزاد) شده است

اجرای اشتباه delete

- اجرای delete روی اشاره‌گر null بدون خطا و بی‌تأثیر است
- اجرای delete روی یک اشاره‌گر غلط، خطرناک است
  - مثلاً اجرای delete روی یک wild pointer یا dangling pointer
  - اشاره‌گری که قبلاً delete شده، یا اصلاً مقداردهی نشده
- البته رفتار C++ در این شرایط اصطلاحاً تعریف‌نشده (undefined) است
  - شاید خطا ندهد، ولی به هرحال این کار غلط و خطرناک است
  - حتی ممکن است ناخواسته به آزادسازی حافظه دیگری منجر شود، که قرار نیست آزاد شود

مثال

```
float* a ;
float* b;

a = new float;
*a = 3.14;

*b = 2.71;

b = a;
*b = 2.71;

delete b;

delete a;
```

پیچیدگی آزادسازی حافظه پویا

- آزادسازی حافظه پویا، کاری مستعد خطا (error prone) است
- اشتباه برنامه‌نویسی در این زمینه بسیار رایج است
- کشف خطا و رفع اشکال در این زمینه هم مشکل است
- در این زمینه باید دقت بیشتری داشته باشیم
  - شرایط مختلف اجرای برنامه و حالت‌های مختلف برنامه را در نظر بگیریم
  - شرایط مختلف را تست کنیم
- در برخی زبان‌های جدید، امکاناتی برای آزادسازی خودکار حافظه ایجاد شده است
  - به نظر شما چطور این کار ممکن است؟

نکته

```
int number;
cin>>number;
int a[number];
```

- این قطعه‌کد را در نظر بگیرید:
- بعضی از کامپایلرها، این کد را مجاز می‌دانند و از آن خطا نمی‌گیرند
  - مثلاً gcc و g++ خطا نمی‌گیرند ولی Visual C++ خطا می‌گیرد
  - با این که number یک متغیر معمولی است و مقدار آن در زمان اجرا مشخص می‌شود
- این امکان، در نسخه‌های جدیدتر C (از C99) مجاز شده است
- زبان استاندارد C++ (ISO C++) اجازه تعریف آرایه به این شکل را نمی‌دهد
- روش مناسب، معمول و طبیعی برای ایجاد آرایه‌ای به طول متغیر، تخصیص حافظه پویا است
  - یعنی به جای `int a[number];`، بنویسید: `int*a = new int[number];`
  - و همچنین در زمان مقتضی آرایه a را delete کنید

مدیریت حافظه پویا در زبان C

- در زبان C ، عملگرهای new و delete وجود ندارد
  - این دو عملگر مخصوص C++ هستند
- در عوض، در زبان C توابع malloc و free داریم
- با کمک تابع malloc حافظه تخصیص می‌یابد و با کمک تابع free آزاد می‌شود
  - (malloc به جای new و free به جای delete)
- تابع malloc ، یک void* برمی‌گرداند: void* malloc (size_in_bytes);
  - پارامتر این تابع هم، برخلاف new، تعداد خانه‌های لازم نیست، بلکه تعداد بایت‌های لازم است

```
int * array = (int*) malloc(10 * sizeof(int));
...
free(array);
```

new یا malloc؟ کدام یک بهتر است؟

- ترجیحاً از روش C++ استفاده کنید
- در مجموع، روش C++ بهتر، امن‌تر و ساده‌تر از روش C است
  - new/delete بهتر از malloc/free
  - البته malloc هم مزایایی دارد، مثل امکان realloc برای افزایش طول حافظه تخصیص‌یافته
  - عملگر new اصطلاحاً type-safe است
    - به جای void*، اشاره‌گری به نوع موردنظر را برمی‌گرداند
  - new در صورت ناتوانی در اختصاص حافظه، خطا (exception) می‌دهد
  - malloc در صورت ناتوانی در اختصاص حافظه، null برمی‌گرداند

مثال: آرایه‌ای از اشاره‌گرها

```
int a=1,b=2,c=3,d=4;
int* ptrs[4] = {&a, &b, &c, &d};
for (int i = 0; i < 4; ++i)
      *(ptrs[i]) = i*i;

cout<<a<<endl;
cout<<b<<endl;
cout<<c<<endl;
cout<<d<<endl;
```

```
0
1
4
9
```

آرایه دوبعدی پویا

- آرایه یک‌بعدی پویا: با کمک اشاره‌گر و new (مثلاً `int*a = new int[n];`)
- ایجاد یک آرایه دوبعدی پویا، با کمک اشاره‌گر به اشاره‌گر (مثلاً int**a)
- تخصیص حافظه برای آرایه دوبعدی پویا:
  - ابتدا باید یک آرایه یک‌بعدی (به عنوان سطرها) تخصیص یابد
  - سپس باید برای هر یک از سطرها، آرایه‌ای پویا (ستون‌ها) تخصیص یابد
- آزادسازی حافظه برای آرایه دوبعدی پویا:
  - ابتدا باید حافظه هر یک از سطرها، آزاد شود
  - سپس باید حافظه کل سطرها آزاد شود

مثال

آرایه دوبعدی متوازن (ماتریس)

```
int**matrix ;
int ROWS =3, COLUMNS=4;
matrix = new int*[ROWS];
for (int i = 0; i < ROWS; i++)
   matrix[i] = new int[COLUMNS];
...
for (int i = 0; i < ROWS; i++)
   delete[] matrix[i];
delete[] matrix;
```

آرایه دوبعدی نامتوازن

```
int**td ;
int ROWS =3;
td = new int*[ROWS];
td[0] = new int[5];
td[1] = new int[3];
td[2] = new int[7];
for (int i = 0; i < ROWS; i++)
   delete[] td[i];
delete[] td;
```

اشاره‌گر به تابع

- در زمان اجرا، تعریف هر تابع در حافظه جای می‌گیرد
- نام هر تابع، همانند آدرسی ثابت به محل این تابع در حافظه است
  - همان‌طور که نام یک آرایه، همانند آدرس شروع آرایه در حافظه است
- می‌توانیم اشاره‌گری تعریف کنیم که به یک تابع اشاره می‌کند
  - اشاره‌گر به تابع، به آدرس شروع تعریف تابع اشاره می‌کند
- برخی کاربردهای اشاره‌گر به تابع:
  - به عنوان پارامتر به یک تابع پاس شود
  - به عنوان خروجی تابع (مقدار برگشتی) برگردانده شود
  - و از اشاره‌گر به تابع، برای فراخوانی آن تابع می‌توان استفاده کرد

Function Pointer
Pointer to Function
(~Delegate)

تعریف اشاره‌گر به تابع

- نحوه تعریف: `Return_Type (*pointer_name)(param_types);`
- مثال: `int (*ptr)(double);`
  - یعنی ptr اشاره‌گری به یک تابع است که این تابع یک پارامتر double دارد و int برمی‌گرداند
- با کمک انتساب، می‌توانیم اشاره‌گر به تابع را به آدرس تابع موردنظر اشاره دهیم
- مثال: `ptr = fun;` یعنی اشاره‌گر `ptr` به تعریف تابع `fun` اشاره کند
- برای فراخوانی یک تابع، با کمک اشاره‌گری به آن تابع:

  از نام اشاره‌گر مثل نام تابع استفاده می‌کنیم
  - مثال: ptr(5.5) یعنی تابعی که ptr به آن اشاره می‌کند، با آرگومان 5.5 فراخوانی شود

مثال

```
#include <iostream>
#include <cmath>
using namespace std;

int (*ptr)(double);

int up(double d){
      return ceil(d);
}
int down(double d){
      return floor(d);
}
int main() {
   ptr = up;
   cout<<ptr(3.5)<<endl;
   ptr = down;
   cout<<ptr(3.5)<<endl;
}
```

4
3

- نام اشاره‌گر به تابع، هنگام تعریف، در پرانتز ذکر می‌شود

- در این کد،
  - `ptr=up` همانند `ptr=&up` است هر دو صحیح هستند و فرقی ندارند
  - پس `ptr=&down` هم صحیح است

استفاده از اشاره‌گر به تابع به عنوان پارامتر

- اشاره‌گر به تابع را به عنوان پارامتر می‌توانیم استفاده کنیم و آن را به یک تابع پاس کنیم
- با ارسال یک تابع به عنوان پارامتر تابع f ، رفتار f وابسته به تابع پارامترش خواهد بود
- در واقع یک تابع، یک «رفتار» را مشخص می‌کند
  - با کمک پارامتری از نوع اشاره‌گر به تابع، رفتار تابع اصلی وابسته به رفتار پارامترش می‌شود
  - یعنی بخشی از رفتار تابع اصلی به عنوان پارامترش مشخص می‌شود
- مثال:
  - تابع مرتب‌سازی، که نحوه مقایسه (صعودی یا نزولی بودن) را به عنوان پارامتر می‌گیرد
  - تابعی بنویسید که روی اعضای یک آرایه یک عملیات انجام و سپس نتیجه‌ها را با هم جمع می‌کند
    - این تابع باید عملیات موردنظر (مثلاً رساندن به توان ۲) را به عنوان یک پارامتر بگیرد

```
void bubble_sort(int a[], int size, bool (*compare)(int,int) ) {
   for (int i = 0; i < size; i++)
      for (int j = 0; j < size - i - 1; j++)
         if (compare(a[j + 1] , a[j]))
            swap(a[j], a[j + 1]);
}
bool ascending(int x, int y){return x<y;}
bool descending(int x, int y){return x>y;}
int main() {
   int array[]={1,5,4,2,3};
   bubble_sort(array, 5, ascending);
   for (int i = 0; i < 5; ++i)
      cout<<array[i]<<endl;
   bool (*desc)(int, int) = descending;
   bubble_sort(array, 5, desc);
   for (int i = 0; i < 5; ++i)
      cout<<array[i]<<endl;
}
```

```
1
2
3
4
5

5
4
3
2
1
```

مثال

تمرین

- تابعی بنویسید که این پارامترها را بگیرد:
  - یک آرایه از اعداد
  - آرایه‌ای از اشاره‌گر به تابع
- و همه توابع مشخص شده را روی تک‌تک اعضای آرایه اجرا کند

```
void apply(int a[], int a_size, void (*ptr[])(int&), int ptr_size)
{
   for (int i = 0; i < ptr_size; ++i)
      for (int j = 0; j < a_size; ++j)
            ptr[i](a[j]);
}
void increment(int& a) {a++;}
void twice(int& a) {a *= 2;}
void sqr(int& a) {a *= a;}
int main() {
   int array[] = { 1, 2, 3, 4 };
   void (*operations[])(int&)= {increment, twice, sqr};
   apply(array, 4, operations, 3);
   for (int i = 0; i < 4; ++i)
      cout << array[i] << endl;
}
```

پاسخ

```
16
36
64
100
```

اشاره‌گر ثابت و اشاره‌گر به ثابت

- اشاره‌گر غیرثابت به داده غیر ثابت : هم اشاره‌گر و هم محتوای محل اشاره قابل تغییرند
- Nonconstant Pointer to Nonconstant Data
- int *countPtr

- اشاره‌گر غیرثابت به داده ثابت : اشاره‌گر قابل تغییر است (می‌تواندبه جای دیگری اشاره کند) ولی محتوای محل اشاره قابل تغییر نیست
- Nonconstant Pointer to Constant Data
- const int *countPtr

- اشاره‌گر ثابت به داده غیرثابت : اشاره‌گر قابل تغییر نیست ولی محتوای محل اشاره قابل تغییر است
- Constant Pointer to Nonconstant Data
- int * const ptr

- اشاره‌گر ثابت به داده ثابت : اشاره‌گر و محتوای محل اشاره هیچ یک قابل تغییر نیست
- Constant Pointer to Constant Data
- const int * const ptr

جمع‌بندی

جمع‌بندی

- مفهوم اشاره‌گر
- آرایه و اشاره‌گر
- ارسال پارامتر با کمک اشاره‌گر
- اشاره‌گر به تابع
- مدیریت حافظه پویا
  - new و delete
  - malloc و free
  - مشکلات و پیچیدگی‌های آزادسازی حافظه پویا

مطالعه

- فصل ۷ از کتاب: C How to Program (Deitel&Deitel) 7th edition
- و یا فصل متناظر درباره اشاره‌گر از کتاب‌های مشابه

جستجوی بیشتر

- درباره مزایا و معایب استفاده از اشاره‌گر فکر و جستجو کنید

- آزادسازی خودکار حافظه پویا چگونه ممکن است؟
  - مفهوم زباله‌روب (garbage collection)
  - چه زبان‌هایی از این مفهوم پشتیبانی می‌کنند؟ (زبان C و C++ پشتیبانی نمی‌کند)
  - آزادسازی خودکار حافظه، چه مزایا و معایبی دارد؟

- اشاره‌گر به تابع چه کاربردهایی دارد؟

جستجوی بیشتر

چگونه با کمک typedef این کار را ساده‌تر کنیم؟

- یک تابع، چگونه می‌تواند اشاره‌گر به تابع برگرداند؟
- مثال:

```
int f1(char c, bool b) { return 1; }
int f2(char c, bool b) { return 2; }
int (*getFunc(double d, long l))(char, bool) {
    if (d + l < 10)
    return f1;
    return f2;
}
int main() {
    int (*ptr)(char, bool);
    ptr = getFunc(3.5, 2);
    cout<<ptr('a', true)<<endl;
    ptr = getFunc(7.5, 5);
    cout<<ptr('a', true)<<endl;
}
```

getFunc تابعی است
که یک double و یک long به عنوان پارامتر می‌گیرد
و اشاره‌گر به تابعی برمی‌گرداند که این تابع
یک char و یک bool به عنوان پارامتر می‌گیرد
و یک int برمی‌گرداند

پایان