درس «مبانی کامپیوتر و برنامه‌سازی»

تابع

صادق علی‌اکبری

سرفصل مطالب

- برنامه‌نویسی پیمانه‌ای
  - مفهوم تابع
  - تعریف تابع
  - فراخوانی تابع
- متغیرهای محلی و سراسری
- محدوده دسترسی به متغیرها
- ارسال با مقدار و ارسال با ارجاع

برنامه‌نویسی پیمانه‌ای (Modular Programming)

divide and conquer

- معمولاً برنامه را به قطعات مختلفی تقسیم می‌کنیم
- هر قطعه (module) را جداگانه پیاده می‌کنیم
- برنامه‌نویسی پیمانه‌ای: یک تکنیک طراحی نرم‌افزار
  - قابلیت‌های نرم‌افزار به بخش‌هایی **مستقل** و **قابل جایگزینی** تقسیم می‌شود،
  - به نحوی که هر بخش پیاده‌سازی یک قابلیت مشخص را انجام می‌دهد
- هر یک از این قطعات یک **زیربرنامه** است
- جنس زیربرنامه‌ها به نوع زبان برنامه‌نویسی بستگی دارد (کلاس، تابع، ...)
- در زبان‌های رویه‌ای (procedural) مثل C ، زیربرنامه‌ها از جنس توابع هستند
  - در زبان‌های شیءگرا مثل C++ : توابع و کلاس‌ها

زیربرنامه

- در زبان‌های مختلف این زیربرنامه‌ها به نام‌های مختلفی خوانده می‌شوند
  - رویه یا روال (Procedure)
  - Subroutine یا Routine
  - متد (Method)
  - **تابع (Function)**
- در زبان C و C++ از اصطلاح **تابع** (function) استفاده می‌شود
- ما هر بخش از برنامه را در یک **تابع** مستقل پیاده می‌کنیم

تابع

- دستوری که قابل فراخوانی است
- قبلاً از توابع مختلفی استفاده کرده‌ایم، مثال؟
- printf, scanf, rand, …
- این توابع قبلاً تعریف شده‌اند، و ما آن‌ها را فراخوانی می‌کنیم
- ما می‌توانیم توابع جدیدی نیز تعریف کنیم
  - مثل این است که دستورات جدیدی به زبان اضافه کنیم
  - مثلاً تابعی با عنوان readNonZeroInt که یک عدد صحیح مثبت از کاربر بگیرد
    - و اگر ورودی بزرگتر از صفر نبود، دوباره بخواند

کتابخانه (Library)

- مجموعه‌ای از توابع مرتبط به هم را یک کتابخانه (Library) می‌گویند
- مثال؟
  - کتابخانه stdio ، iostream ، stdlib و ...
- اگر بخواهیم از توابع یک کتابخانه استفاده کنیم، از دستور include استفاده می‌کنیم
- مثال: #include <stdio.h>
- کتابخانه استاندارد C
  - مجموعه‌ای توابع و امکانات پایه‌ای برای زبان C
  - شامل stdio.h ، stdlib.h ، time.h و ...
  - به همین ترتیب کتابخانه استاندارد C++ هم برای زبان C++ وجود دارد
    - شامل iostream ، ctime ، cmath و ...

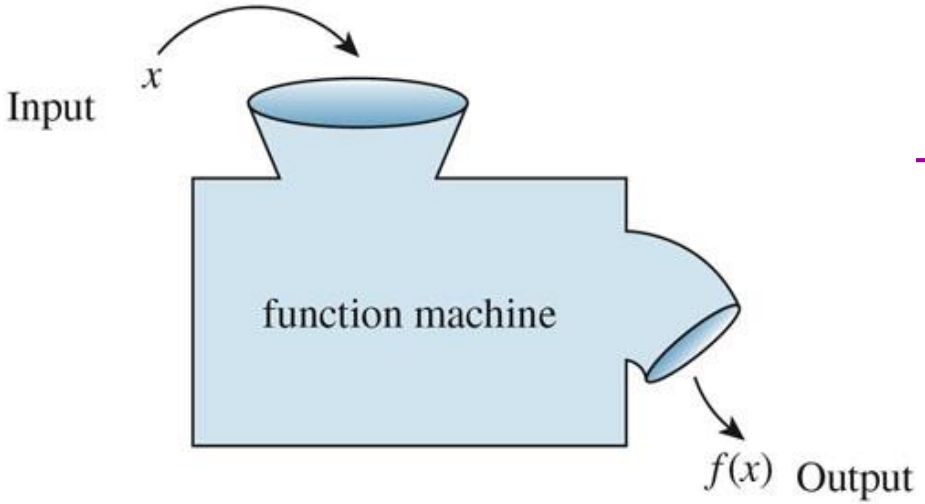
مفهوم تابع

- یک تابع، مجموعه‌ای از دستورات است
- هرگاه یک تابع فراخوانی شود، همه این دستورات اجرا می‌شوند
- یک تابع ممکن است بارها در یک برنامه اجرا شود
  - مثل تابع printf یا scanf که بارها در یک برنامه اجرا می‌شوند
- هر گاه مجموعه‌ای از دستورات، وظیفه منسجم و مستقلی انجام می‌دهند، خوب است آن‌ها را به یک تابع تبدیل کنیم
  و هر گاه به این وظیفه نیازمندیم، تابع موردنظر را فراخوانی کنیم
- توابع مورداستفاده در یک برنامه ممکن است:
  - توسط دیگران تولید شده باشد، مثل توابع استاندارد زبان یا کتابخانه‌های متن‌باز (open source)
  - یا توسط خودمان تعریف و تولید شوند

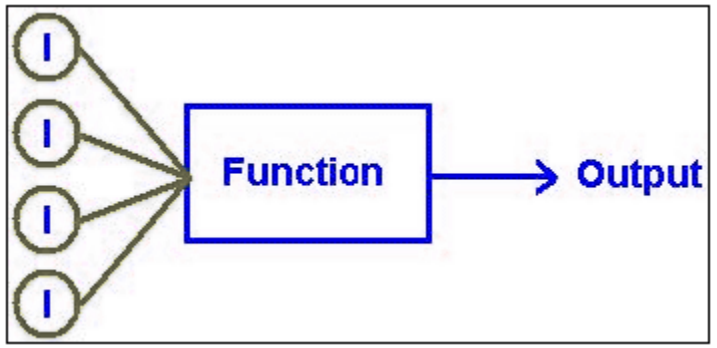
مزایای برنامه‌نویسی پیمانه‌ای

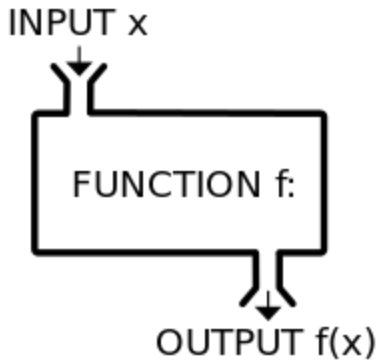
- تولید برنامه تسهیل و تسریع می‌شود
- فهمیدن برنامه ساده‌تر می‌شود
- تغییر برنامه آسان‌تر می‌شود
- تست و آزمایش برنامه راحت‌تر می‌شود
- رفع اشکال (debug) آسان‌تر می‌شود
- تولید برنامه در یک تیم (تولید گروهی) آسان‌تر می‌شود
- خلاصه: مدیریت تولید، آزمایش و نگهداری برنامه آسان‌تر می‌شود

ورودی و خروجی تابع



- یک تابع، مانند یک دستگاه است
  که صفر یا چند ورودی دارد
  و صفر یا یک خروجی دارد
- به ورودی‌های تابع، آرگومان (argument) یا پارامتر (parameter) می‌گوییم
- به خروجی تابع، مقدار برگشتی (return value) می‌گوییم





فراخوانی تابع

- **function call** or **function invocation**

- برای فراخوانی تابع:
  - عنوان تابع را ذکر می‌کنیم
  - پارامترها (آرگومان‌ها) را در پرانتز ذکر می‌کنیم
  - از مقدار برگشتی (خروجی) استفاده می‌کنیم (اگر مقدار برگشتی وجود داشته باشد)
- مثال:

```
int random = rand() ;
```

```
int random = rand() % 100 + 1;
```

```
printf("%d", rand() % 100 );
```

- ورودی‌ها و خروجی‌های rand ؟
- ورودی‌های printf ؟

مثال: کتابخانه cmath

Method	Description	Example
ceil(x)	rounds *x* to the smallest integer not less than *x*	ceil(9.2) is 10.0 ceil(-9.8) is -9.0
cos(x)	trigonometric cosine of *x* (*x* in radians)	cos(0.0) is 1.0
exp(x)	exponential function *ex*	exp(1.0) is 2.71828 exp(2.0) is 7.38906
fabs(x)	absolute value of *x*	fabs(5.1) is 5.1 fabs(0.0) is 0.0 fabs(-8.76) is 8.76
floor(x)	rounds *x* to the largest integer not greater than *x*	floor(9.2) is 9.0 floor(-9.8) is -10.0
fmod(x, y)	remainder of *x/y* as a floating-point number	fmod(13.657, 2.333) is 1.992
log(x)	natural logarithm of *x* (base *e*)	log(2.718282) is 1.0 log(7.389056) is 2.0
log10(x)	logarithm of *x* (base 10)	log10(10.0) is 1.0 log10(100.0) is 2.0
pow(x, y)	*x* raised to power *y* (*xy*)	pow(2, 7) is 128 pow(9, .5) is 3
sin(x)	trigonometric sine of *x* (*x* in radians)	sin(0.0) is 0
sqrt(x)	square root of *x*	sqrt(900.0) is 30.0 sqrt(9.0) is 3.0
tan(x)	trigonometric tangent of *x* (*x* in radians)	tan(0.0) is 0

مثال

```
#include <cmath>
```

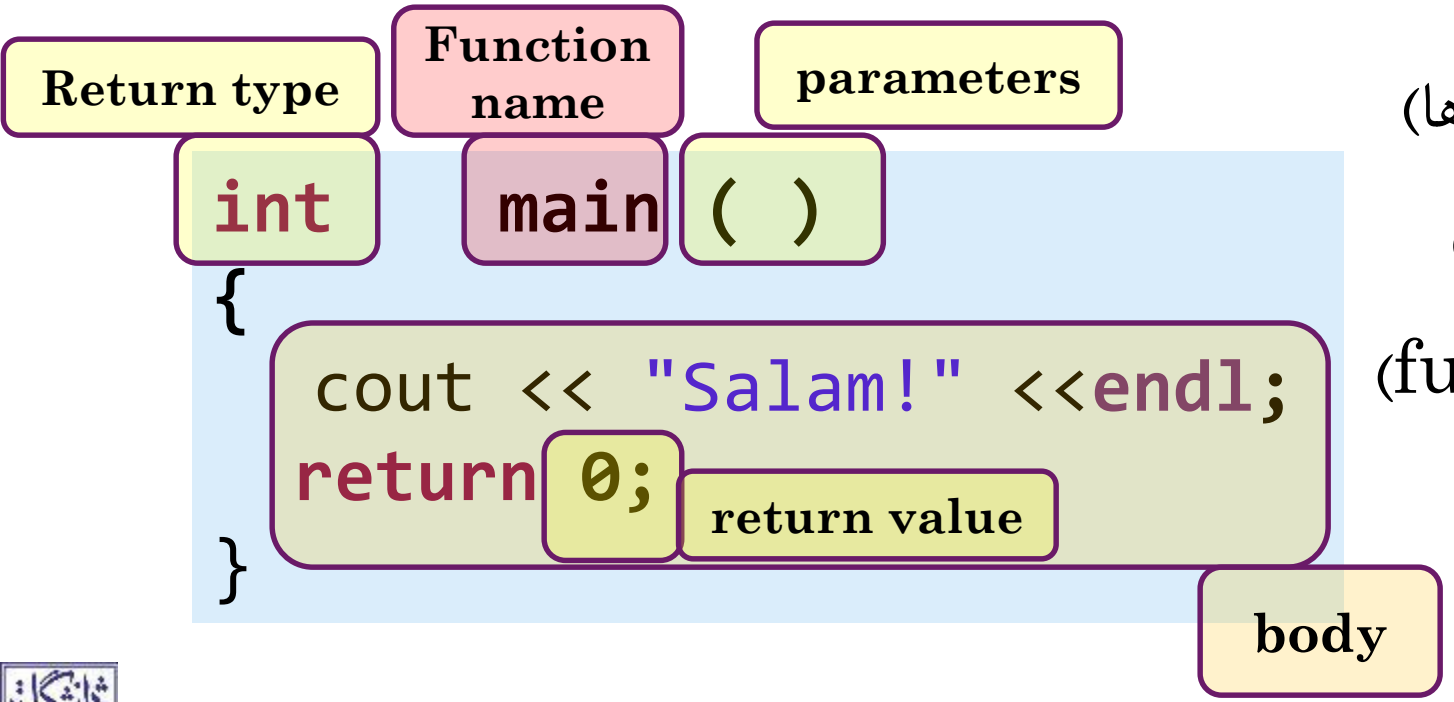
```
cout << sqrt( 900.0 ) <<endl;
double d = pow(3, 2);
cout << d <<endl;
cout << sin(3.14/2) <<endl;
d = floor (3.14);
cout << d <<endl;
d = ceil (3.14);
cout << d <<endl;
```

```
30
9
1
3
4
```

تعریف تابع

- Function Definition (implementation)

- برای تعریف تابع باید موارد زیر را مشخص کنیم:
  - نام تابع
  - آرگومان‌های ورودی (پارامترها)
  - نوع مقدار برگشتی (خروجی)
  - بدنه تابع (function body)
- مثال:



تمرین

- تابعی که یک عدد صحیح n به عنوان ورودی (پارامتر) بگیرد
  و مجموع اعداد ۱ تا n را برگرداند
- توجه: این تابع قرار نیست از کاربر ورودی بگیرد
  و یا قرار نیست چیزی را برای کاربر نمایش دهد (چاپ کند)

پاسخ

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

int sum(int n) {
    int result = 0;
    for (int i = 1; i <= n; i++)
        result += i;
    return result;
}

int main() {
    cout << sum(4) << endl;
    int n ;
    cin >> n;
    int s = sum(n);
    cout << s << endl;
    return 0;
}
```

تمرین

```
8
5
1
81
-1
```

- تابعی که
  دو عدد صحیح مثبت
  به عنوان پارامتر بگیرد
  و اولی را به توان دومی برساند
  و نتیجه را برگرداند
- دقت کنید: تابع موردنظر
  تعاملی با کاربر ندارد
  (مثلاً cin و cout ندارد)

```
#include <iostream>
using namespace std;
int power(int base, int pow){
   int result = 1;
   for(int i=0;i<pow;i++)
      result*=base;
   return result;
}
int main() {
   cout << power(2,3) << endl;
   cout << power(5,1) << endl;
   cout << power(2,0) << endl;
   int temp = power(3,4);
   cout << temp << endl;
   temp = 2 * power(2,2) - power(3,2);
   cout << temp << endl;
   return 0;
}
```

تمرین

```
#include <iostream>
using namespace std;
int readPositiveInt(){
   int read;
   do{
      cout<< "Enter a positive integer: ";
      cin >> read;
   }while(read<=0);
   return read;
}
int main() {
   int positive = readPositiveInt();
   cout<<positive<<endl;
   cout<<readPositiveInt()<<endl;
   return 0;
}
```

```
Enter a positive integer:-2
Enter a positive integer:0
Enter a positive integer:1
1
Enter a positive integer:-3
Enter a positive integer:5
5
```

تابعی با عنوان readPositiveInt

که یک عدد صحیح مثبت از کاربر بگیرد

وهمان را برگرداند

و مادامی که ورودی بزرگتر از صفر نبود،

دوباره بخواند

دقت کنید: تابع موردنظر با کاربر تعامل دارد

ولی ورودی و خروجی کاربر با ورودی و خروجی تابع متفاوت است

نوع برگشتی void

- گاهی یک تابع خروجی خاصی ندارد (مقدار برگشتی ندارد)
- کارهایی انجام می‌دهد ولی چیزی برنمی‌گرداند
- مثال:

```
void printMenu(){
   cout<<"1) Mouse \n";
   cout<<"2) Laptop \n";
   cout<<"3) Mobile \n";
   cout<<"4) Exit \n";
   cout<<"Please Select: ";
}
int main() {
   printMenu();
   return 0;
}
```

```
1) Mouse
2) Laptop
3) Mobile
4) Exit
Please Select:
```

نکته

- می‌توانیم مقدار برگشتی تابع را نادیده بگیریم
- حتی توابعی که مقدار برگشتی دارند (void نیستند) را می‌توانیم بدون استفاده از مقدار برگشتیشان فراخوانی کنیم
- این کار اشکال نحوی (syntax error) ندارد
- البته ممکن است باعث خطای ناخواسته برنامه‌نویس شود
- راستش را بخواهید توابعی مثل printf و scanf هم در واقع void نیستند!

```
int f(){
    cout<<"Salam";
    return 2;
}
int main() {
    f();
    int x = f();
}
```

int printf (const char * format, ...)

تمرین

- تابعی با عنوان isUppercase بنویسید
  که یک کاراکتر به عنوان پارامتر بگیرد
  و مشخص کند که
  آیا یک حرف انگلیسی بزرگ است یا خیر؟
- از این تابع در یک برنامه استفاده کنید

```
bool isUppercase(char ch) {
  return ch >= 'A' && ch <= 'Z';
}
int main() {
  if (isUppercase('5'))
     cout << "5 is uppercase!";
  bool x = isUppercase('X');
  if (x)
    cout << "X is uppercase!";
}
```

تمرین

$$P(n,k) = \underbrace{n \cdot (n-1) \cdot (n-2) \cdots (n-k+1)}_{k \text{ factors}}$$

$$P(n,k) = \frac{n!}{(n-k)!}$$

$$P(5,2) = \frac{5!}{(5-2)!} = \frac{5!}{3!} = \frac{120}{6} = 20$$

$$C(n,k) = \frac{P(n,k)}{P(k,k)} = \frac{n!}{(n-k)!k!}.$$

- تابعی بنویسید که فرمول جایگشت را پیاده‌سازی کند
- تابع دیگری بنویسید که فرمول ترکیب را پیاده‌سازی کند
- روش مناسب: تابع فاکتوریل را تعریف کنیم و در هر دو فرمول **بازاستفاده** کنیم
  - فراخوانی یک تابع درون یک تابع دیگر
- مفهوم استفاده مجدد یا بازاستفاده یا **reuse**

راه حل

```
int factorial(int n) {
   if(n<0)
      return 0;
   int f = 1;
   while(n>1)
      f*=n--;
   return f;
}
```

```
int permutation(int n, int k){
   if(n<0 || k<0 || k>n)
      return 0;
   return factorial(n)/factorial(n-k);
}
```

```
int combination(int n, int k){
   if(n<0 || k<0 || k>n)
      return 0;
   return factorial(n)/(factorial(n-k)*factorial(k));
}
```

```
int main() {
   cout << factorial(3) <<endl;
   cout << permutation(5,2) <<endl;
   cout << combination(5,2) <<endl;
}
```

متغیرهای سراسری (global variables)

```
#include <iostream>
using namespace std;
int x;
int main() {
   int y;
   cout<<x<<endl;
   cout<<y<<endl;
}
```

```
0
4202078
```

- متغیرهایی که خارج از هر تابعی تعریف می‌شوند
- در هر تابعی قابل فراخوانی هستند
- با صفر مقداردهی اولیه می‌شوند

- بهتر است حتی‌الامکان از متغیرهای سراسری استفاده نکنیم
  - چرا؟
  - این متغیرها، موجوداتی بین زیربرنامه‌ها (modules) و در دسترس همه آن‌ها هستند
  - استقلال و انسجام زیربرنامه‌ها را تهدید می‌کنند، فهمیدن نحوه تغییر و استفاده آن‌ها سخت است و ...

متغیرهای محلی (local variables)

- متغیرهای محلی داخل یک بلوک (block) تعریف می‌شوند
  - مثلاً داخل یک تابع (بدنه تابع هم یک بلوک است)
- فقط داخل همان بلوک قابل استفاده هستند
- طول عمر متغیر محلی: از شروع اجرای بلوک تا انتهای بلوک
- طول عمر متغیر سراسری: از شروع اجرای برنامه تا انتهای کل برنامه (از اول تا همیشه)
- متغیرهای محلی مقداردهی اولیه نمی‌شوند
  - مقدار اولیه آن‌ها معلوم و قابل پیش‌بینی نیست
- پارامترها هم متغیرهای محلی هستند

مثال

- متغیرهای سراسری و محلی را مشخص کنید

```
#include <iostream>
using namespace std;
double PI = 3.14;
int multiply(int x, int y){
   int result = x*y;
   return result;
}
int main() {
   int a,b;
   cin>>a;
   cin>>b;
   int mult = multiply(a,b);
   cout<<mult<<endl;
   cout<<"PI="<<PI<<endl;
   return 0;
}
```

محدوده متغیرها (scope)

- محدوده اعتبار متغیر: هر متغیر، در یک محدوده‌ای از برنامه قابل استفاده است
- مثال
  - متغیرهای سراسری در کل برنامه
  - متغیرهای محلی فقط در بلوکی که در آن تعریف شده‌اند
- در یک محدوده امکان تعریف دو متغیر هم‌نام وجود ندارد
  - ولی در دو محدوده مختلف این کار ممکن است

مثال

```
#include <iostream>
using namespace std;
int x = 1;
void function(){
      cout << "4->" << x << endl;
}
int main() {
   int x = 2;
   cout << "1->" << x << endl;
   for (int i = 0; i < 5; i++) {
      int x = 3;
      cout << "2->" << x << endl;
   }
   cout << "3->" << x << endl;
   function();
   return 0;
}
```

```
1->2
2->3
2->3
2->3
2->3
2->3
3->2
4->1
```

- خروجی؟
- محدوده متغیر i ؟
  - بلوک حلقه

عملگر ::

- به این عملگر Scope resolution operator گفته می‌شود
- این عملگر، برای دسترسی به متغیر سراسری قابل استفاده است
- مثال:

```
#include <iostream>
using namespace std;
int x = 1;
int main() {
    int x = 2;
    cout << x <<endl;
    cout << ::x <<endl;
    return 0;
}
```

```
2
1
```

نکته

- هم‌نام بودن متغیرها
- دقت کنید:
  - متغیرهای x و y و mult در دو محدوده مختلف تعریف شده‌اند
  - تعریف و مقادیر یک محدوده ربطی به محدوده دیگر ندارد

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

int multiply(int x, int y) {
    int mult = x * y;
    return mult;
}
int main() {
    int x, y;
    cin >> x;
    cin >> y;
    int mult = multiply(x, y);
    cout << mult << endl;
    return 0;
}
```

امضا و پروتوتایپ تابع (Signature & Prototype)

- پروتوتایپ تابع شامل اطلاعات زیر است:
  - نام تابع
  - نوع داده برگشتی
  - ترتیب و نوع پارامترها
- اگر کسی بخواهد یک تابع را فراخوانی کند، کافیست موارد فوق را بداند
  - لازم نیست جزئیات پیاده‌سازی (function definition) را بداند
- کامپایلر هم برای بررسی نحوه صحیح فراخوانی یک تابع، فقط اطلاعات فوق را لازم دارد
- به اطلاعات فوق به جز «نوع داده برگشتی»، **امضای تابع** (signature) می‌گویند
  - امضا : نام تابع + ترتیب و نوع پارامترها

پروتوتایپ

- برای فراخوانی یک تابع، کافیست پروتوتایپ آن را اعلان کرده باشید
- می‌توانید بدنه تابع را بعداً تعریف کنید

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

int power(int, int) ;
int main() {
      cout << power(2, 3) << endl;
}
int power(int base, int pow) {
   int result = 1;
   for (int i = 0; i < pow; i++)
      result *= base;
   return result;
}
```

Function Signature

Function declaration (Prototype)

نکته: ذکر نام پارامترها در اعلان تابع لازم نیست

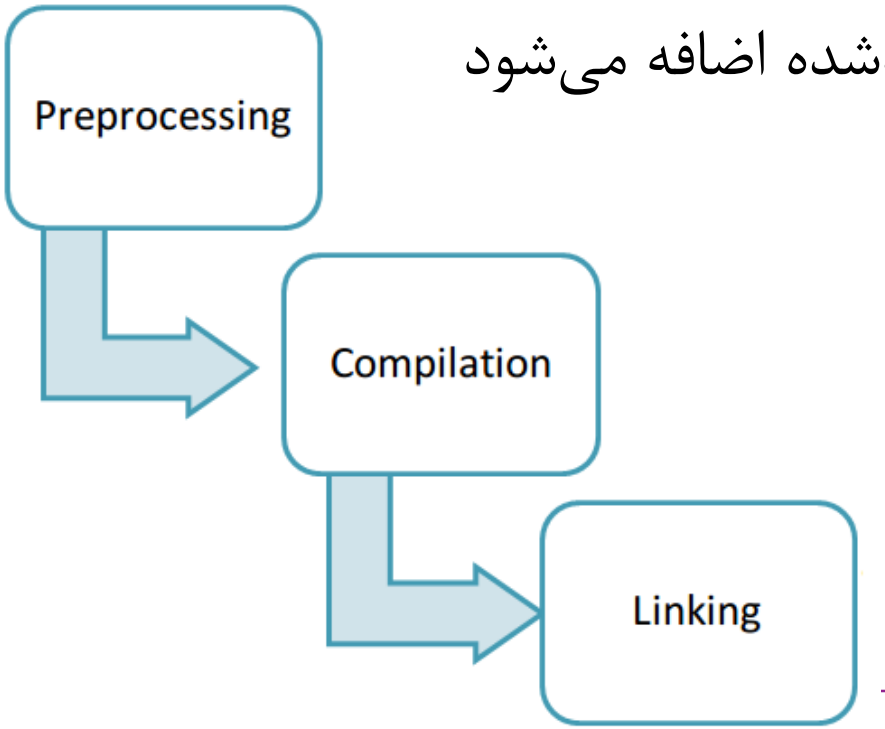
- اگر قبل از فراخوانی یک تابع، آن تابع **تعریف** و یا **اعلان** نشود: خطای کامپایل

Function definition (implementation)

کتابخانه‌ها و دستور include

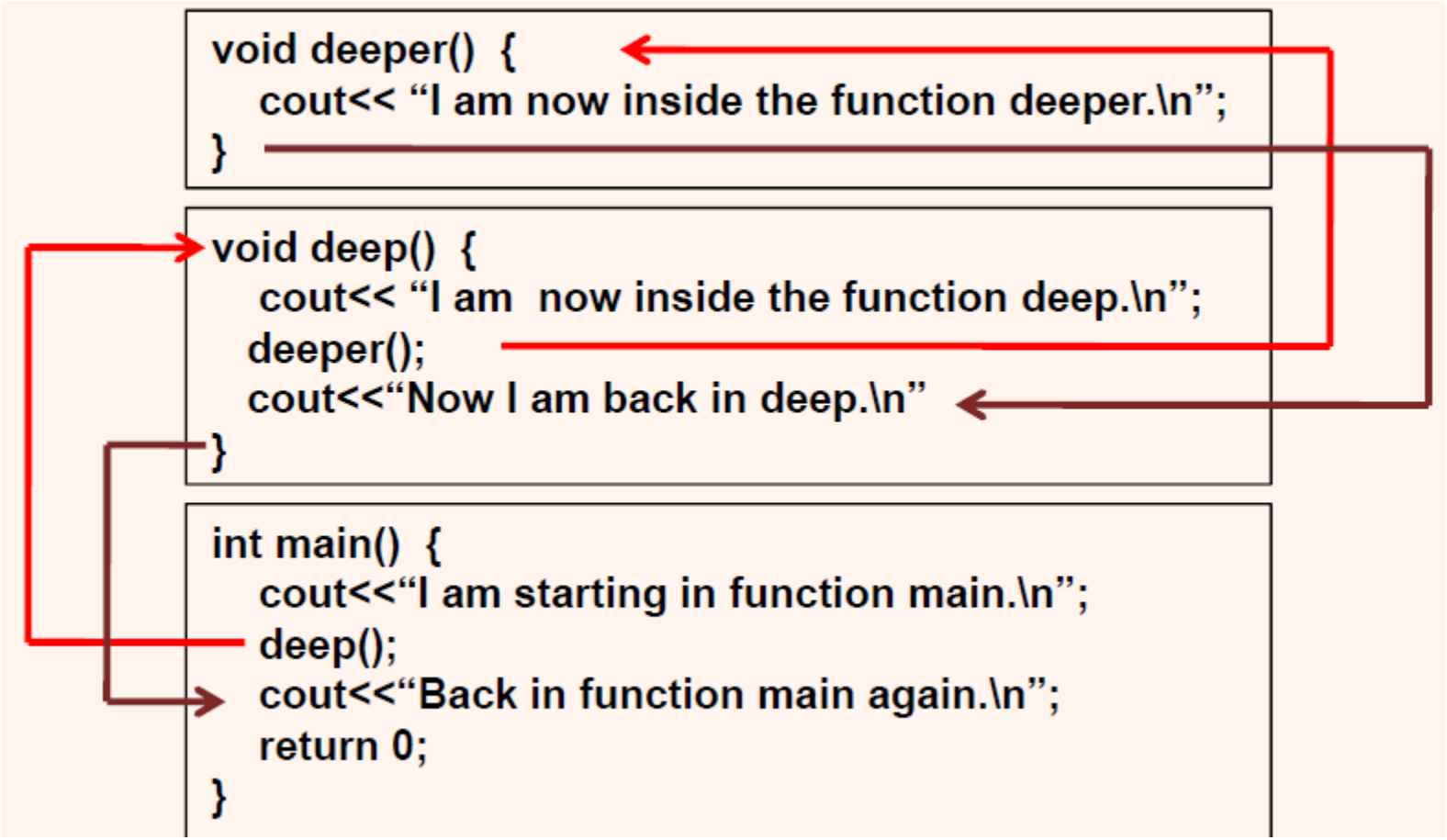
```
#include <stdio.h>
```

- فایل‌هایی با پسوند h. شامل مجموعه‌ای از اعلان‌ها برای توابع مختلف هستند
- با دستور include همه این اعلان‌ها به ابتدای برنامه اضافه می‌شود
  - این دستور، پیش از کامپایل، محتوای فایل h. را در ابتدای برنامه کپی می‌کند
- «تعریف» این توابع بعداً در مرحله link به برنامه ترجمه‌شده اضافه می‌شود
  - برای فراخوانی توابع، «اعلان» آن‌ها کافیست
  - برای اجرای برنامه، بدنه توابع هم لازم است



فراخوانی تودرتو

- از داخل یک تابع می‌توانیم تابع دیگری را فراخوانی کنیم



نکته: کارکرد دستور return

- دستور return مقدار برگشتی تابع را مشخص می‌کند
- دقت کنید:

  بلافاصله بعد از این دستور،

  از تابع خارج می‌شویم

- مثلاً بعد از return

  نیازی به else نیست

```
int fib(int n) {
   if(n<=0)
      return 0;
   if(n==1)
      return 1;
   int a = 1, b = 1;
   for (int i = 3; i <= n; i++) {
      int c = a + b;
      a = b;
      b = c;
   }
   return b;
}
```

اصول طراحی توابع

- انسجام بالا (High Coherence)
  - هر تابع، یک کار مستقل و واحد را انجام دهد
- وابستگی کم (Low Coupling)
  - هر تابع، حداقل وابستگی ممکن را به سایر توابع داشته باشد
- نام مناسب برای تابع
- مستندسازی مناسب
  - به خصوص: کامنت مناسب

متغیرهای محلی ایستا (static local variables)

- متغیرهای محلی، در هر بار فراخوانی تابع، ایجاد می‌شوند (حافظه می‌گیرند) و در خاتمه اجرای تابع، از بین می‌روند (حافظه آن‌ها آزاد می‌شود)
- متغیر محلی ایستا (static):

  در اجرای بعدی همان تابع، آخرین مقدار متغیر محلی حفظ می‌شود
- یعنی متغیر محلی ایستا در پایان اجرای تابع، آزاد نمی‌شود (در حافظه می‌ماند)
- طول عمر متغیر محلی ایستا (مثل متغیرهای سراسری) تا پایان برنامه است
- مقدار اولیه پیش‌فرض متغیر محلی ایستا (مثل متغیرهای سراسری) صفر است
- متغیر محلی ایستا مانند یک متغیر سراسری اما با محدوده دسترسی مشخص است

مثال

- خروجی؟

```
void f0(){
   int local;
   cout<<local<<endl;
}
```

```
void f1(){
   int local = 5;
   cout<<local<<endl;
   local++;
}
```

```
void f2(){
   static int local;
   cout<<local<<endl;
}
```

```
void f3(){
   static int local = 5;
   cout<<local<<endl;
   local++;
}
```

```
int main(){
   f0();
   f1(); f1();f1();
   f2();
   f3(); f3(); f3();
}
```

```
4202238
5
5
5
0
5
6
7
```

آرگومان پیش‌فرض

- می‌توانیم برای برخی پارامترها، مقدار پیش‌فرض تعیین کنیم
- در این صورت، هنگام فراخوانی تابع می‌توانیم مقدار این پارامترها را مشخص نکنیم
- همان آرگومان پیش‌فرض در نظر گرفته می‌شود
- مثال:

```
#include <iostream>
#include <cmath>
using namespace std;
double logarithm(double x, double base=10){
      return log(x)/log(base);
}
int main(){
   cout<<logarithm(10)<<endl;
   cout<<logarithm(10, 10)<<endl;
   cout<<logarithm(8, 2)<<endl;
}
```

مثال دیگر

```
#include <iostream>
using namespace std;
double div(double x=10, double y=5){
     return x/y;
}
int main(){
   cout<<div()<<endl;          //x=10,y=5
   cout<<div(20)<<endl;        //x=20,y=5
   cout<<div(9, 3)<<endl;      //x=9,y=3
}
```

نکته

- بعد از تعریف اولین آرگومان پیش‌فرض برای یک تابع،
  آرگومان‌های بعدی آن تابع هم باید مقدار پیش‌فرض داشته باشند
- به عبارت دیگر، آرگومان‌های پیش‌فرض باید در انتهای فهرست پارامترها بیایند

```
double div(double x=10, double y){
      return x/y;
}
```

- مثلاً کد روبرو خطای کامپایل دارد
- چرا؟
- زیرا قرار است آرگومان‌ها از جایی به بعد تعیین نشوند
- در مثال فوق، نمی‌توانید تابع div را با مقدار پیش‌فرض x و مقدار y=5 فراخوانی کنید

توصیه

- آرگومان پیش‌فرض و متغیر محلی استاتیک، امکانات مجاز زبان C++ هستند
- ولی می‌توانند باعث کاهش خوانایی برنامه شوند
- بهتر است (حتی‌الامکان) کمتر از این امکانات استفاده کنیم

سربار کردن توابع (function overloading)

- می‌توانیم توابع مختلف هم‌نام تعریف کنیم،
  به شرطی که **امضای** این توابع متفاوت باشد
- به این کار، **سربار کردن تابع** (function overloading) می‌گویند
- یادآوری: امضای تابع = نام تابع + ترتیب و نوع پارامترهایش
- پس سربار کردن تابع یعنی:
  تعریف چندباره یک تابع با مجموعه متفاوت پارامترها
- هر یک از این تعریف‌ها به صورت مستقل قابل فراخوانی هستند

مثال

تابع print با سه امضای مختلف

```
void print(int a){
      cout<<"Print int: "<<a<<endl;
}
void print(double a){
      cout<<"Print double: "<<a<<endl;
}
void print(int a, int b){
      cout<<"Print two integers: "<<a<<","<<b<<endl;
}
int main(){
   print(5);
   print(5.0);
   print(5,6);
   print(5.0,6.0);
}
```

```
Print int: 5
Print double: 5
Print two integers: 5,6
Print two integers: 5,6
```

سربار: فقط براساس تفاوت در پارامترها ممکن است

- براساس مقدار برگشتی نمی‌توانیم توابع را سربار کنیم

```
int f(){return 0;}
char f(){ return 'A';}
```

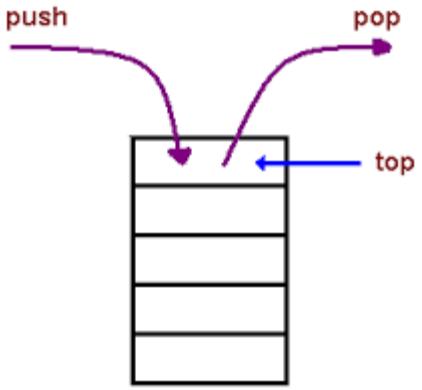
خطای کامپایل

- چرا؟
  - فراخوانی ;f() کدام یک را اجرا می‌کند؟

- اما این حالت اشکالی ندارد:

```
int f(){return 0;}
void f(int a){}
```

مفهوم پشته متغیرها (stack)



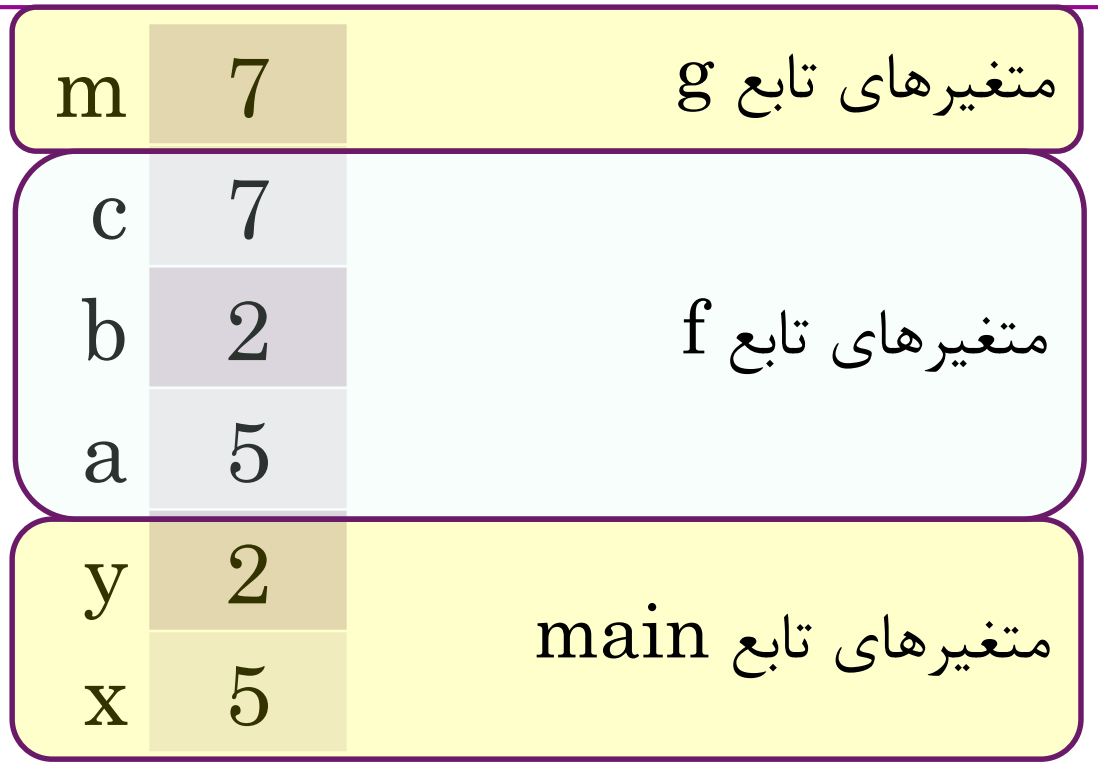
- بخشی از حافظه، شامل نگهداری مقدار متغیرهای محلی توابع مختلف است
- این بخش، به حافظه **پشته** (**stack**) موسوم است
- هر متغیر محلی، در حافظه پشته ذخیره می‌شود
- دقت کنید:

به ازای هر بار فراخوانی یک تابع، همه متغیرهای محلی آن در حافظه جای می‌گیرند

- هر تابع که فراخوانی شود، بخشی بر روی حافظه پشته اضافه می‌شود
  - برای نگهداری مقدار متغیرهای این تابع

مثال

```
void g(int m){
      cout<<m<<endl;
}
void f(int a, int b){
   int c = a + b;
   cout << c << endl;
   g(c);
}
int main(){
   int x =5 , y =2;
   f(x,y);
}
```



- بعد از اتمام اجرای تابع g (در آخرین خط تابع f) حافظه متغیرهای g از پشته آزاد می‌شود
- بعد از اتمام اجرای تابع f (در آخرین خط تابع main) حافظه متغیرهای f از پشته آزاد می‌شود
- در انتهای برنامه، حافظه متغیرهای main از پشته آزاد می‌شود

توابع درخط (inline)

- اگر یک تابع را به صورت inline تعریف کنیم،
  کامپایلر در هر جا که این تابع فراخوانی شود، محتوای تابع را جایگزین می‌کند
- یعنی به جای فراخوانی تابع،
  (که شامل ایجاد حافظه جدید روی پشته و ... است)
  کامپایلر دستور فراخوانی تابع را حذف، و بدنه تابع را در برنامه جایگزین می‌کند
- این کار، راهی برای افزایش کارایی برنامه است (صرفه‌جویی در زمان و حافظه)
  - برای توابع محاسباتی ساده که بسیار فراخوانی می‌شوند مناسب است
  - در صورت استفاده نابجا از این امکان، ممکن است کارایی برنامه کاهش یابد

مثال

```
inline int square(int x){
      return x*x;
}
int main(){
   cout<<square(5)<<endl;
   int x;
   cin>>x;
   cout<<square(5*x -1)<<endl;
}
```

نکته مهم:
کلیدواژه inline فقط یک **پیشنهاد** به کامپایلر است
لزوماً کامپایلر از پیشنهاد ما اطاعت نمی‌کند
(مانند کلیدواژه register که صرفاً یک پیشنهاد است)

- برنامه فوق (احتمالاً) مانند برنامه زیر ترجمه می‌شود:

```
int main(){
   cout<<(5)*(5)<<endl;
   int x;
   cin>>x;
   cout<<(5*x -1)*(5*x -1)<<endl;
}
```

نحوه ارسال پارمترها به توابع چگونه است؟

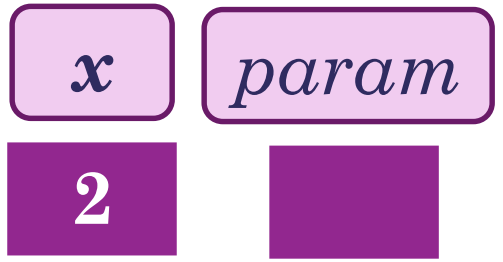
- زبان C++ از سه روش ارسال پارامتر پشتیبانی می‌کند
  - **Call by value**
  - Call by reference
  - Call by pointer
- فعلاً با روش Call by value آشنا می‌شویم:
  - یعنی: متغیر پارامتر در واقع یک کپی از مقداری است که به تابع پاس شده است
  - تغییرات به روی این کپی‌ها، تأثیری بر مقدار متغیر پاس شده اصلی نخواهد داشت

فراخوانی با مقدار (call by value)

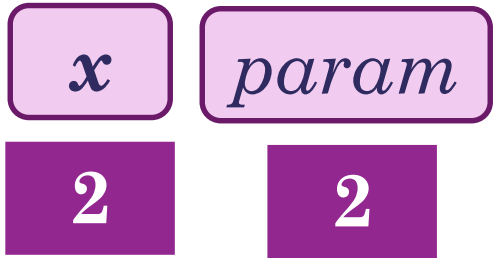
```
int x = ...
f(x);
```

```
void f(int param){
   ...
}
```

- قبل از فراخوانی تابع:



- بعد از فراخوانی تابع:



- انگار که چنین اتفاقی می‌افتد:

```
void f(){
   int param = x;
...
}
```

  - البته کد روبرو صحیح نیست
  - چون در f ممکن است x در دسترس نباشد

سؤال

```
void method(int number){
      number = 5;
}
int main() {
   int x = 2;
   method(x);
   cout<<x;
   return 0;
}
```

آیا بعد از فراخوانی **method**، مقدار x تغییر می‌کند؟

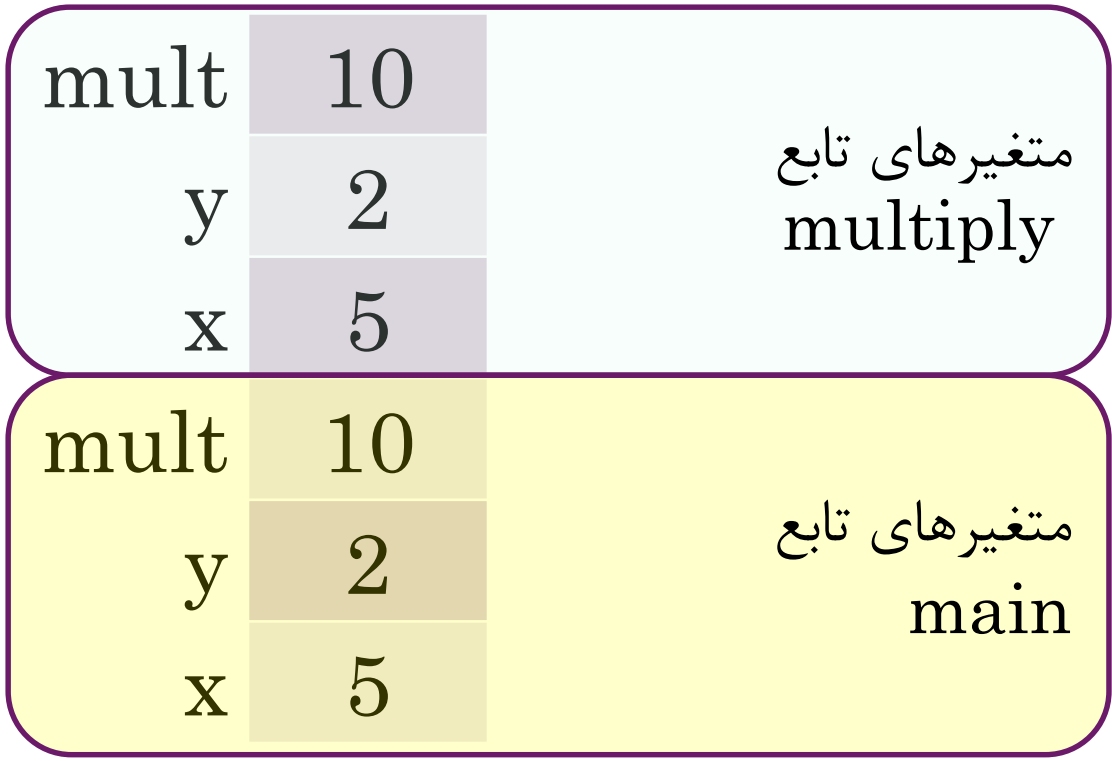
- خیر
- چرا؟
  - number یک کپی از مقدار x را دارد
  - تغییر number باعث تغییر x نمی‌شود

نحوه فراخوانی تابع

- مرور مفهوم ارسال با مقدار (call by value)

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

int multiply(int x, int y) {
    int mult = x * y;
    return mult;
}
int main() {
    int x, y;
    cin >> x;
    cin >> y;
    int mult = multiply(x, y);
    cout << mult << endl;
    return 0;
}
```



ارسال با ارجاع (call by reference)

- در روش «ارسال با مقدار»،
  یک کپی از آرگومان ارسالی در یک متغیر محلی (پارامتر) نگهداری می‌شود
- برخلاف این روش، در روش «ارسال با ارجاع»، پارامترها متغیرهای مستقلی نیستند
- یعنی هر پارامتر یک کپی از آرگومان ارسالی نیست
- بلکه نام دیگری برای همان متغیر اصلی است که به این تابع پاس شده است
- بنابراین اگر مقدار این پارامترها عوض شود، مقدار متغیر اصلی تغییر خواهد کرد
- در این روش، پارامترها با یک & مشخص می‌شوند
- مثال:

```
void f(int & x){...}
```

مثال

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

```
1,2
2,1
```

ارسال با ارجاع

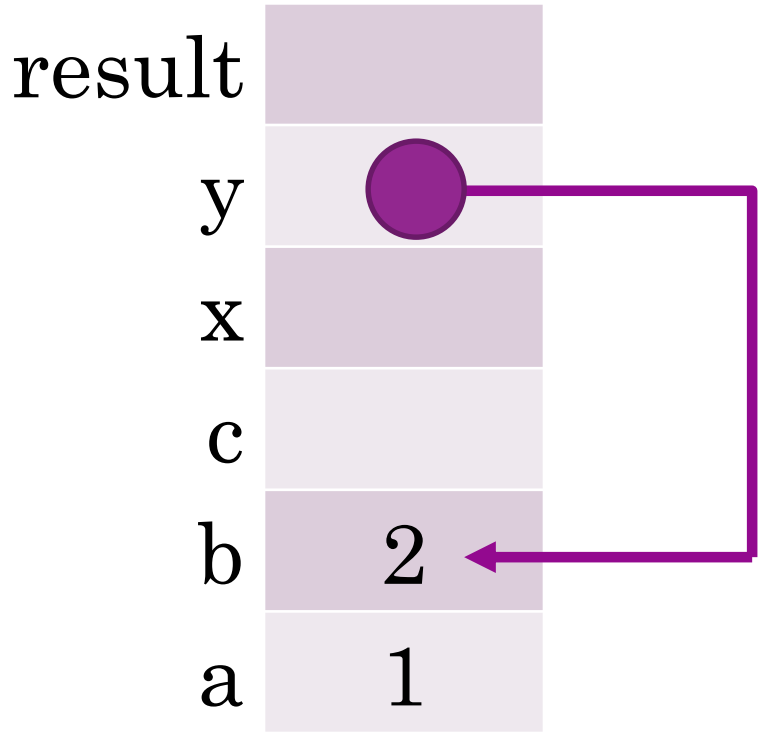
- متغیر ارجاعی نام دیگری برای متغیر اصلی است
  - مانند یک نام مستعار برای متغیر اصلی عمل می‌کند
  - متغیر و حافظه جداگانه‌ای روی پشته نخواهد بود
- اگر مقدار متغیر ارجاعی را تغییر دهید، متغیر اصلی نیز تغییر خواهد کرد
- در مواردی که لازم است تغییرات به روی متغیر اصلی نیز صورت پذیرد:
  آرگومان های تابع را به صورت ارجاعی تعریف کنید
- این رویکرد تولید چند خروجی در یک تابع را نیز ممکن می‌سازد
  - گفتیم هر تابع صفر یا یک خروجی دارد
  - اما در یک تابع می‌توانیم چند خروجی تولید کنیم و در متغیرهای ارجاعی ذخیره کنیم

تمرین

- خروجی این برنامه چیست؟

```
int f(int x, int &y){
    int result = x+y;
    x=5;
    y=5;
    return result;
}
int main(){
    int a=1, b=2;
    int c = f(a,b);
    cout<<a<<endl;
    cout<<b<<endl;
    cout<<c<<endl;
}
```

```
1
5
3
```



نکته

```
void f(int x) {}
void g(int& x) {}
int main() {
int a = 4;
f(a);
f(5);
f(4*a + 2);
g(a);
g(5);
g(4*a + 2);
}
```

خطای کامپایل

- هرگونه مقداری را از روش «ارسال با مقدار» می‌توانیم به توابع پاس کنیم
  - مثلاً یک مقدار، متغیر، یا عبارت
- اما فقط یک «متغیر» را می‌توانیم از روش «ارسال با ارجاع» به توابع پاس کنیم
  - پاس کردن یک مقدار یا عبارت با خطای کامپایل مواجه می‌شود

تولید چند خروجی با کمک call-by-ref

```
void circle(double , double& , double& );

int main()
{
      double radius = 2, ar, per;
      circle(radius, ar, per);
}
void circle(double r, double& area, double& perimiter)
{
   const double PI = 3.141592;
   area = PI * r * r;
   perimiter = 2*PI*r;
}
```

چه زمانی ارسال با ارجاع بهتر است؟

- اگر مقادیر ارسالی دارای حجم بالا باشند، call by value مقرون‌به‌صرفه نیست
  - مثلاً در مورد ساختارها و اشیاء بزرگ، نه درباره انواع داده کوچکی که تا این‌جا دیده‌ایم
  - کپی متغیرها حافظه زیادی هدر می‌دهد
- اگر بخواهیم تغییرات، روی متغیر اصلی نیز صورت پذیرد : call by ref
  - مثل تابع swap
- اگر بخواهیم چند خروجی از یک تابع داشته باشیم
  - مکانیزم return ، فقط یک خروجی را ممکن می‌سازد، ولی call-by-ref چند خروجی

چه زمانی ارسال با مقدار (call by value) بهتر است؟

- حالت پیش‌فرض متغیرها: call by value
  - ساده‌تر است، فهم برنامه را راحت می‌کند، استفاده از آن برای برنامه‌نویس ساده است
  - وابستگی کمتری بین توابع ایجاد می‌کند
- ارسال از طریق ارجاع، فقط برای متغیرها ممکن است
  - استفاده از ثابت‌ها و عبارت‌ها ممکن نیست، پس دست برنامه‌نویس را در استفاده می‌بندد (محدود می‌شود)
  - مثلاً در توابع math ، اگر پارامتر sin از نوع call-by-ref بود، نمی‌توانیم بگوییم:
    - double s = sin(5) یا double s = sin(5*x + 3);
- ارسال از طریق ارجاع، تغییر ناخواسته پارامترها را ممکن می‌سازد
  - ممکن است به اشتباه برنامه‌نویس منجر شود
  - راه حل: ثابت (const) کردن متغیر ارجاعی

ارجاع ثابت

- گاهی می‌خواهیم متغیر را به صورت ارجاعی پاس کنیم
  - چون حجم آن زیاد است و نمی‌خواهیم یک کپی از آن بسازیم و حافظه هدر برود
- همچنین نمی‌خواهیم اجازه دهیم این پارامتر، متغیر اصلی را تغییر دهد
- در این موارد می‌توانیم پارامتر را به صورت **متغیر ارجاعی ثابت** تعریف کنیم
- مثال: **const double& param**
- اگر کسی سعی کند این پارامتر را در تابع تغییر دهد: خطای کامپایل

مثال

```
void f(double a, double& b, const double& c)
{
   a = 1;
   b = 2;
   c = 3;                    خطای کامپایل
}
int main()
{
   double x=5,y=6,z=7;
   f(x,y,z);
}
```

جمع‌بندی

جمع‌بندی

- برنامه‌نویسی پیمانه‌ای
  - مفهوم تابع
  - تعریف تابع
  - فراخوانی تابع
- متغیرهای محلی و سراسری
- محدوده دسترسی به متغیرها
- ارسال با مقدار و ارسال با ارجاع

مطالعه

- Chapter 5 of C How to Program (Deitel & Deitel), 7th edition
  - و یا فصل‌های متناظر در کتاب C++

جستجوی بیشتر

- توابعی با تعداد آرگومان‌های متغیر (variable argument list)
  - معنای Variable Argument List یا vararg چیست؟
  - چطور می‌توانیم تابعی تعریف کنیم که تعداد آرگومان‌های آن نامشخص است؟
  - مثلاً printf (که ممکن است یک یا صد یا ... آرگومان داشته باشد) چگونه تعریف شده است؟
- اصطلاح Stack Overflow به چه معنایی است؟
  - چرا مهم است؟
  - سایت stackoverflow.com را هم دیده‌اید؟!
- مفهوم Function Template چیست؟
  - چگونه می‌توان با این امکان یک تابع تعریف کرد و با انواع مختلف داده فراخوانی نمود؟

جستجوی بیشتر (ادامه)

- زبان‌های برنامه‌نویسی دیگر (مثل جاوا و C#) چه روش‌های ارسال پارامتری را پشتیبانی می‌کنند؟
  - Call by value?
  - Call by reference?
  - Call by pointer?
- اصطلاح برنامه نویسی تابعی (functional programming) یعنی چه؟
  - چه شباهت‌ها و چه تفاوت‌هایی با روش رویه‌ای (procedural) دارد؟
  - چه شباهت‌ها و تفاوت‌هایی با رویکرد شیءگرا (Object Oriented) دارد؟
  - (در این درس با روش رویه‌ای پیش می‌رویم)

پایان