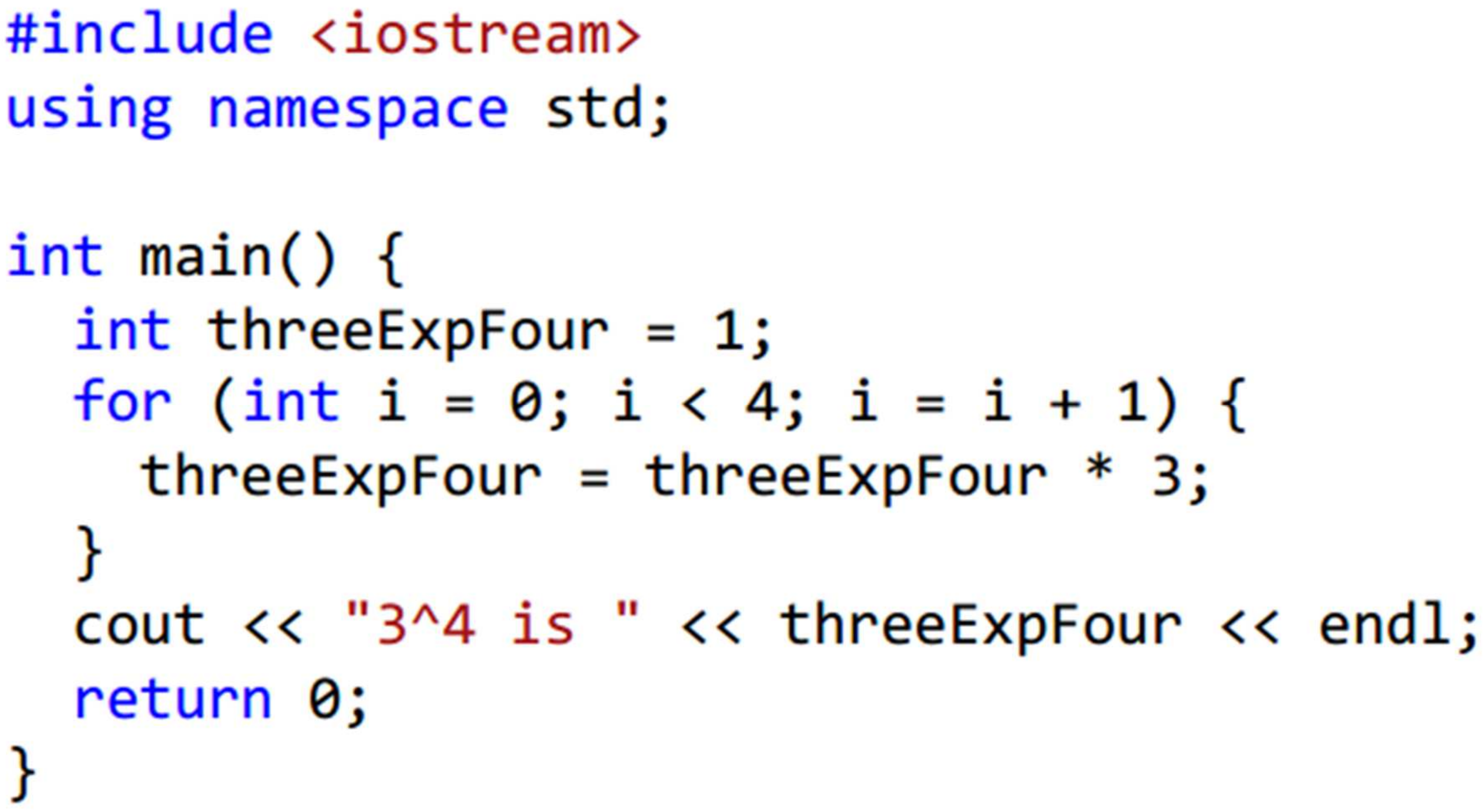
函数Function

如何重用代码

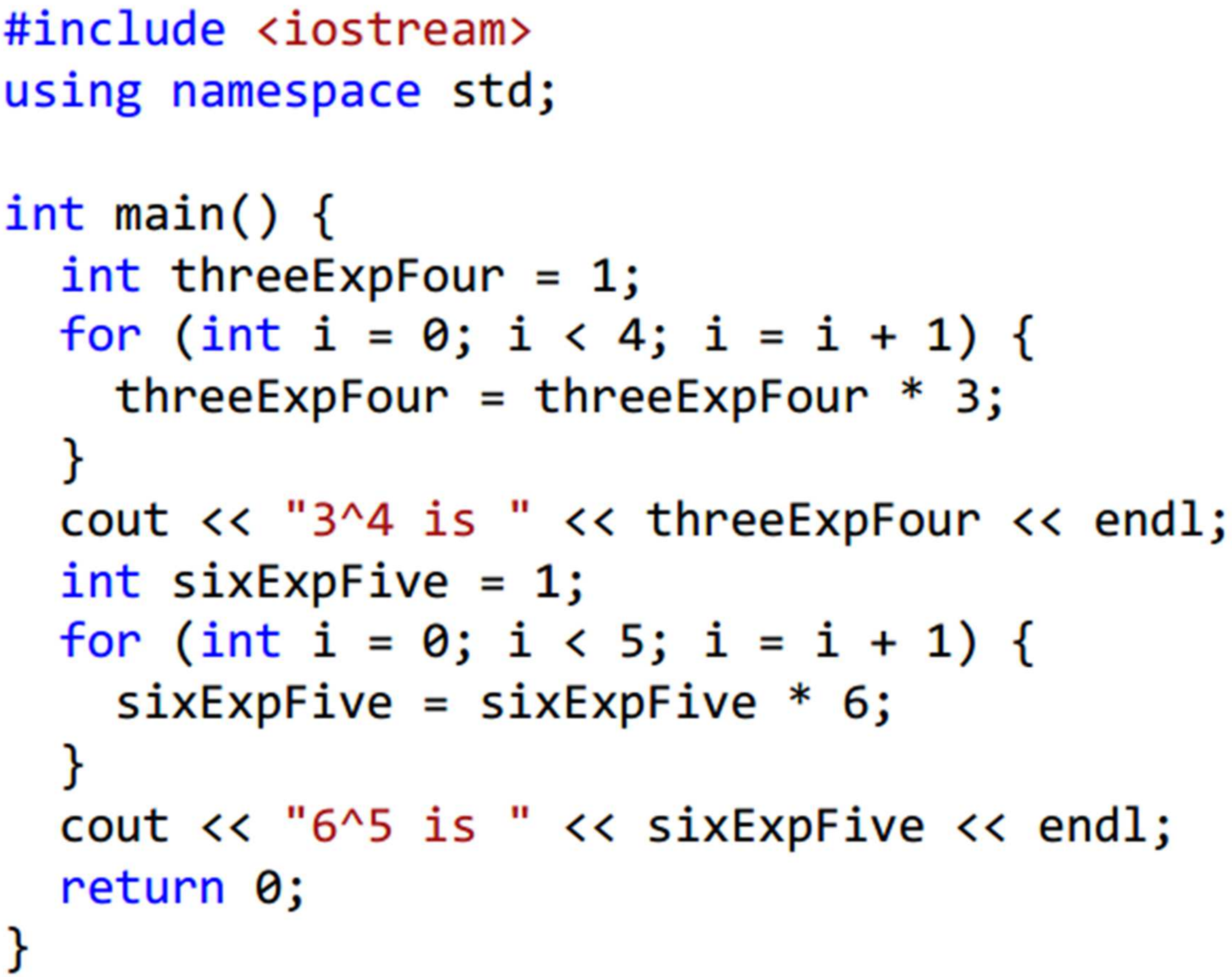
How to reuse code

$3^4 = 3*3*3*3$



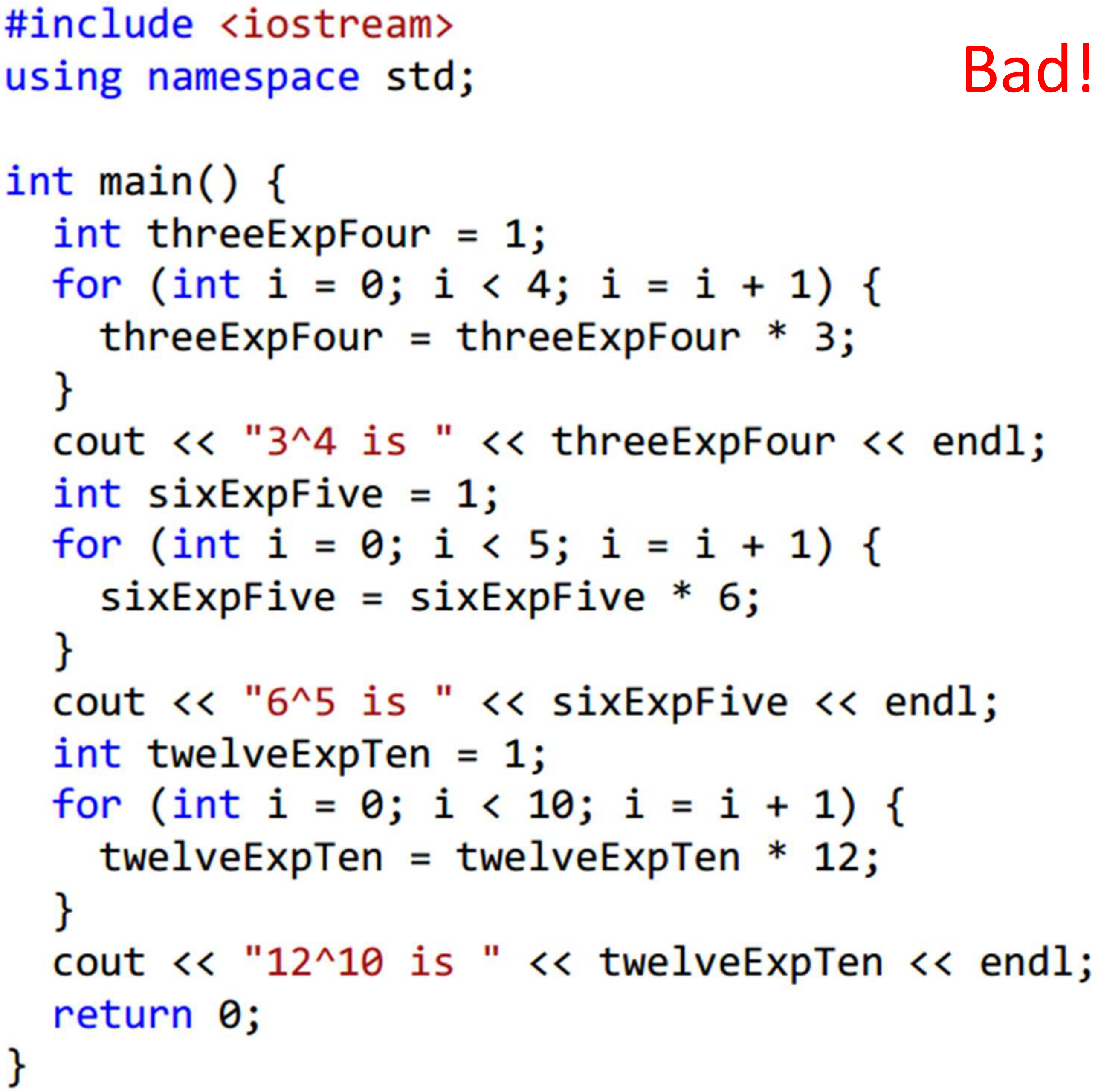
```
#include <iostream>
using namespace std;

int main() {
  int threeExpFour = 1;
  for (int i = 0; i < 4; i = i + 1) {
    threeExpFour = threeExpFour * 3;
  }
  cout << "3^4 is " << threeExpFour << endl;
  return 0;
}
```

3^4 ,6^5 :拷贝-粘帖代码(Copy-paste code)



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

int main() {
  int threeExpFour = 1;
  for (int i = 0; i < 4; i = i + 1) {
    threeExpFour = threeExpFour * 3;
  }
  cout << "3^4 is " << threeExpFour << endl;
  int sixExpFive = 1;
  for (int i = 0; i < 5; i = i + 1) {
    sixExpFive = sixExpFive * 6;
  }
  cout << "6^5 is " << sixExpFive << endl;
  return 0;
}
```

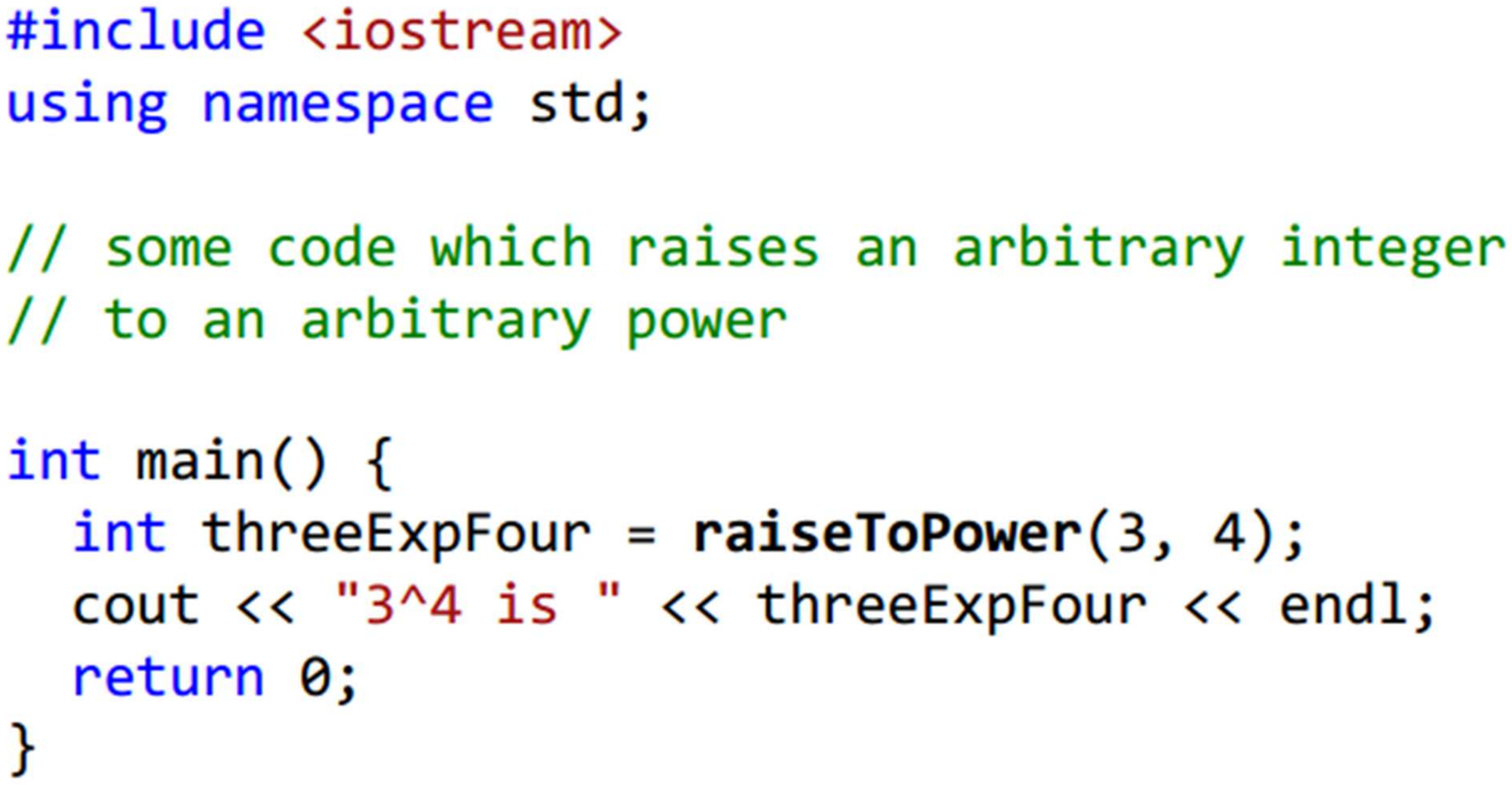
3^4 ,6^5 ,12^{10} :拷贝-粘帖代码(Copy-paste code)



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

int main() {
  int threeExpFour = 1;
  for (int i = 0; i < 4; i = i + 1) {
    threeExpFour = threeExpFour * 3;
  }
  cout << "3^4 is " << threeExpFour << endl;
  int sixExpFive = 1;
  for (int i = 0; i < 5; i = i + 1) {
    sixExpFive = sixExpFive * 6;
  }
  cout << "6^5 is " << sixExpFive << endl;
  int twelveExpTen = 1;
  for (int i = 0; i < 10; i = i + 1) {
    twelveExpTen = twelveExpTen * 12;
  }
  cout << "12^10 is " << twelveExpTen << endl;
  return 0;
}
```

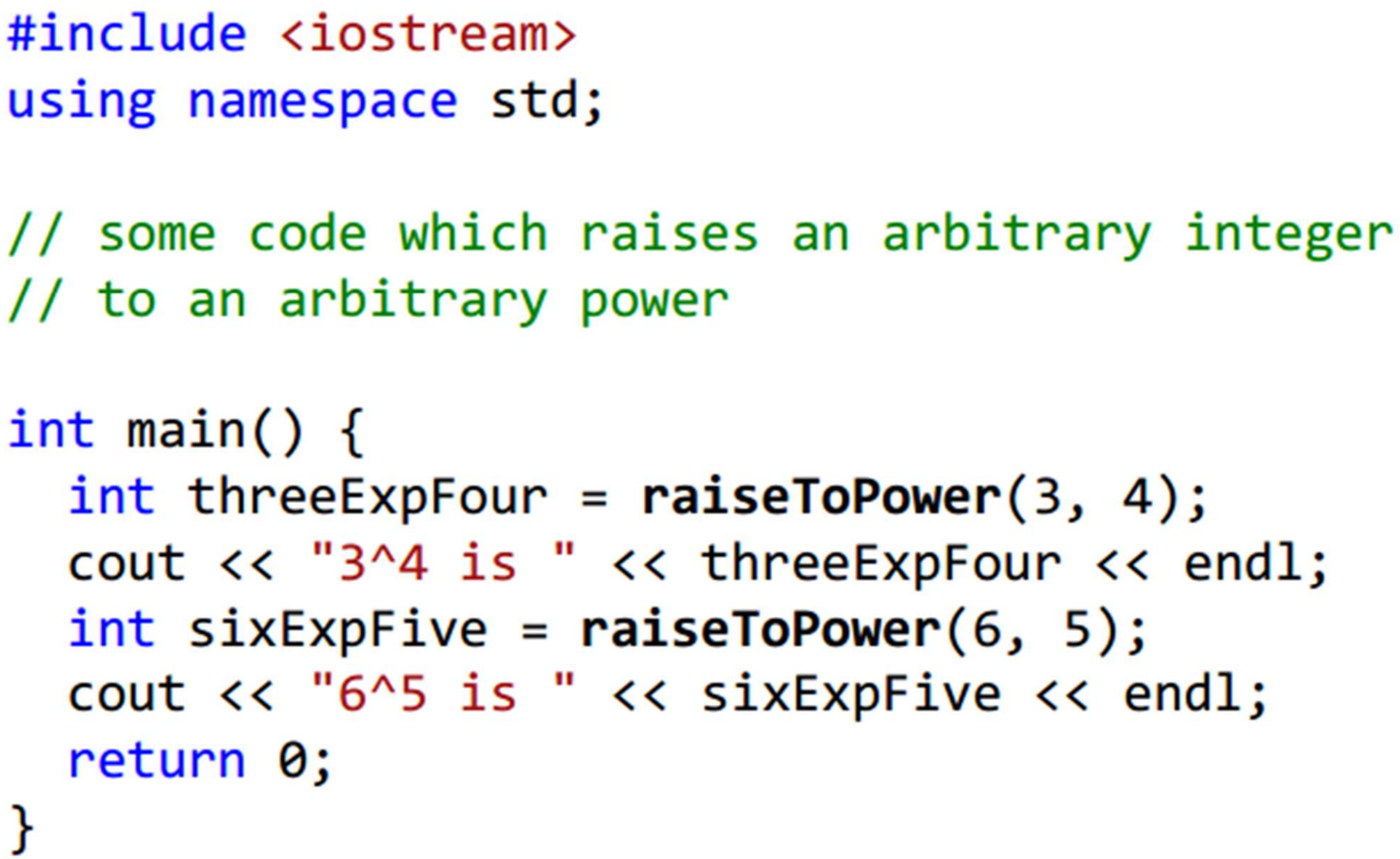
Bad!

使用函数(with a function)



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

// some code which raises an arbitrary integer
// to an arbitrary power

int main() {
  int threeExpFour = raiseToPower(3, 4);
  cout << "3^4 is " << threeExpFour << endl;
  return 0;
}
```

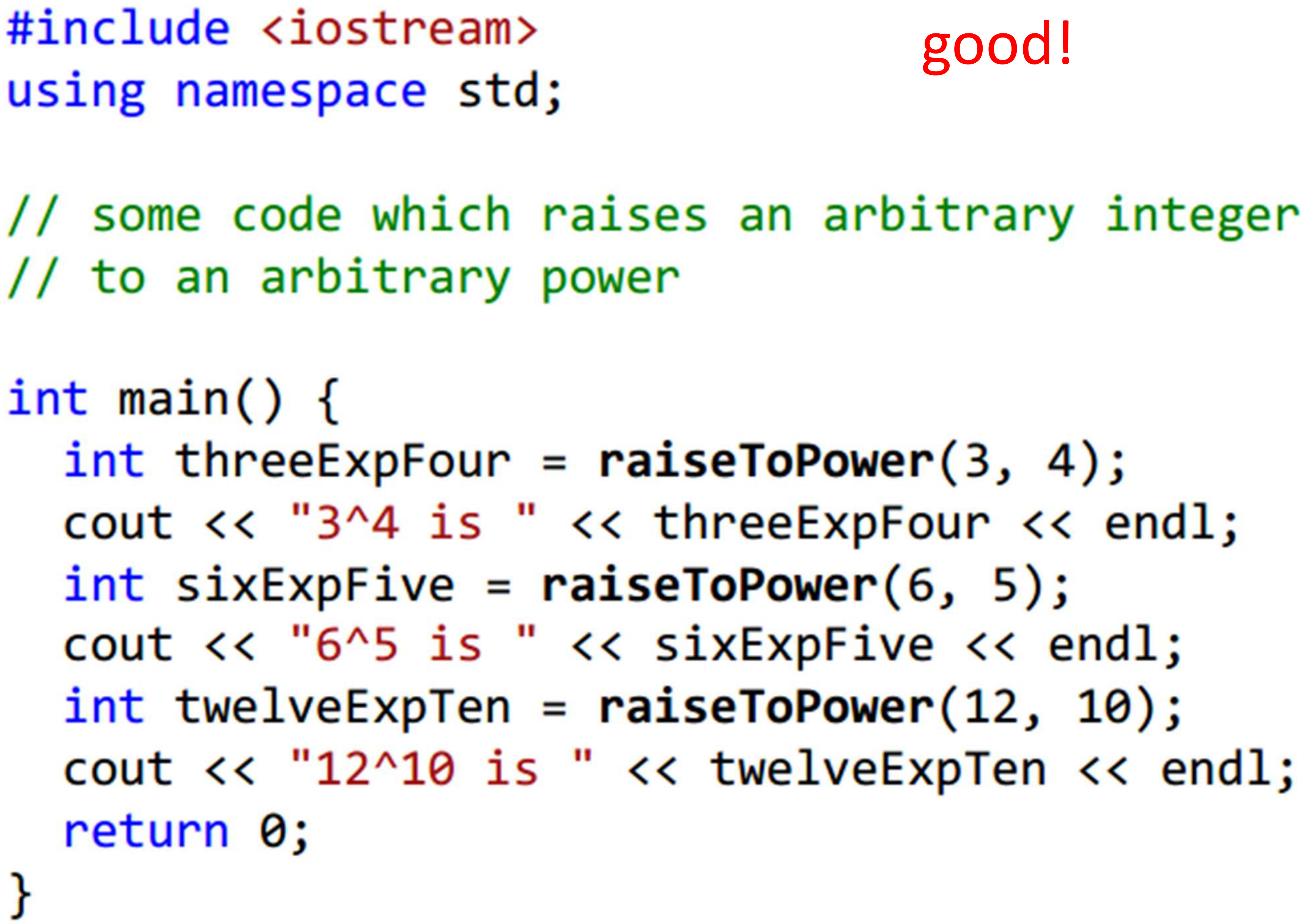
使用函数(with a function)



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

// some code which raises an arbitrary integer
// to an arbitrary power

int main() {
  int threeExpFour = raiseToPower(3, 4);
  cout << "3^4 is " << threeExpFour << endl;
  int sixExpFive = raiseToPower(6, 5);
  cout << "6^5 is " << sixExpFive << endl;
  return 0;
}
```

使用函数(with a function)

good!



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

// some code which raises an arbitrary integer
// to an arbitrary power

int main() {
  int threeExpFour = raiseToPower(3, 4);
  cout << "3^4 is " << threeExpFour << endl;
  int sixExpFive = raiseToPower(6, 5);
  cout << "6^5 is " << sixExpFive << endl;
  int twelveExpTen = raiseToPower(12, 10);
  cout << "12^10 is " << twelveExpTen << endl;
  return 0;
}
```

为什么定义自己的函数？

- 可读性：sqrt(5)比粘帖一段计算平方根的代码更清楚！
- 可维护性：改变一个算法，只需要修改这个算法的函数（相比修改每一处代码）。
- 代码重用：其他人可以使用你实现的算法！

函数定义语法

函数名

```
int raiseToPower ( int base, int exponent)
{
   int result = 1;
   for ( int i = 1; i< exponent ; i++){
      result = result*base;
  }
   return result;
}
```

函数定义语法

返回类型

```
int raiseToPower ( int base, int exponent)
{
    int result = 1;
    for ( int i = 1; i< exponent ; i++){
       result = result*base;
   }
    return result;
}
```

函数定义语法

参数1

```
int raiseToPower ( int base, int exponent)
{
    int result = 1;
    for ( int i = 1; i< exponent ; i++){
        result = result*base;
   }
    return result;
}
```

函数定义语法

参数2

```
int raiseToPower ( int base, int exponent)
{
    int result = 1;
    for ( int i = 1; i< exponent ; i++){
        result = result*base;
   }
    return result;
}
```

函数定义语法

参数列表

```
int raiseToPower ( int base, int exponent)
{
    int result = 1;
    for ( int i = 1; i< exponent ; i++){
        result = result*base;
    }
    return result;
}
```

参数是有次序的：

- raiseToPower(2,3) is 2^3=8
- raiseToPower(3,2) is 3^2=9

函数定义语法

函数签名

```
int raiseToPower ( int base, int exponent)
{
    int result = 1;
    for ( int i = 1; i< exponent ; i++){
       result = result*base;
   }
    return result;
}
```

函数定义语法

```
int raiseToPower ( int base, int exponent)
{
    int result = 1;
    for ( int i = 1; i< exponent ; i++){
       result = result*base;
   }
    return result;
}
```

函数体

函数定义语法

```
int raiseToPower ( int base, int exponent)
{
    int result = 1;
    for ( int i = 1; i< exponent ; i++){
      result = result*base;
   }
    return result;
}
```

返回语句：返回与返回类型相同的值

函数调用

函数定义

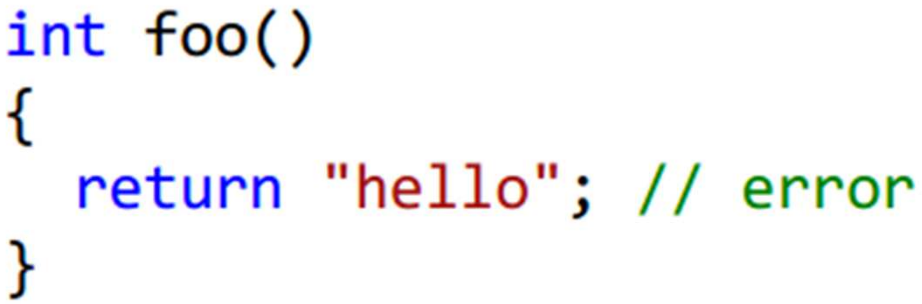
```
int raiseToPower ( int base, int exponent){
    int result = 1;
    for ( int i = 1; i< exponent ; i++){
       result = result*base;
   }
    return result;
}
```

函数调用

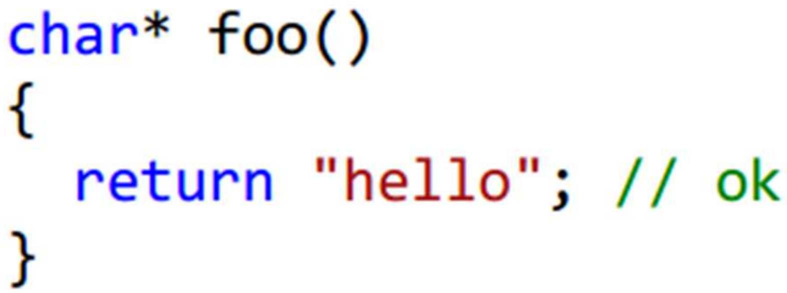
```
int main() {
    int threeExpFour = raiseToPower(3, 4) ;
    cout << "3^4 is " << threeExpFour << endl;
    return 0;
}
```

返回值

- 最多只能返回一个值，其类型需与返回类型一致.



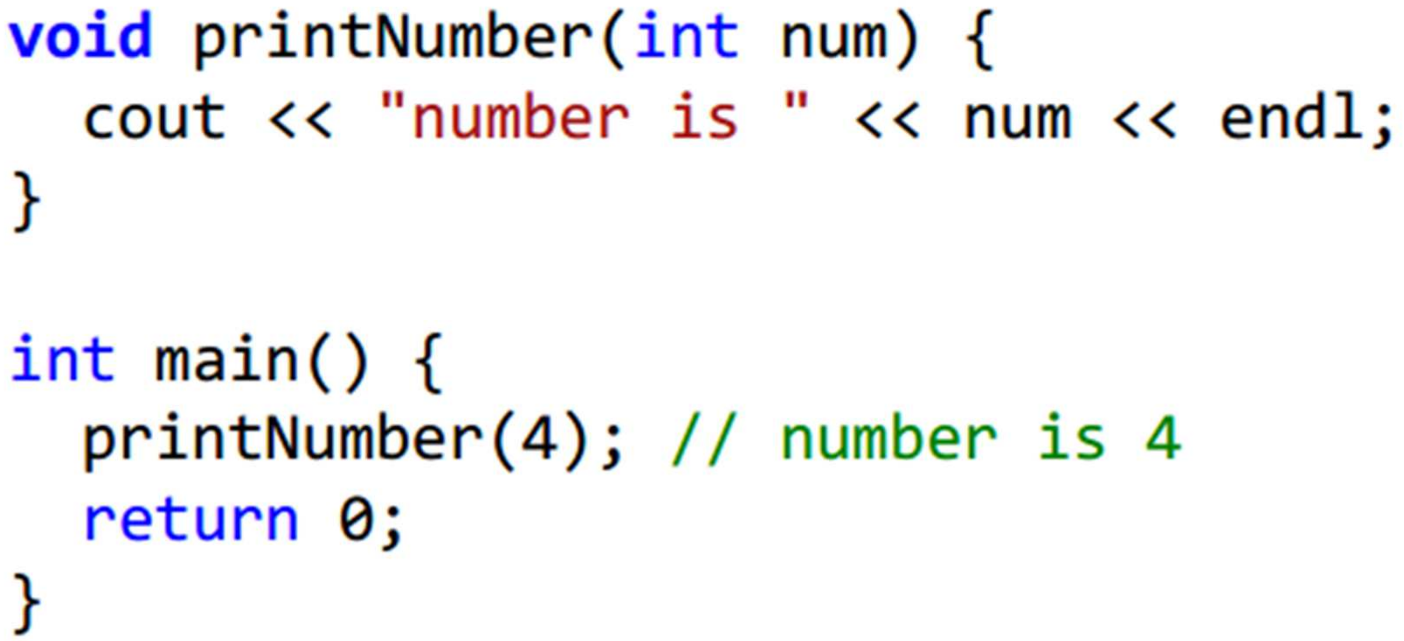
```
int foo()
{
  return "hello"; // error
}
```



```
char* foo()
{
  return "hello"; // ok
}
```

返回值

- 最多只能返回一个值，其类型需与返回类型一致.
- 如果没有返回值，则函数返回类型为void类型

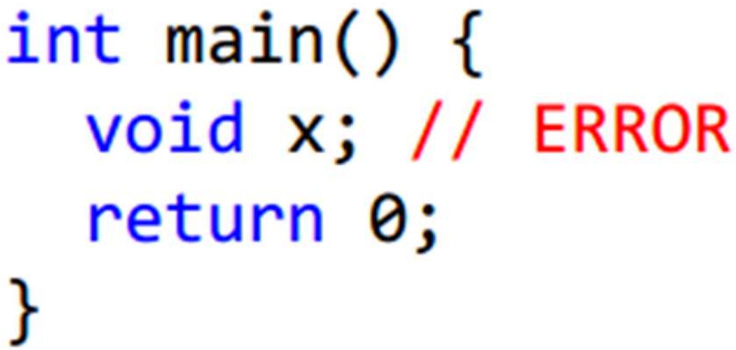


```
void printNumber(int num) {
  cout << "number is " << num << endl;
}

int main() {
  printNumber(4); // number is 4
  return 0;
}
```

返回值

- 最多只能返回一个值，其类型需与返回类型一致.
- 如果没有返回值，则函数返回类型为void类型

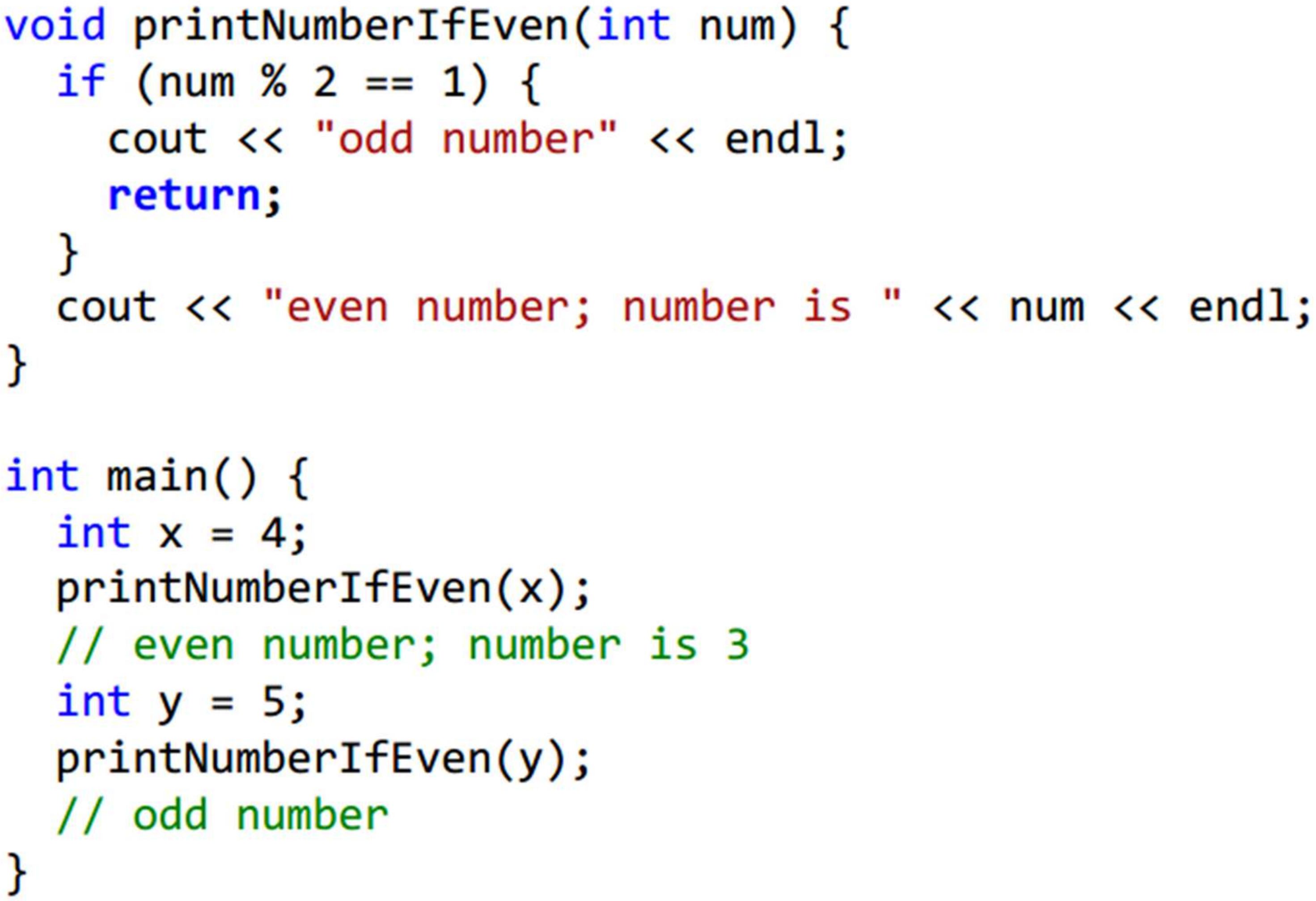
  - 注意：不能声明/定义一个void类型的变量！



```
int main() {
  void x; // ERROR
  return 0;
}
```

返回值

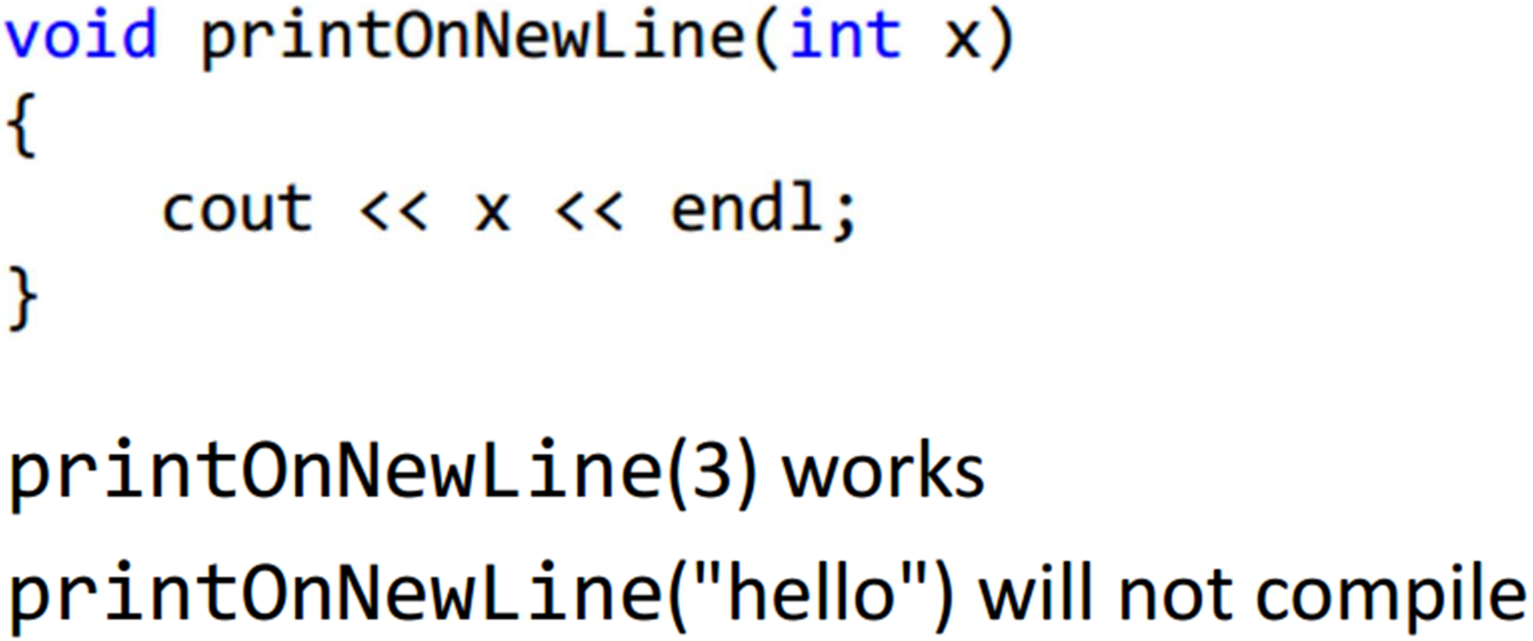
- return语句不一定在函数的最后.
- 只要执行一个return语句，函数执行就结束了！



```
void printNumberIfEven(int num) {
  if (num % 2 == 1) {
    cout << "odd number" << endl;
    return;
  }
  cout << "even number; number is " << num << endl;
}

int main() {
  int x = 4;
  printNumberIfEven(x);
  // even number; number is 3
  int y = 5;
  printNumberIfEven(y);
  // odd number
}
```

实参与形参类型要一致

- 函数定义的参数列表中变量称为形式参数(简称形参)； 而函数调用时传给函数的参数叫做实际参数(实参)。实参的类型要和对应的形参类型一致（或能转化为形参类型）.



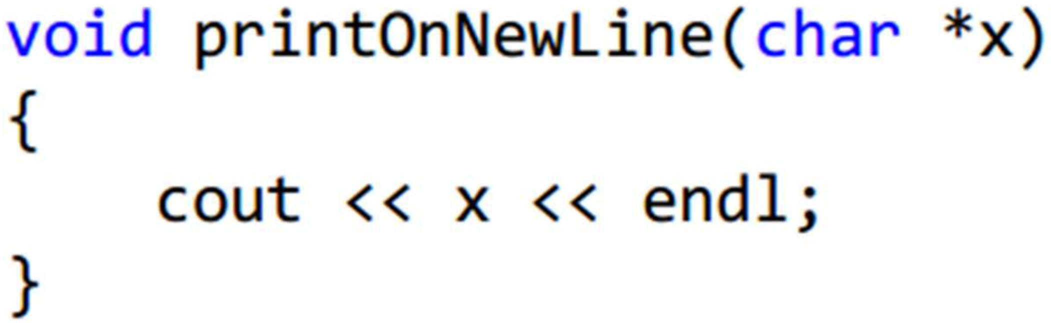
```
void printOnNewLine(int x)
{
    cout << x << endl;
}
```

printOnNewLine(3) works

printOnNewLine("hello") will not compile

实参与形参类型要一致

- 函数定义的参数列表中变量称为形式参数(简称形参)； 而函数调用时传给函数的参数叫做实际参数(实参)。实参的类型要和对应的形参类型一致（或能转化为形参类型）.



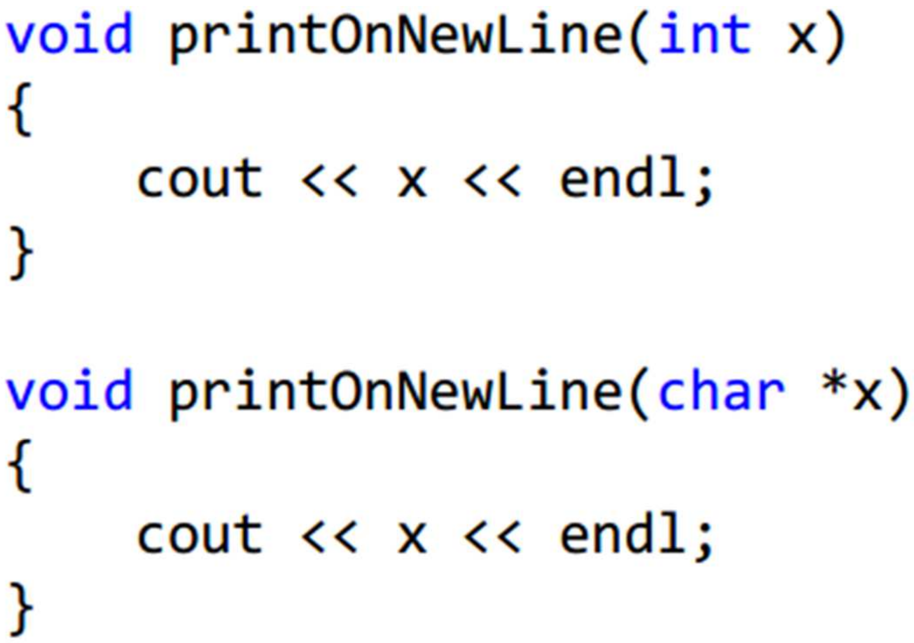
```
void printOnNewLine(char *x)
{
    cout << x << endl;
}
```

printOnNewLine(3) will not compile

printOnNewLine("hello") works

实参与形参类型要一致

- 函数定义的参数列表中变量称为形式参数(简称形参)； 而函数调用时传给函数的参数叫做实际参数(实参)。实参的类型要和对应的形参类型一致（或能转化为形参类型）.

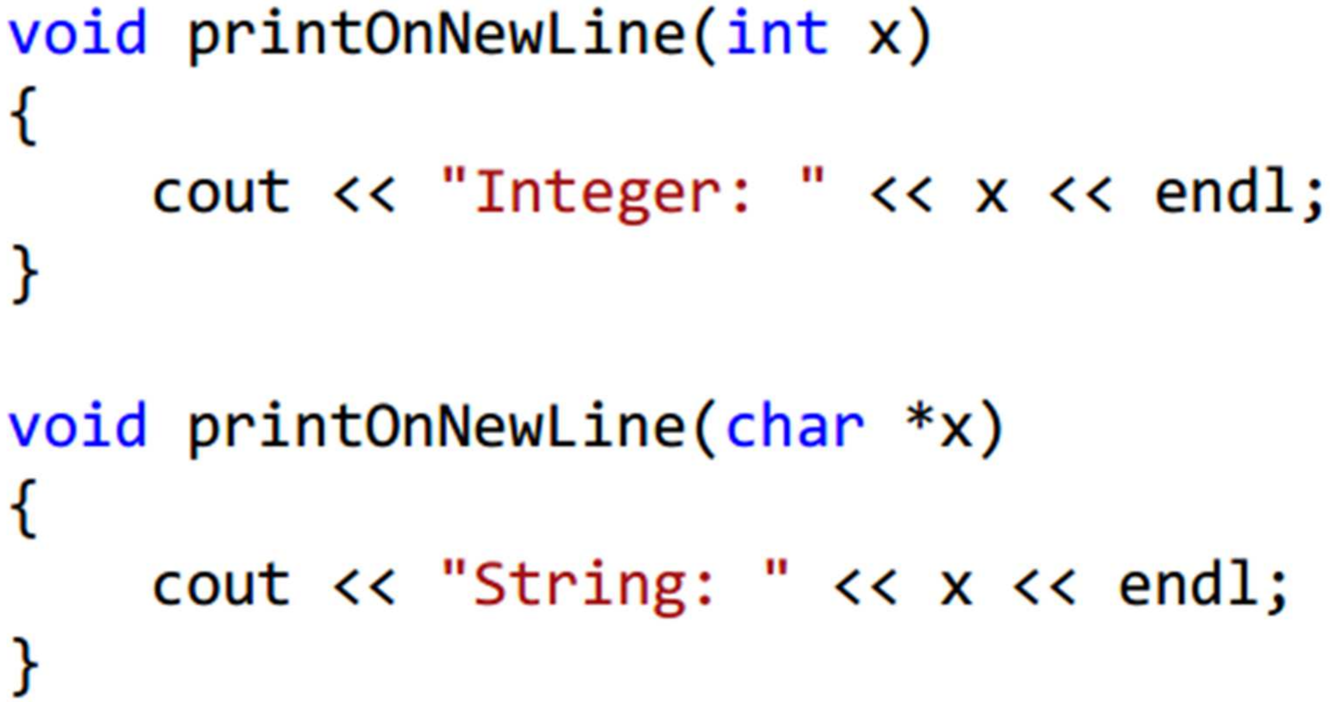


```
void printOnNewLine(int x)
{
    cout << x << endl;
}

void printOnNewLine(char *x)
{
    cout << x << endl;
}
```

printOnNewLine(3) works

printOnNewLine("hello") also works

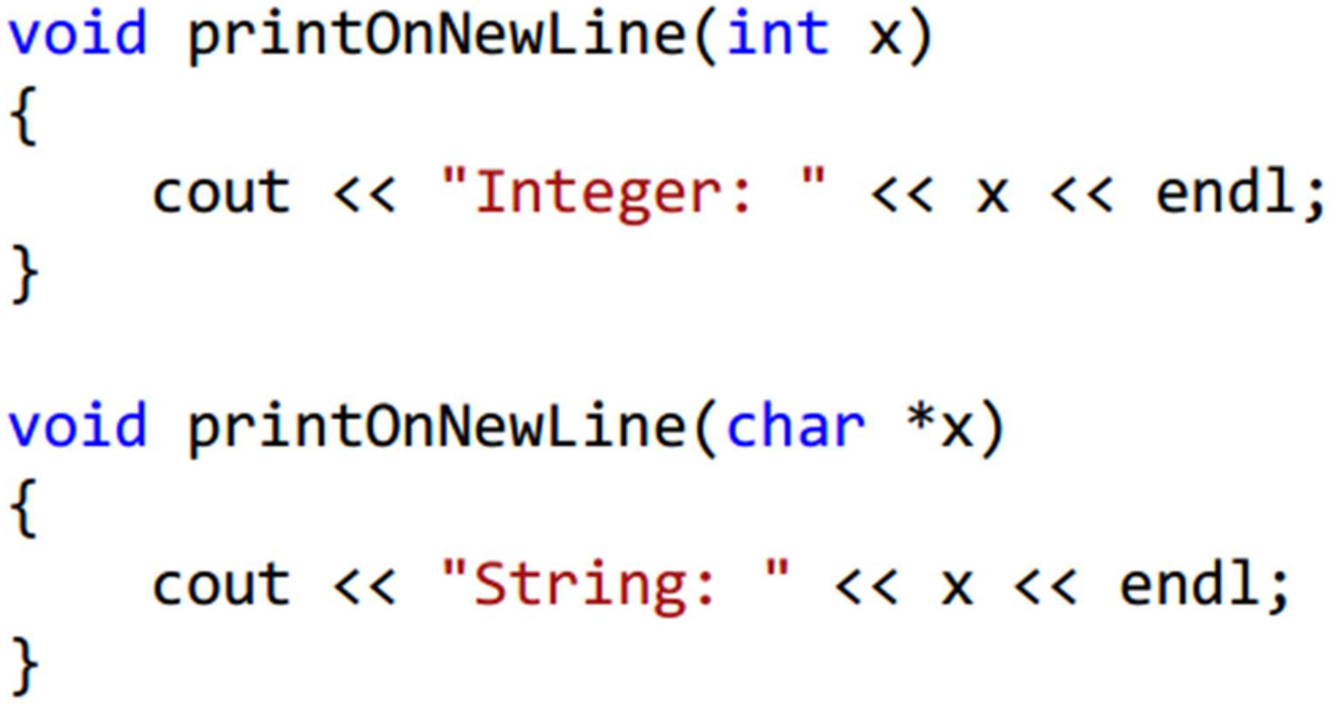
函数重载



```
void printOnNewLine(int x)
{
    cout << "Integer: " << x << endl;
}

void printOnNewLine(char *x)
{
    cout << "String: " << x << endl;
}
```

- 许多函数可以有相同的名字，但不同的参数。
- 调用时，根据参数匹配确定调用哪个函数。

函数重载

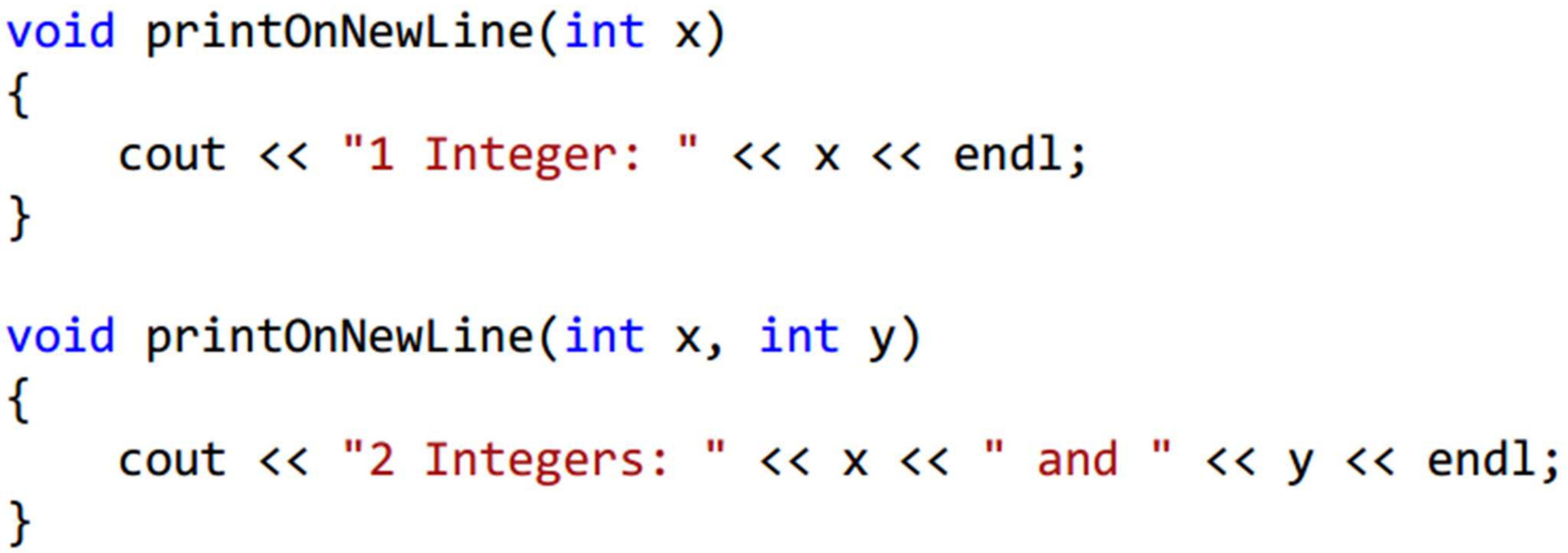


```
void printOnNewLine(int x)
{
    cout << "Integer: " << x << endl;
}

void printOnNewLine(char *x)
{
    cout << "String: " << x << endl;
}
```

printOnNewLine(3) prints “Integer: 3”

printOnNewLine(“hello”) prints “String: hello”

函数重载

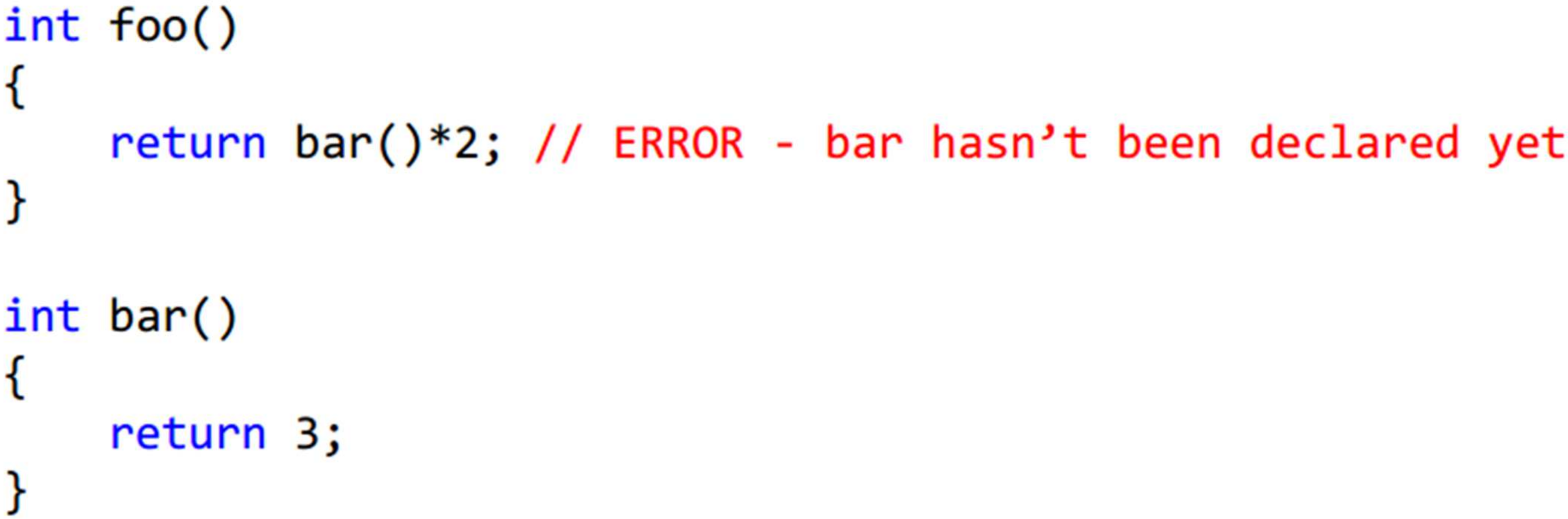


```
void printOnNewLine(int x)
{
    cout << "1 Integer: " << x << endl;
}

void printOnNewLine(int x, int y)
{
    cout << "2 Integers: " << x << " and " << y << endl;
}
```

printOnNewLine(3) prints “1 Integer: 3”

printOnNewLine(2, 3) prints “2 Integers: 2 and 3”

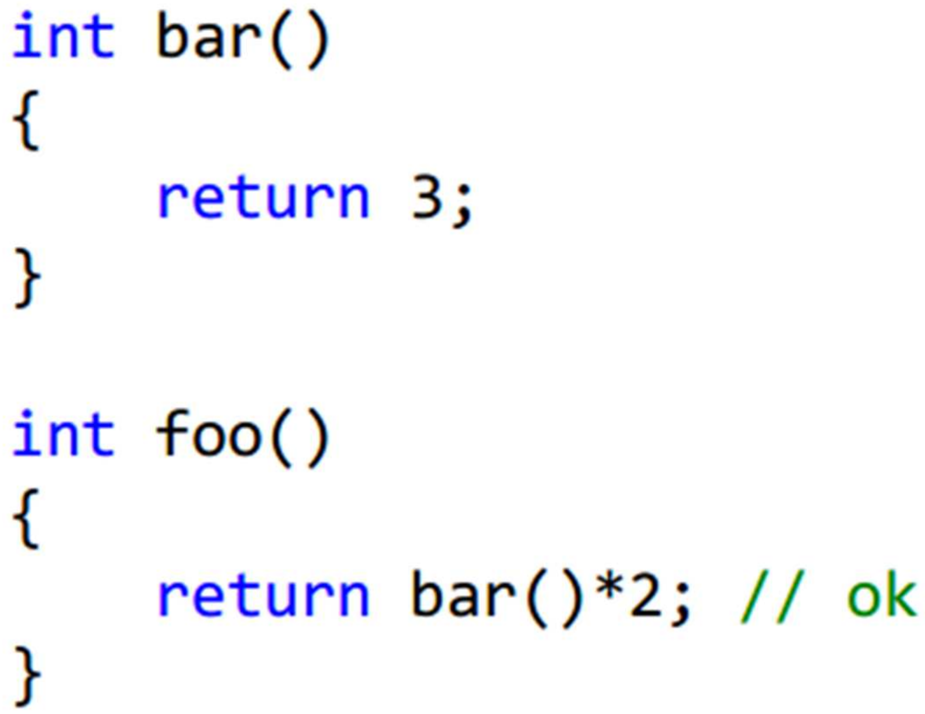
- 函数声明必须在函数被调用之前。



```
int foo()
{
    return bar()*2; // ERROR - bar hasn’t been declared yet
}

int bar()
{
    return 3;
}
```

- 函数声明必须在函数被调用之前。

-解决方法1：　改变函数声明次序

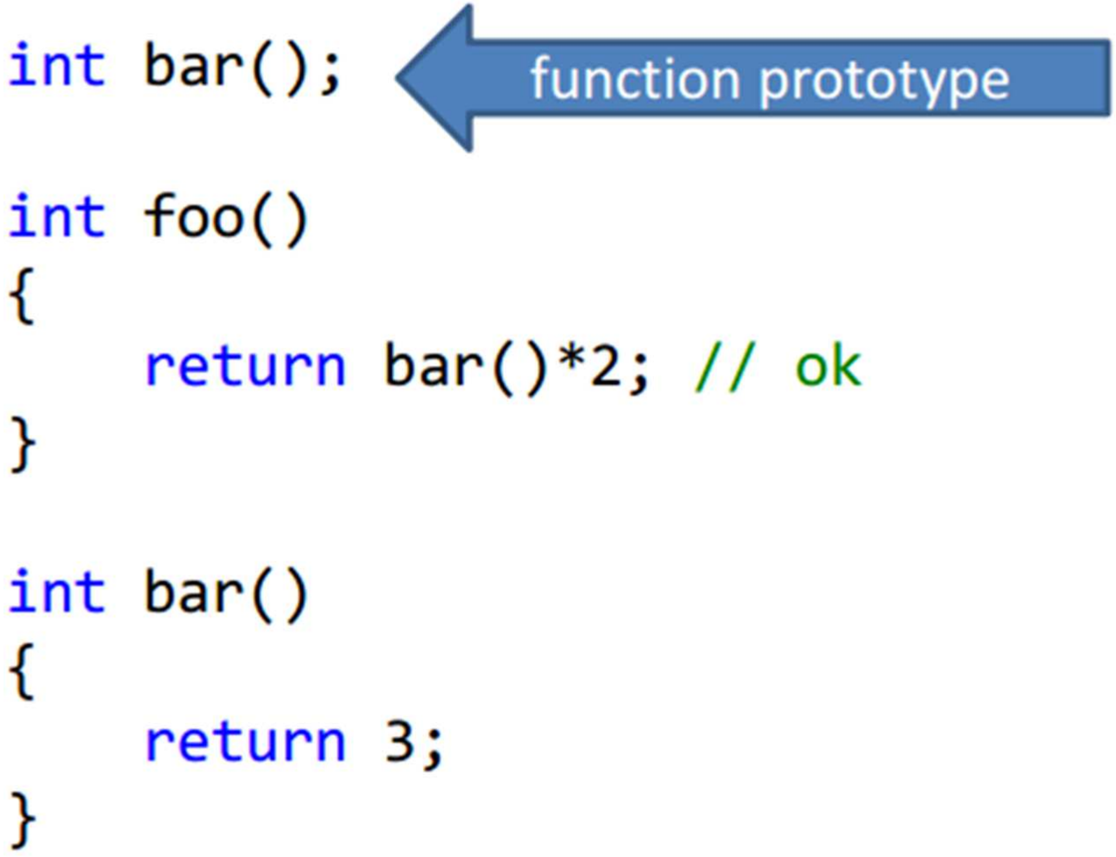


```
int bar()
{
    return 3;
}

int foo()
{
    return bar()*2; // ok
}
```

- 函数声明必须在函数被调用之前。

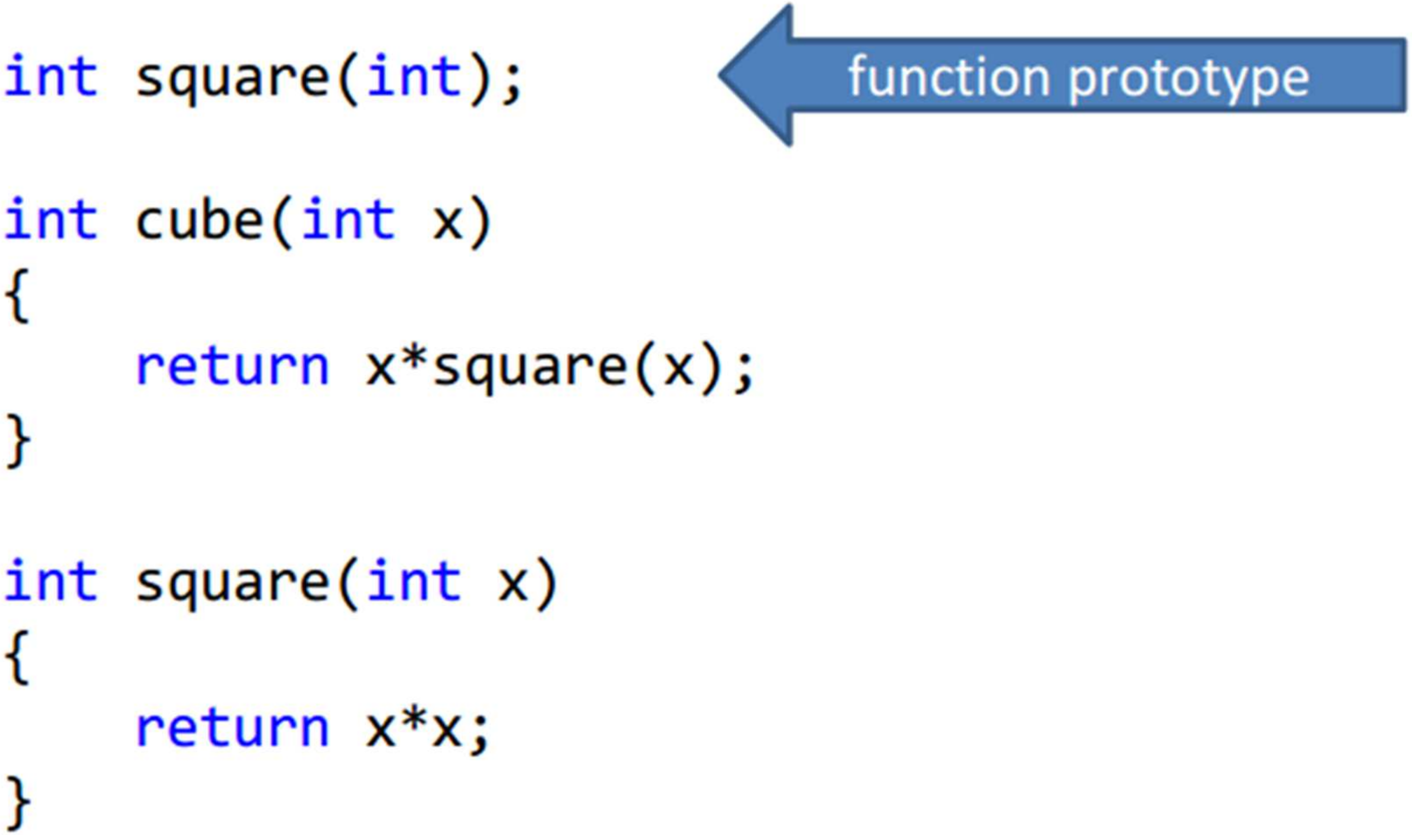
 -解决方法1： 改变函数声明次序

 -解决方法2： 使用一个函数原型(function prototype)，通知编译器将有一个这种函数。



```
int bar();

int foo()
{
    return bar()*2; // ok
}

int bar()
{
    return 3;
}
```

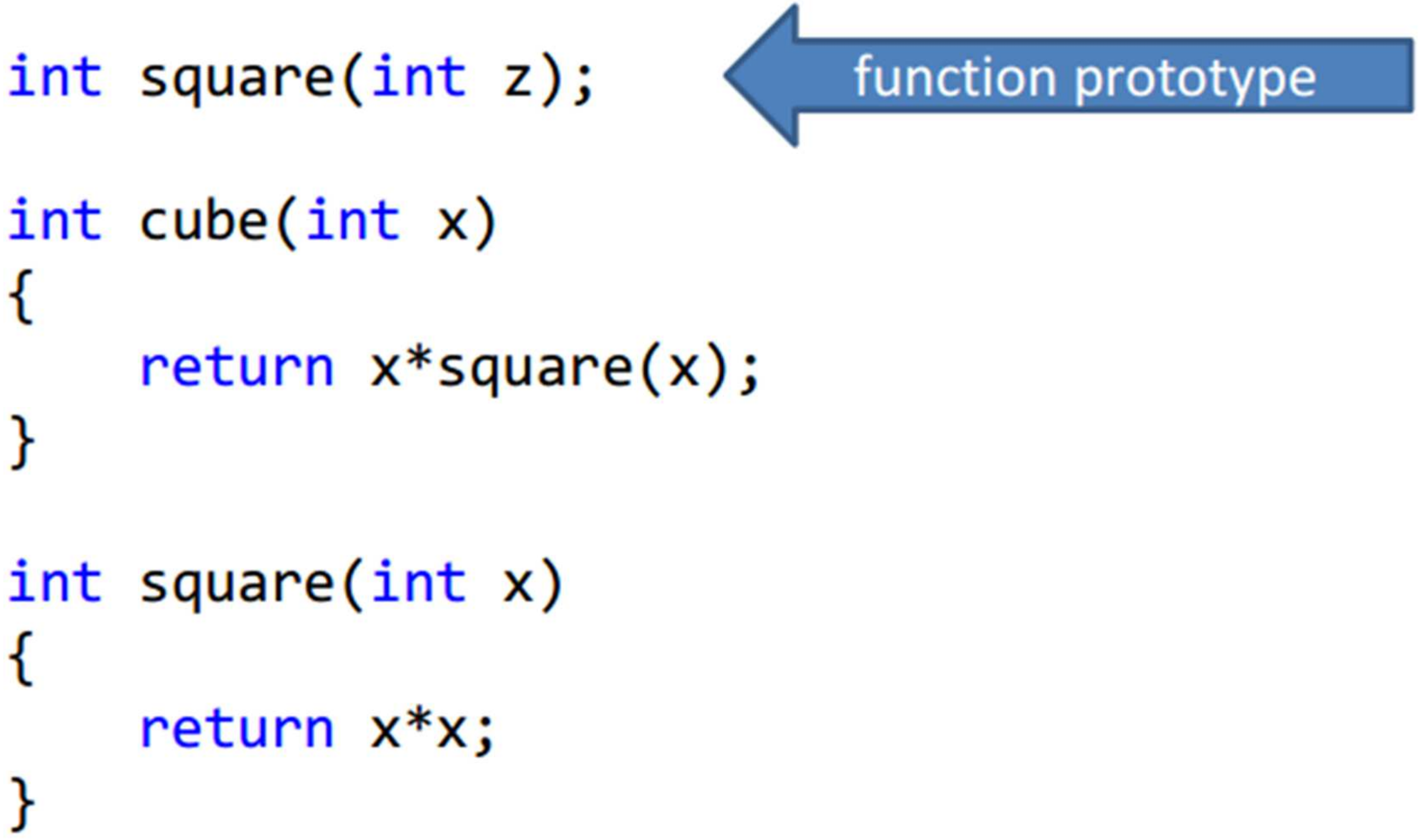
- 函数原型(prototype)必须和函数签名(signature)匹配，虽然参数名字可以变化。



```
int square(int);

int cube(int x)
{
    return x*square(x);
}

int square(int x)
{
    return x*x;
}
```

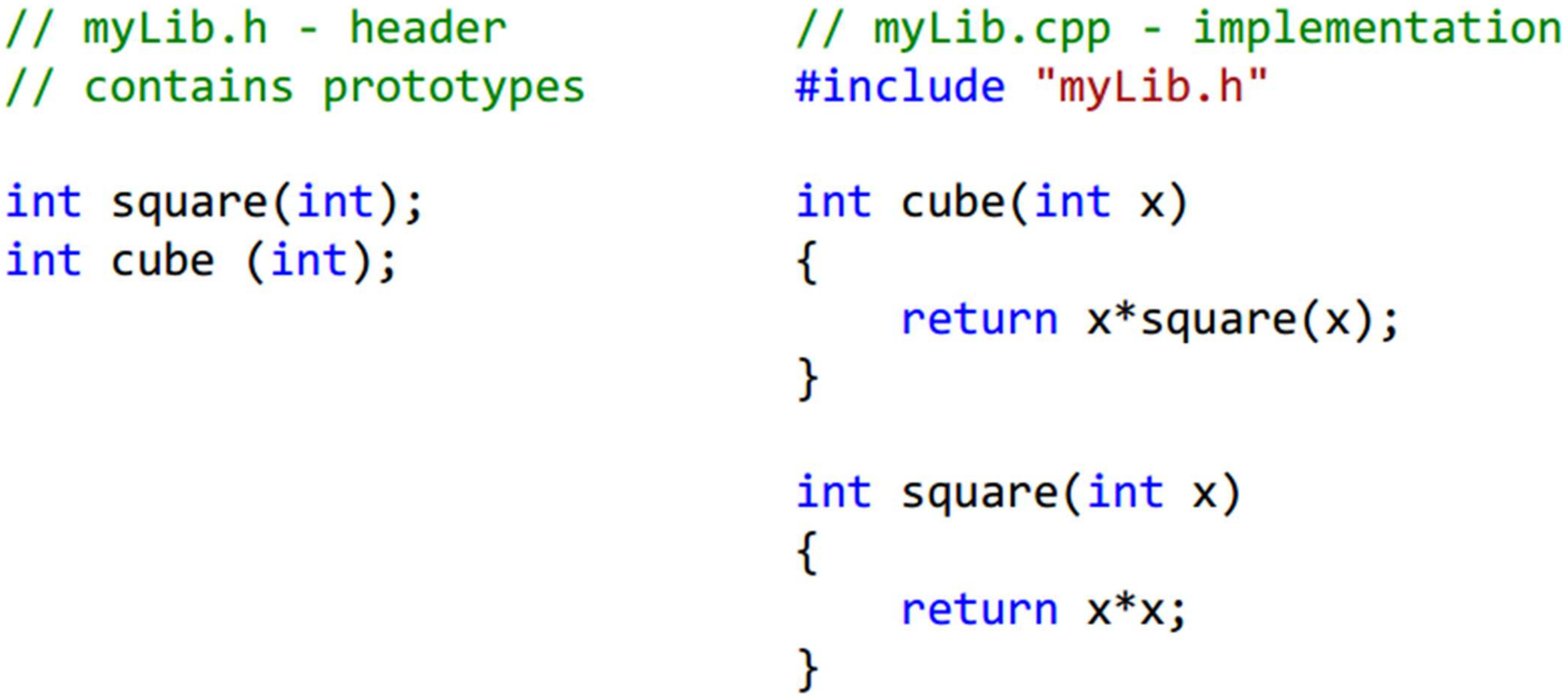
function prototype

- 函数原型(prototype)必须和函数签名(signature)匹配，虽然参数名字可以变化。



```
int square(int z);

int cube(int x)
{
    return x*square(x);
}

int square(int x)
{
    return x*x;
}
```

- 函数原型通常放在一个单独的头文件中，将函数原型和函数的实现隔离开来。

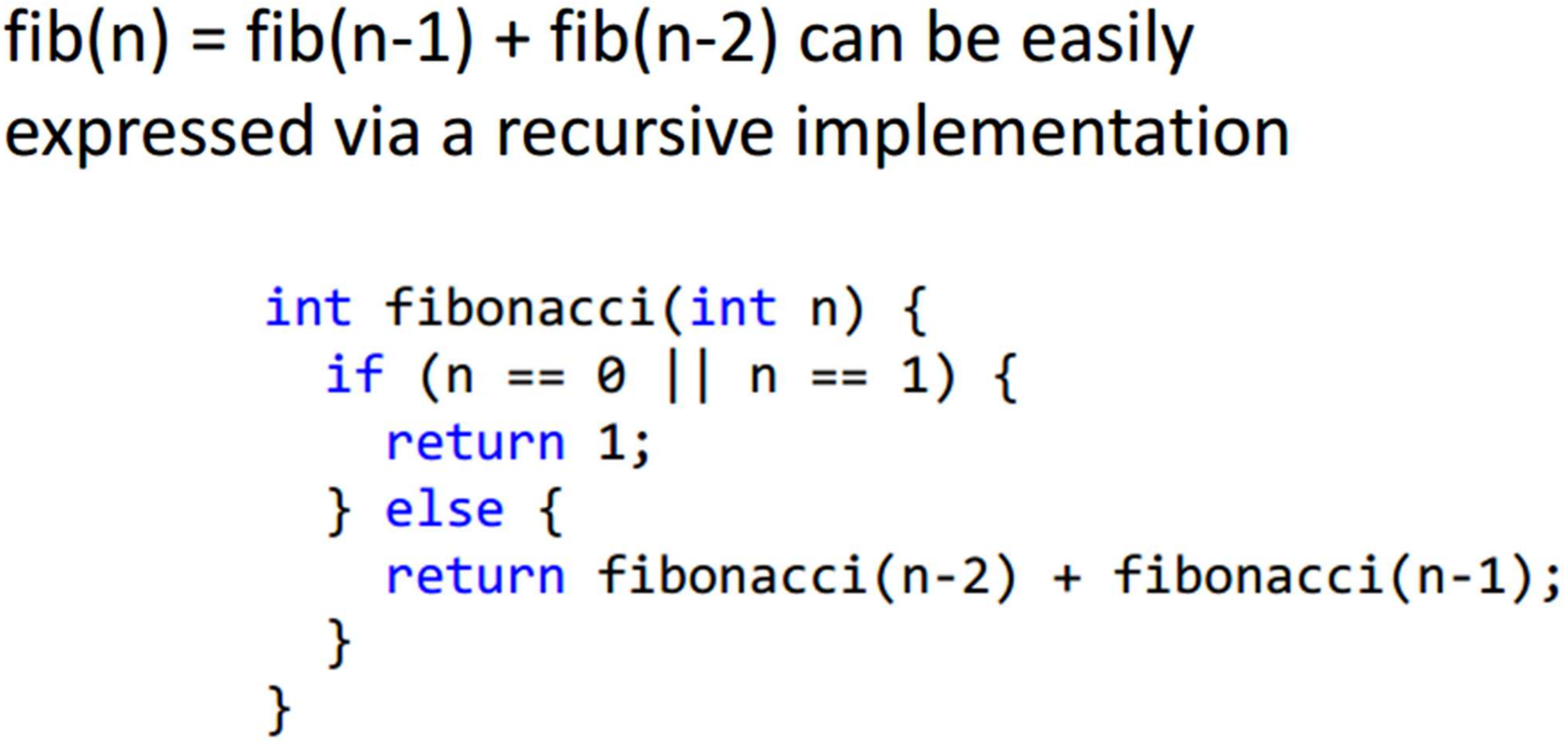


```
// myLib.h - header
// contains prototypes

int square(int);
int cube (int);
```

```
// myLib.cpp - implementation
#include "myLib.h"

int cube(int x)
{
    return x*square(x);
}

int square(int x)
{
    return x*x;
}
```

递归(Recursion)

- 函数可调用它们自身



fib(n) = fib(n-1) + fib(n-2) can be easily expressed via a recursive implementation

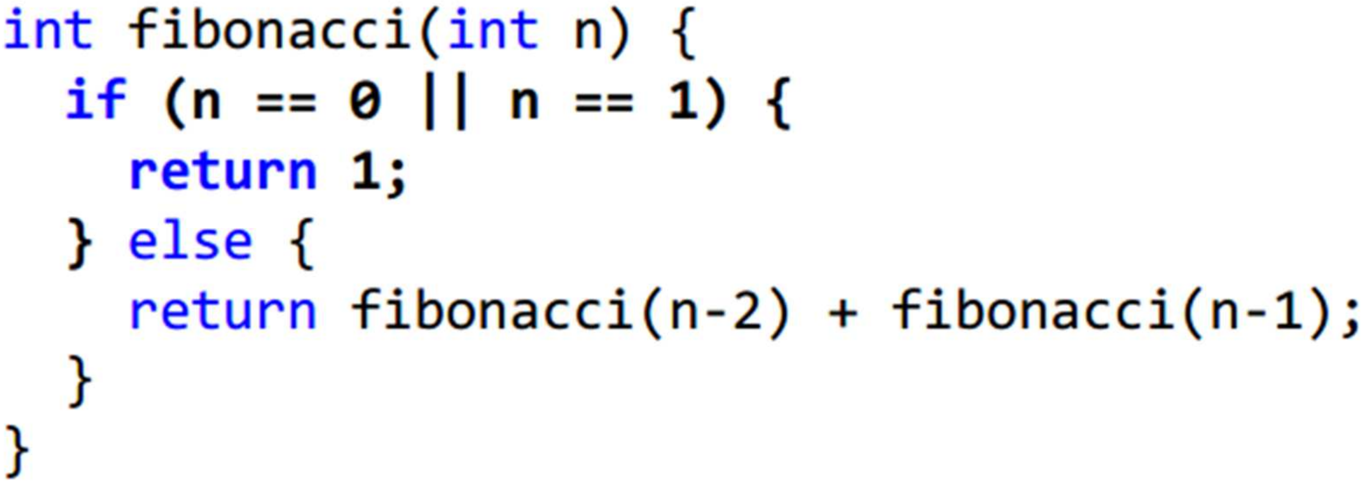
```
int fibonacci(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return fibonacci(n-2) + fibonacci(n-1);
  }
}
```

递归(Recursion)

- 函数可调用它们自身

fib(n) = fib(n-1) + fib(n-2) can be easily expressed via a recursive implementation

base case



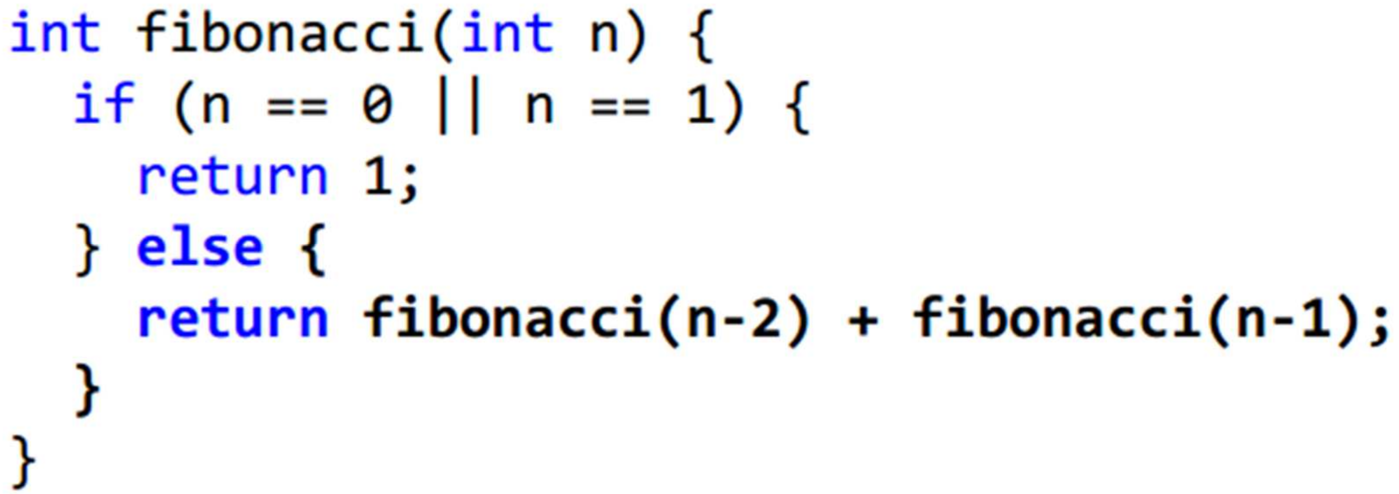
```
int fibonacci(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return fibonacci(n-2) + fibonacci(n-1);
  }
}
```

递归(Recursion)

- 函数可调用它们自身

fib(n) = fib(n-1) + fib(n-2) can be easily expressed via a recursive implementation

recursive step



```
int fibonacci(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return fibonacci(n-2) + fibonacci(n-1);
  }
}
```

全局变量(Global Variables)

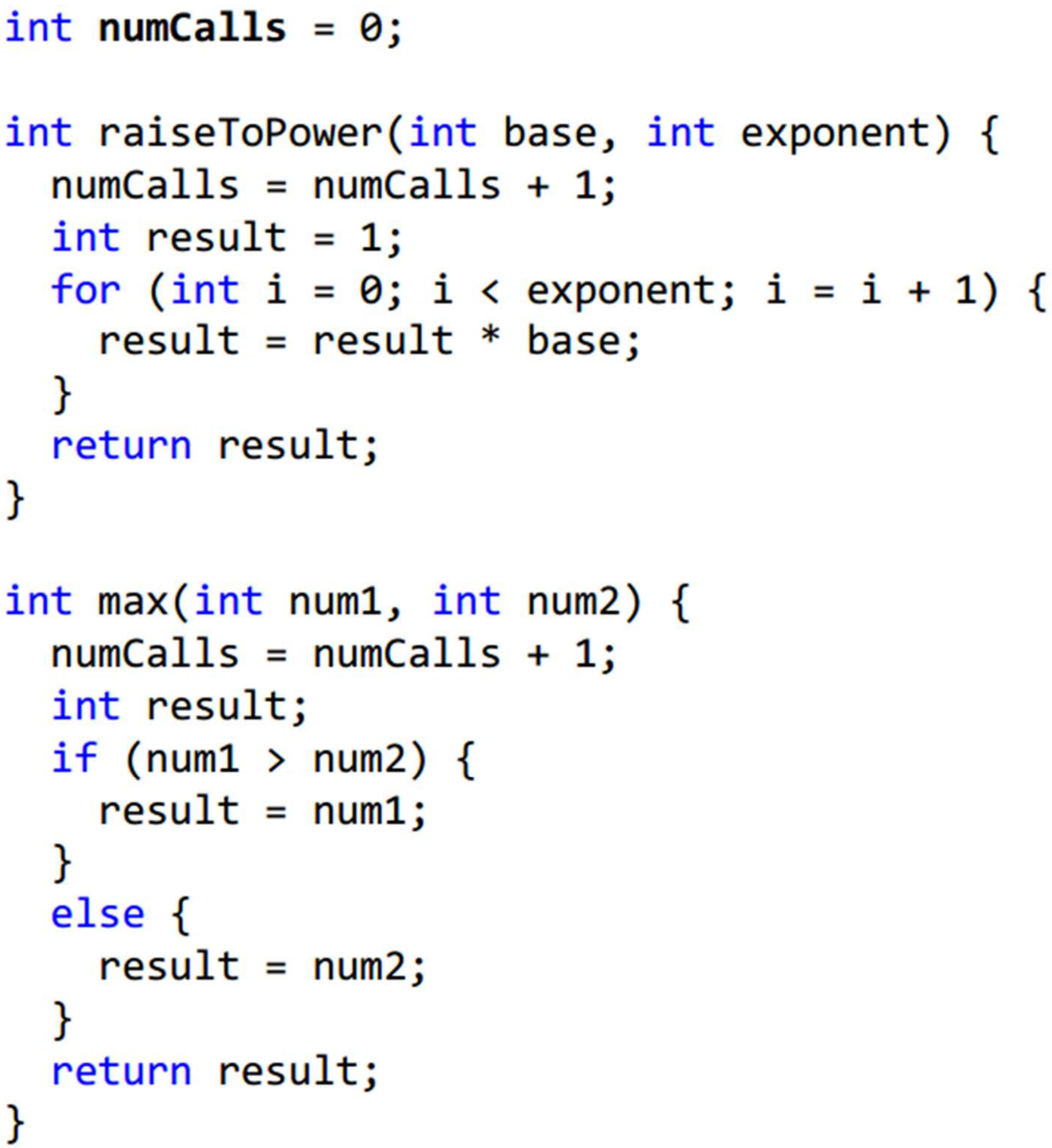
- foo()函数被调用了多少次？ 使用一个全局变量
  - 可被任何函数访问

```
int numCalls = 0;

void foo() {
 ++numCalls;
}
int main() {
   foo(); foo(); foo();
   cout << numCalls << endl; // 3
   return 0;
}
```

全局变量

作用域(Scope)

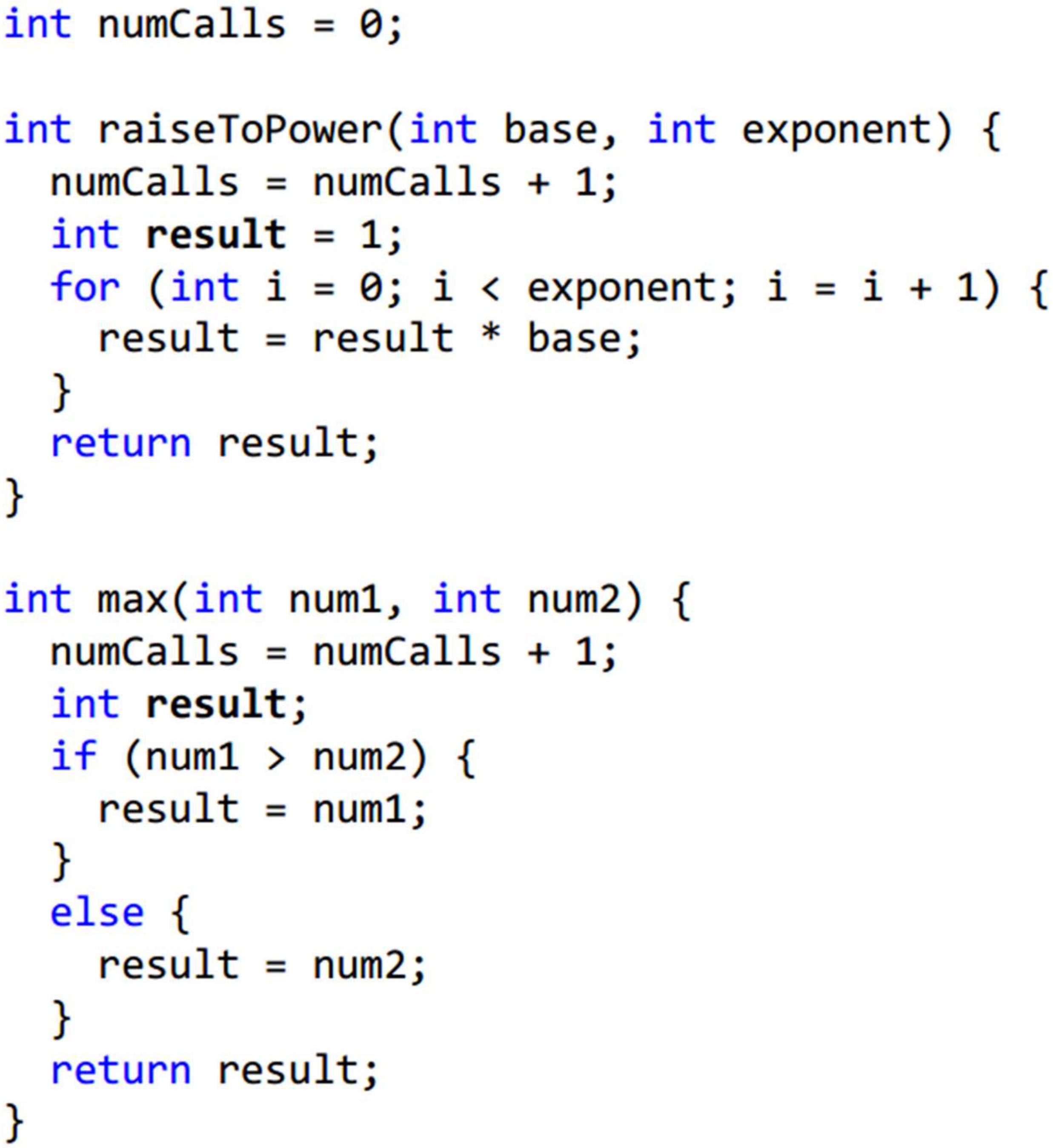
- 变量在哪里声明，决定了变量在哪里能被访问。
- numCalls具有全局作用域，能被任何函数访问。



```
int numCalls = 0;

int raiseToPower(int base, int exponent) {
  numCalls = numCalls + 1;
  int result = 1;
  for (int i = 0; i < exponent; i = i + 1) {
    result = result * base;
  }
  return result;
}

int max(int num1, int num2) {
  numCalls = numCalls + 1;
  int result;
  if (num1 > num2) {
    result = num1;
  }
  else {
    result = num2;
  }
  return result;
}
```

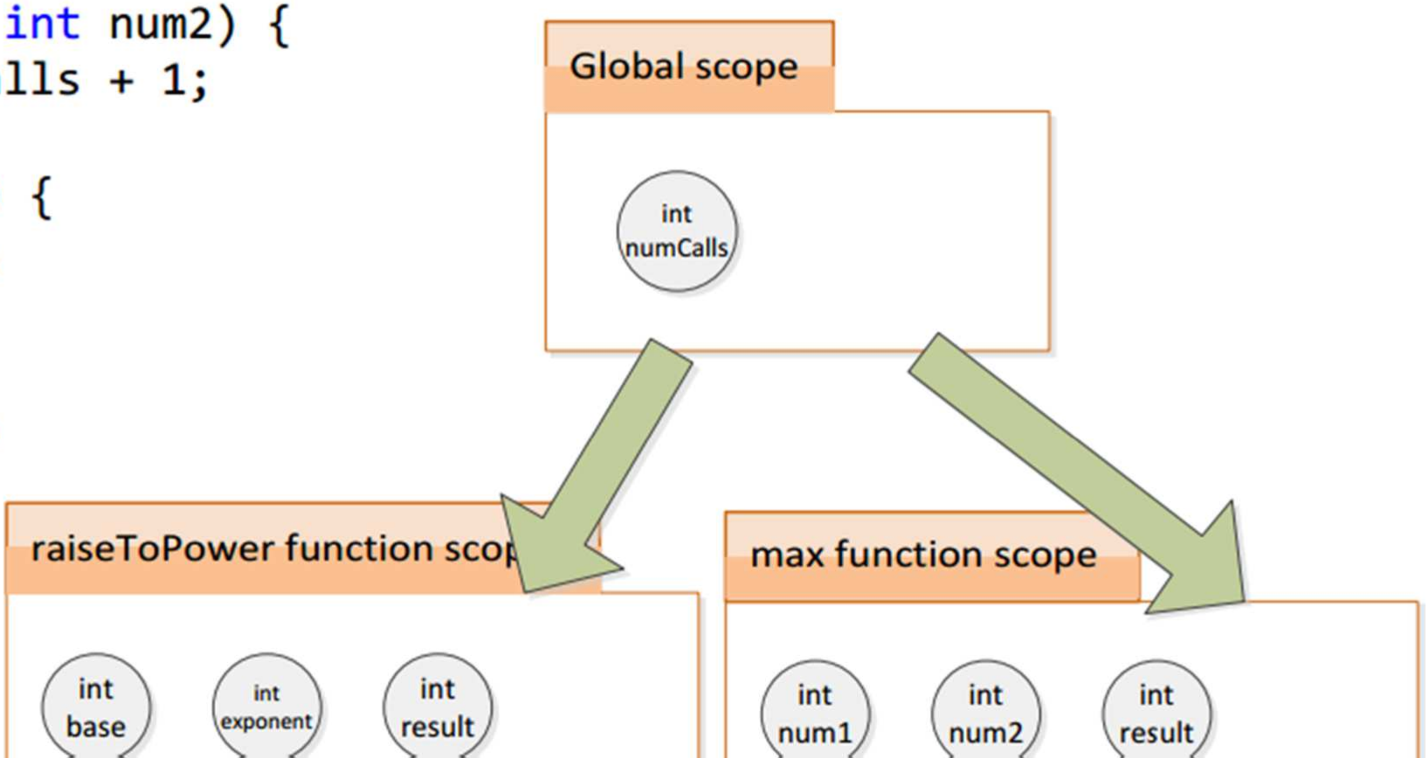
作用域(Scope)

- 变量在哪里声明，决定了变量在哪里能被访问。
- numCalls具有全局作用域，能被任何函数访问。
- result具有函数作用域，每个函数有自己单独的result变量。



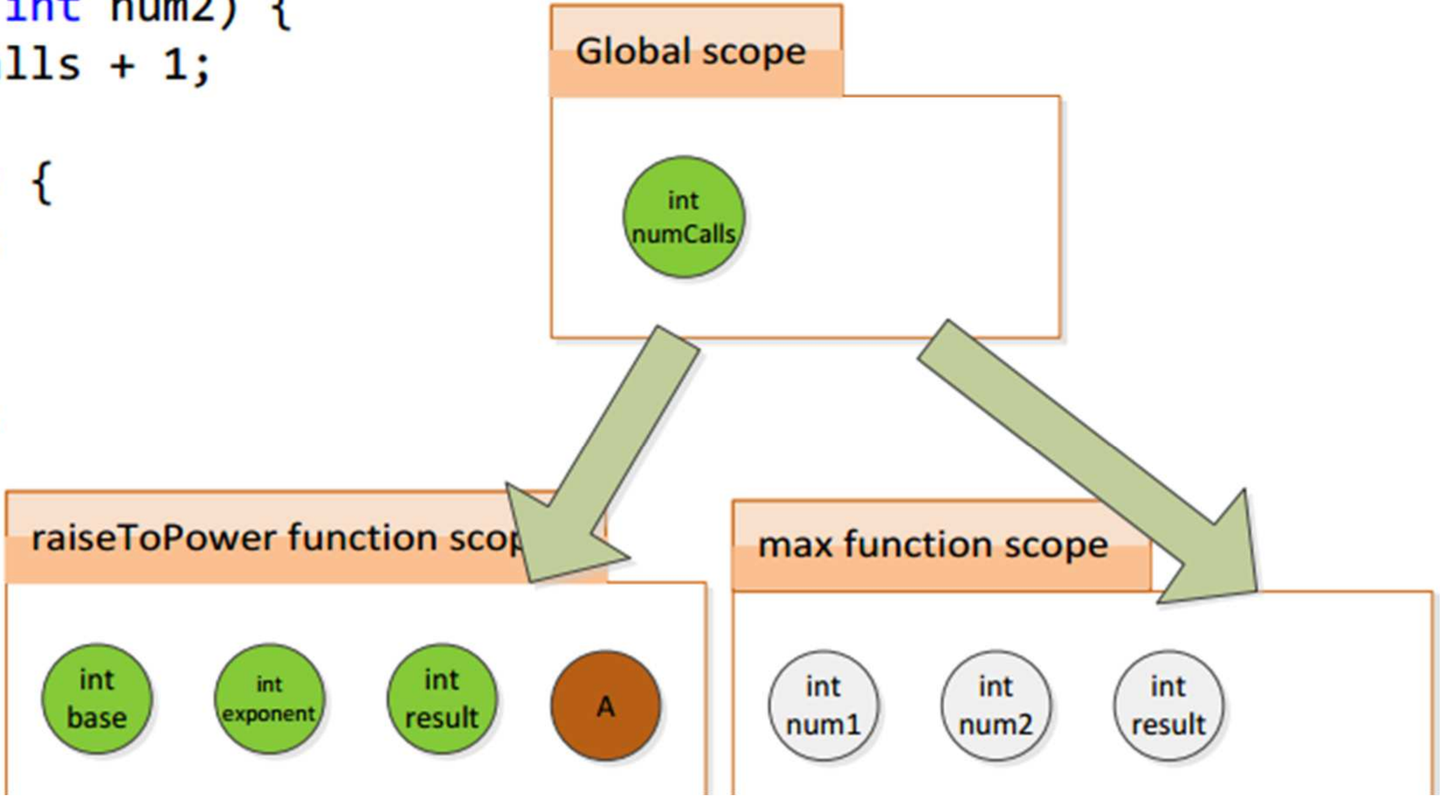
```
int numCalls = 0;

int raiseToPower(int base, int exponent) {
  numCalls = numCalls + 1;
  int result = 1;
  for (int i = 0; i < exponent; i = i + 1) {
    result = result * base;
  }
  return result;
}

int max(int num1, int num2) {
  numCalls = numCalls + 1;
  int result;
  if (num1 > num2) {
    result = num1;
  }
  else {
    result = num2;
  }
  return result;
}
```

```
int raiseToPower(int base, int exponent) {
  numCalls = numCalls + 1;
  int result = 1;
  for (int i = 0; i < exponent; i = i + 1) {
    result = result * base;
  }
  // A
  return result;
}
int max(int num1, int num2) {
  numCalls = numCalls + 1;
  int result;
  if (num1 > num2) {
    result = num1;
  }
  else {
    result = num2;
  }
  // B
  return result;
}
```



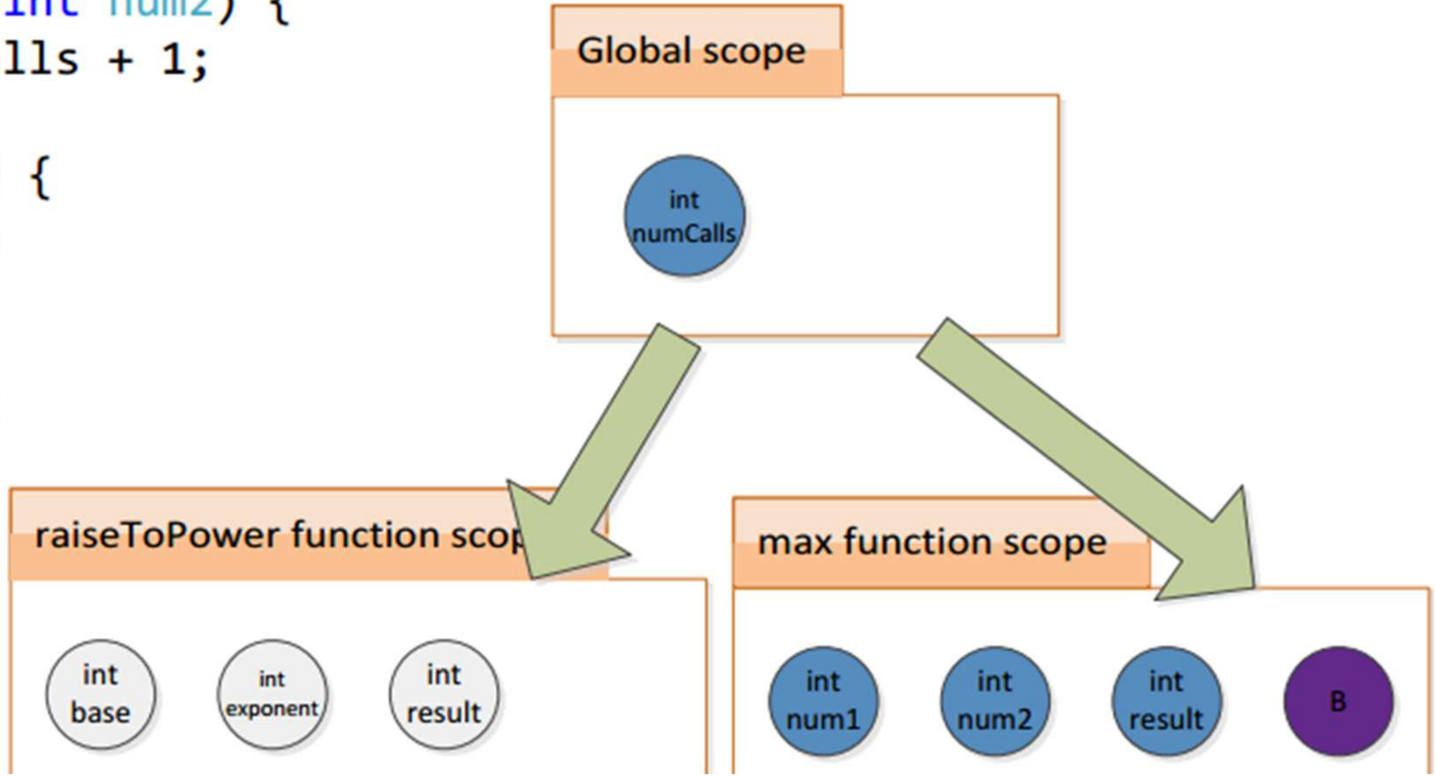
```
int numCalls = 0;
int raiseToPower(int base, int exponent) {
  numCalls = numCalls + 1;
  int result = 1;
  for (int i = 0; i < exponent; i = i + 1) {
    result = result * base;
  }
  // A
  return result;
}
int max(int num1, int num2) {
  numCalls = numCalls + 1;
  int result;
  if (num1 > num2) {
    result = num1;
  }
  else {
    result = num2;
  }
  // B
  return result;
}
```

- At A, variables marked in green are in scope



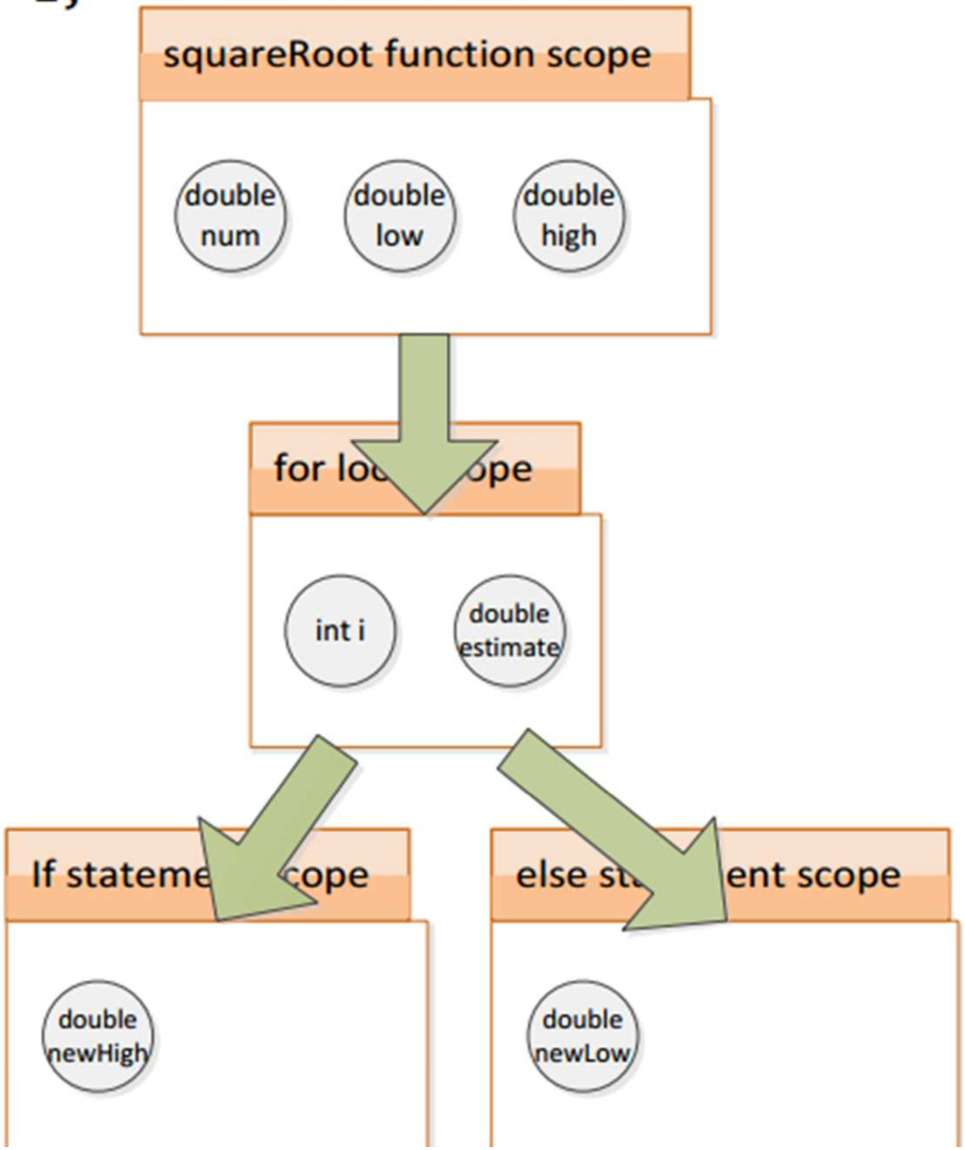
```
int numCalls = 0;
int raiseToPower(int base, int exponent) {
  numCalls = numCalls + 1;
  int result = 1;
  for (int i = 0; i < exponent; i = i + 1) {
    result = result * base;
  }
  // A
  return result;
}
int max(int num1, int num2) {
  numCalls = numCalls + 1;
  int result;
  if (num1 > num2) {
    result = num1;
  }
  else {
    result = num2;
  }
  // B
  return result;
}
```

- At B, variables marked in blue are in scope

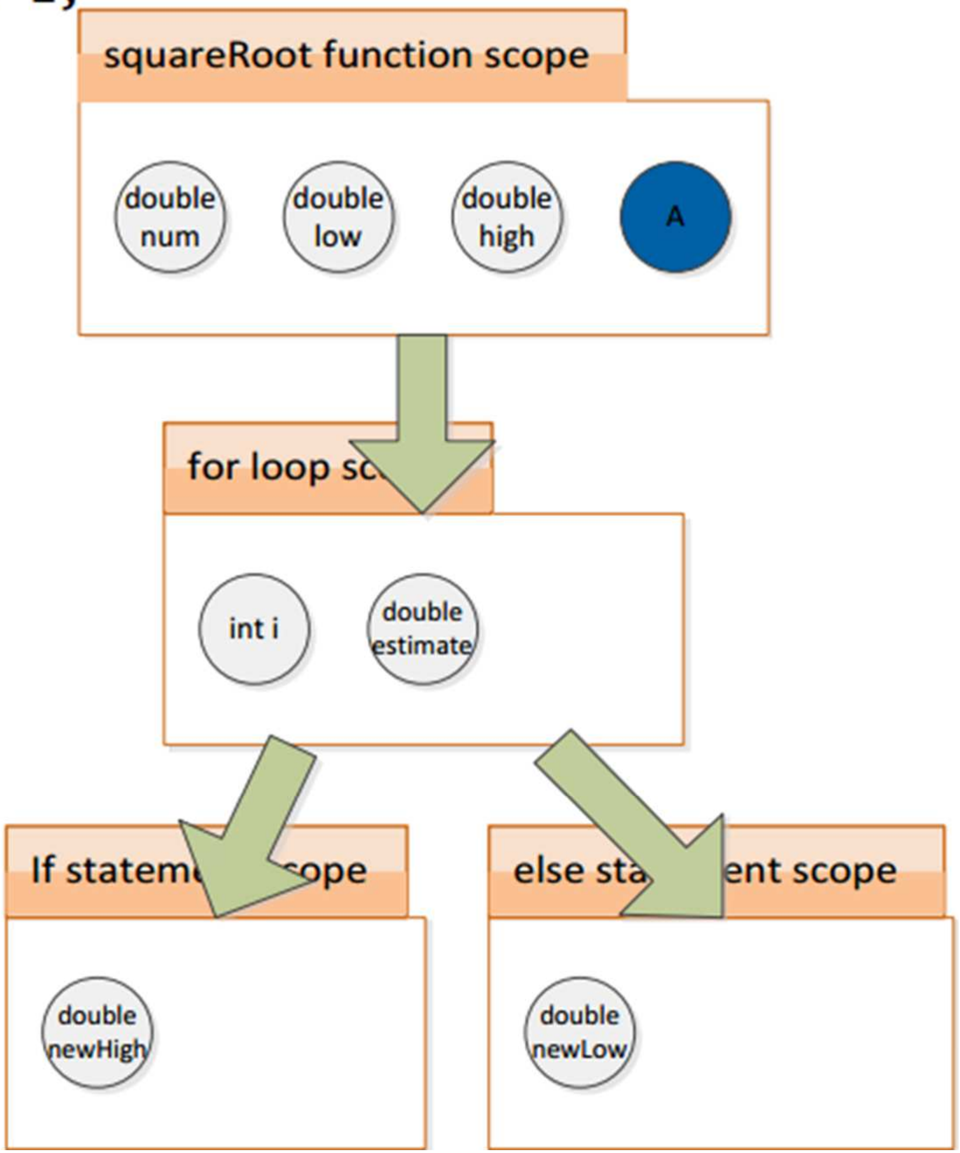


```
double squareRoot(double num) {
  double low = 1.0;
  double high = num;
  for (int i = 0; i < 30; i = i + 1) {
    double estimate = (high + low) / 2;
    if (estimate*estimate > num) {
      double newHigh = estimate;
      high = newHigh;
    } else {
      double newLow = estimate;
      low = newLow;
    }
  }
  return (high + low) / 2;
}
```

- Loops and if/else statements also have their own scopes
  - Loop counters are in the same scope as the body of the for loop



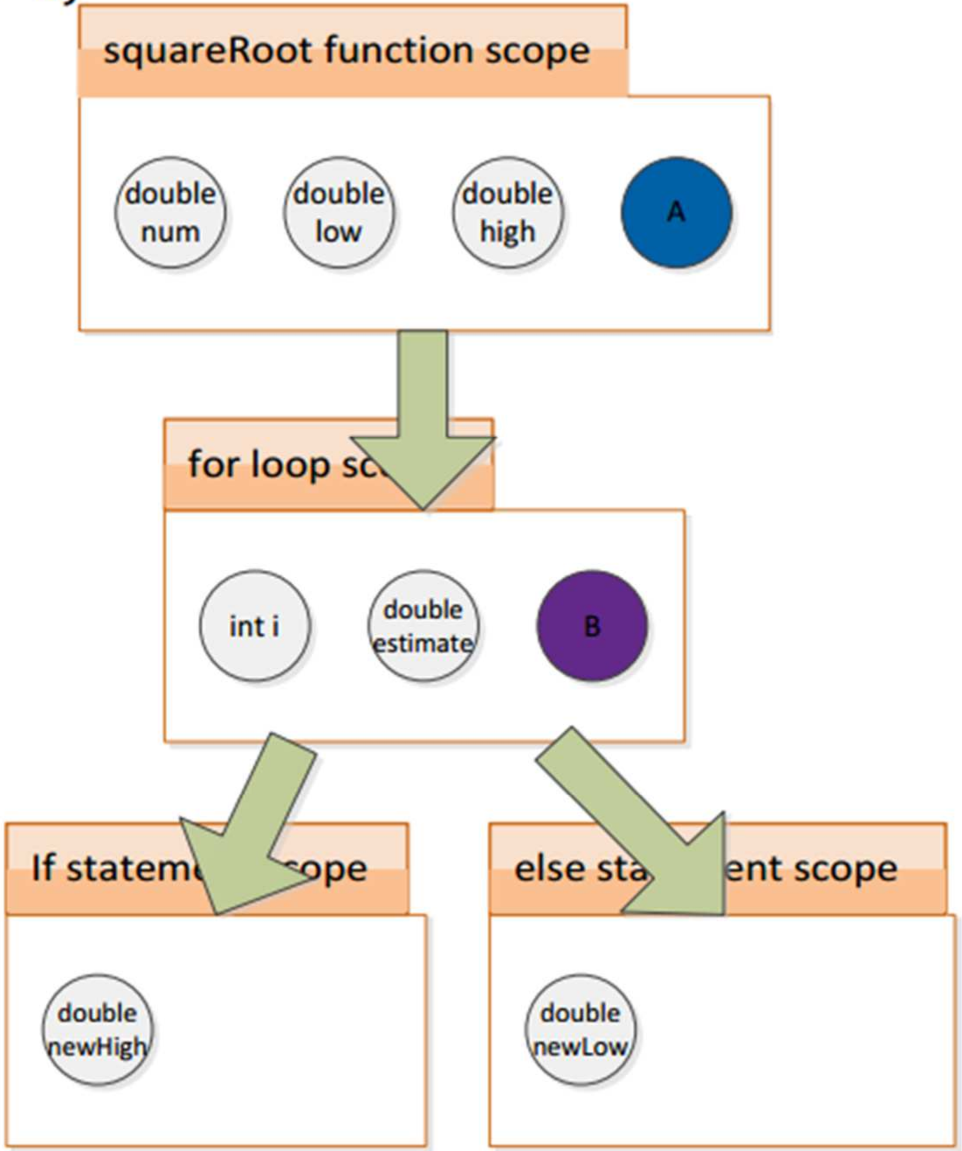
```
double squareRoot(double num) {
  double low = 1.0;
  double high = num;
  for (int i = 0; i < 30; i = i + 1) {
    double estimate = (high + low) / 2;
    if (estimate*estimate > num) {
      double newHigh = estimate;
      high = newHigh;
    } else {
      double newLow = estimate;
      low = newLow;
    }
  }
  // A
  return estimate; // ERROR
}
```



- Cannot access variables that are out of scope

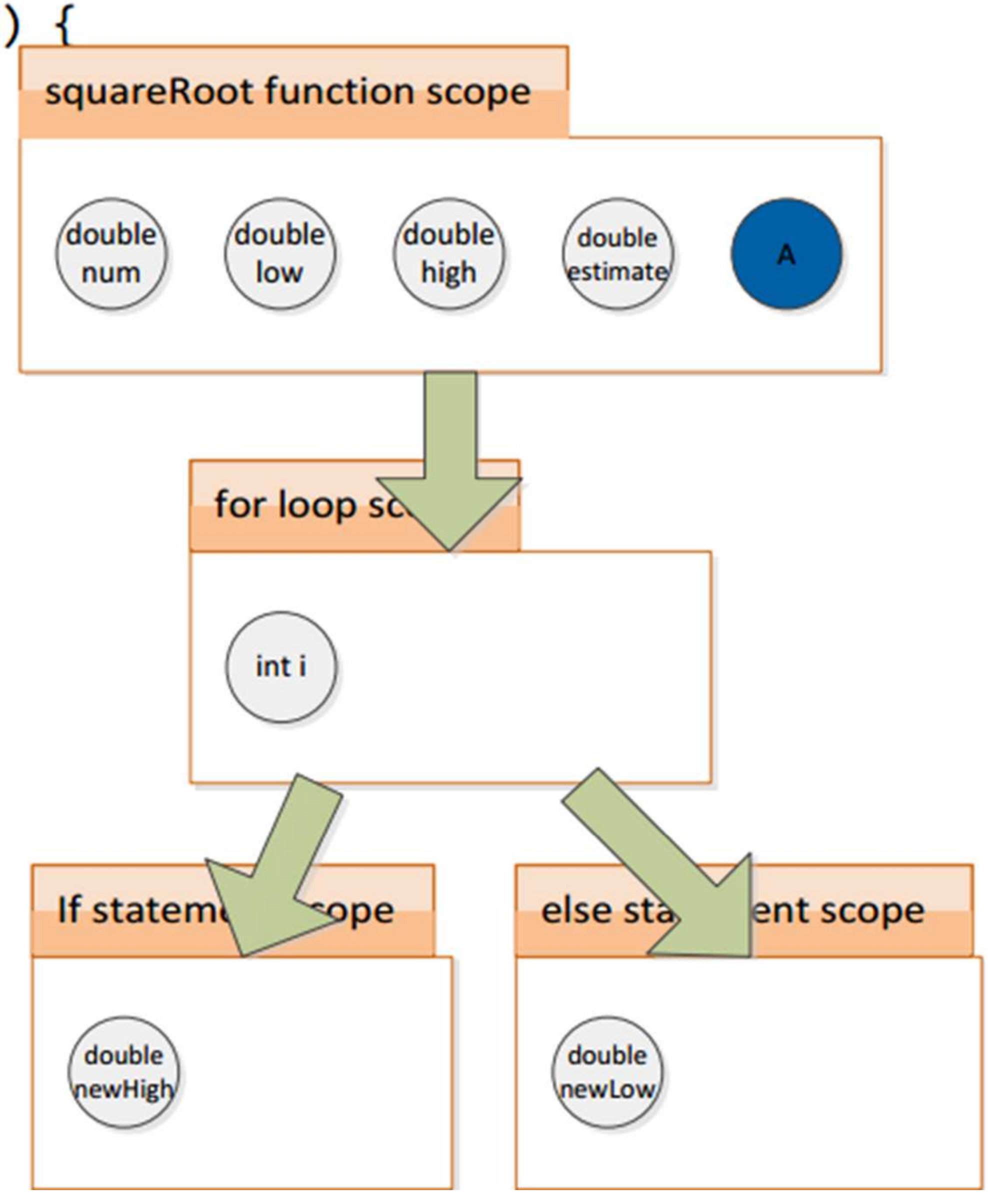
```
double squareRoot(double num) {
  double low = 1.0;
  double high = num;
  for (int i = 0; i < 30; i = i + 1) {
    double estimate = (high + low) / 2;
    if (estimate*estimate > num) {
      double newHigh = estimate;
      high = newHigh;
    } else {
      double newLow = estimate;
      low = newLow;
    }
    if (i == 29)
      return estimate; // B
  }
  return -1; // A
}
```

- Cannot access variables that are out of scope
- Solution 1: move the code



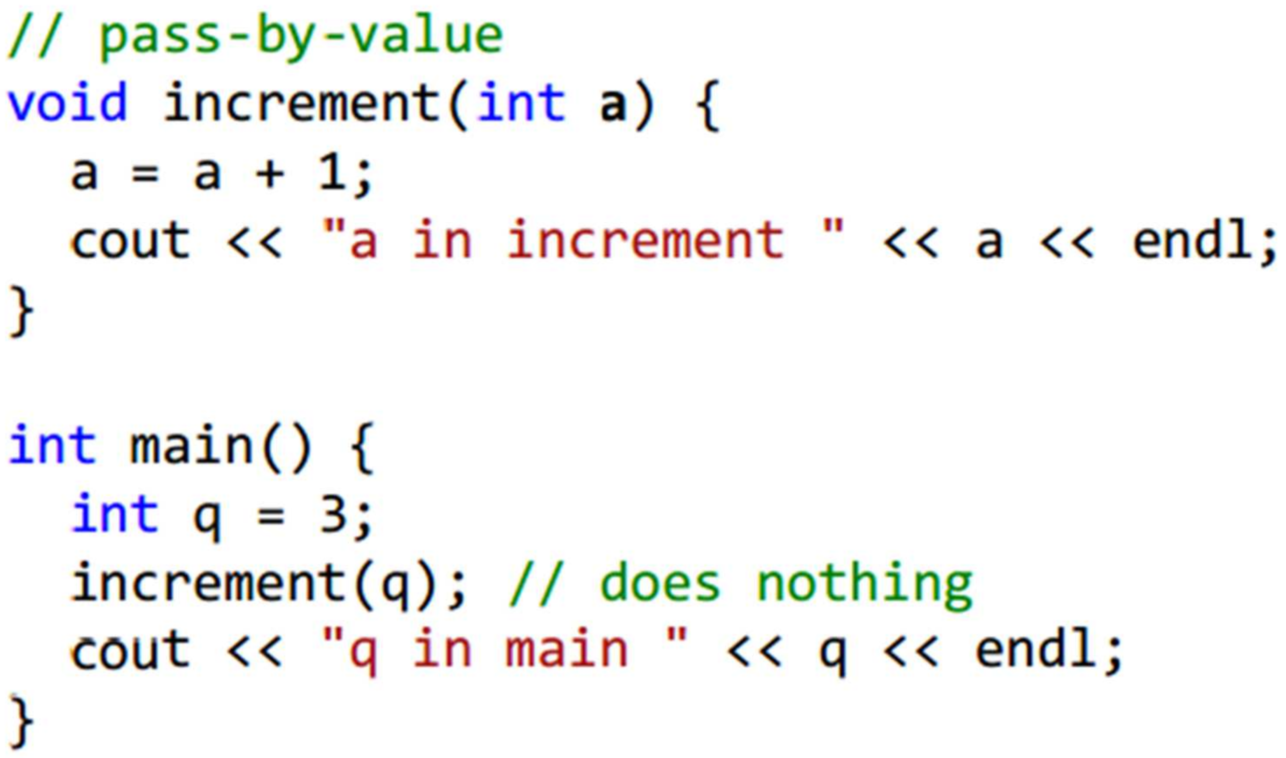
```
double squareRoot(double num) {
  double low = 1.0;
  double high = num;
  double estimate;
  for (int i = 0; i < 30; i = i + 1) {
    estimate = (high + low) / 2;
    if (estimate*estimate > num) {
      double newHigh = estimate;
      high = newHigh;
    } else {
      double newLow = estimate;
      low = newLow;
    }
  }
  return estimate; // A
}
```

- Cannot access variables that are out of scope
- Solution 2: declare the variable in a higher scope



传值(value)还是传引用(reference)
Pass by value vs by reference

- 目前调用函数都是传值-形参是实参的拷贝. 函数内对形参变量的修改不会影响函数外的变量(包括实参).



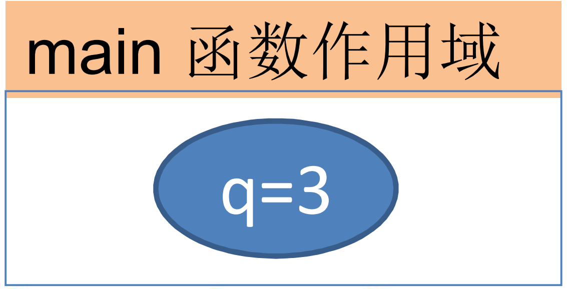
```
// pass-by-value
void increment(int a) {
  a = a + 1;
  cout << "a in increment " << a << endl;
}

int main() {
  int q = 3;
  increment(q); // does nothing
  cout << "q in main " << q << endl;
}
```

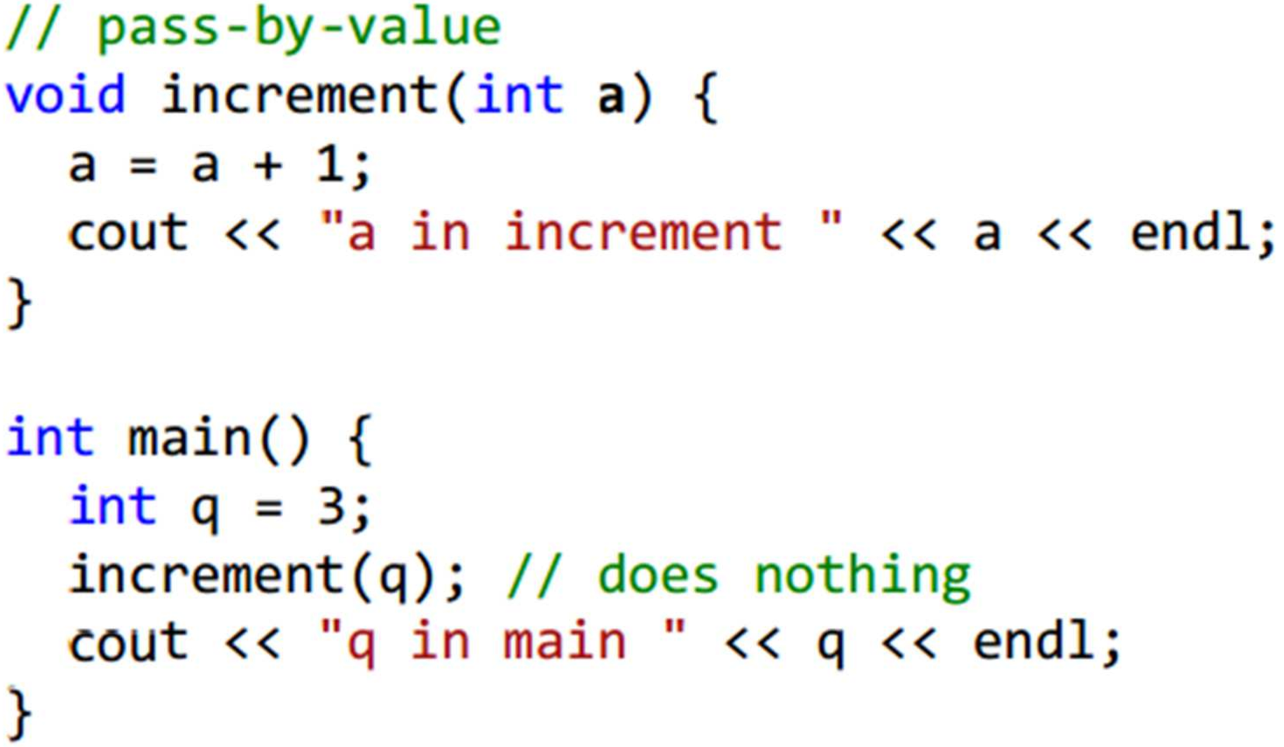
输出结果

a in increment 4

q in main **3**

传值(value)还是传引用(reference)
Pass by value vs by reference

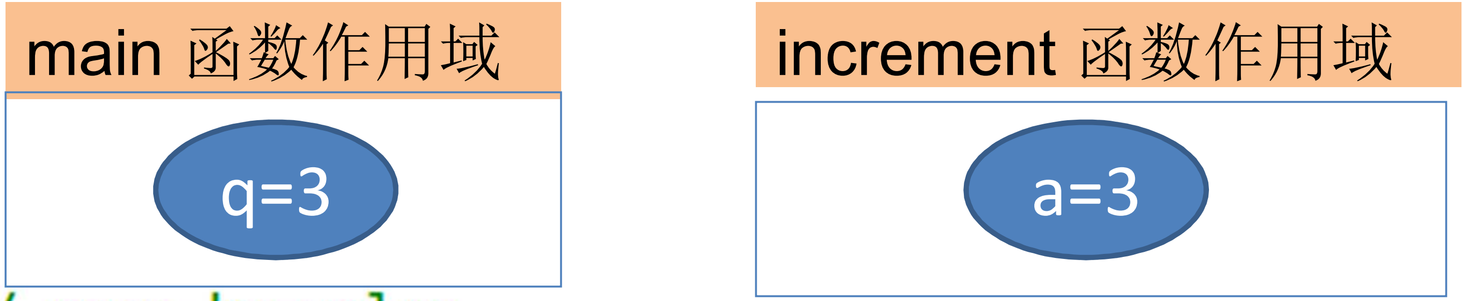


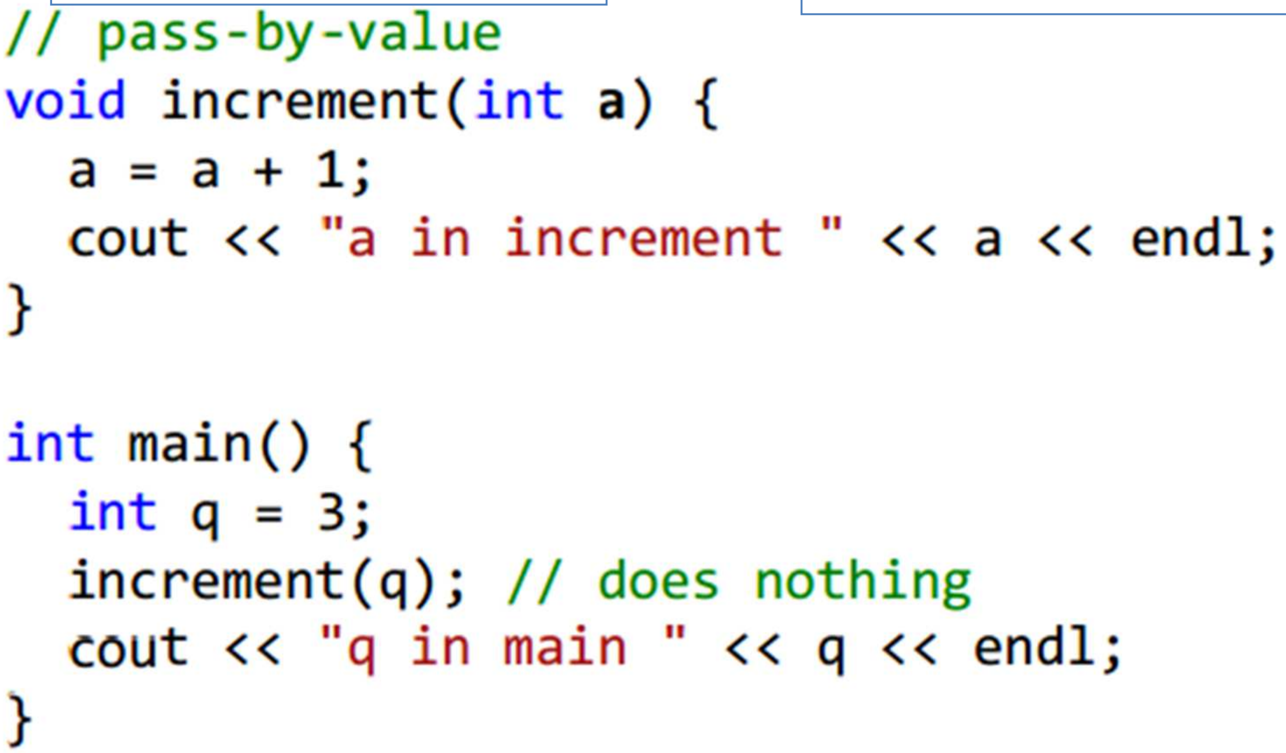

```
// pass-by-value
void increment(int a) {
  a = a + 1;
  cout << "a in increment " << a << endl;
}

int main() {
  int q = 3;
  increment(q); // does nothing
  cout << "q in main " << q << endl;
}
```

输出结果

a in increment 4
q in main **3**

传值(value)还是传引用(reference)
Pass by value vs by reference





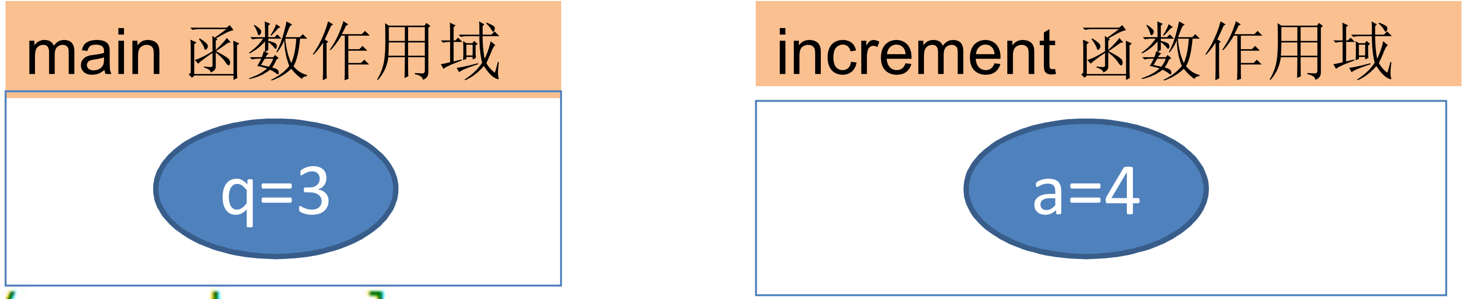
```
// pass-by-value
void increment(int a) {
  a = a + 1;
  cout << "a in increment " << a << endl;
}

int main() {
  int q = 3;
  increment(q); // does nothing
  cout << "q in main " << q << endl;
}
```

输出结果

a in increment 4

q in main **3**

传值(value)还是传引用(reference)
Pass by value vs by reference

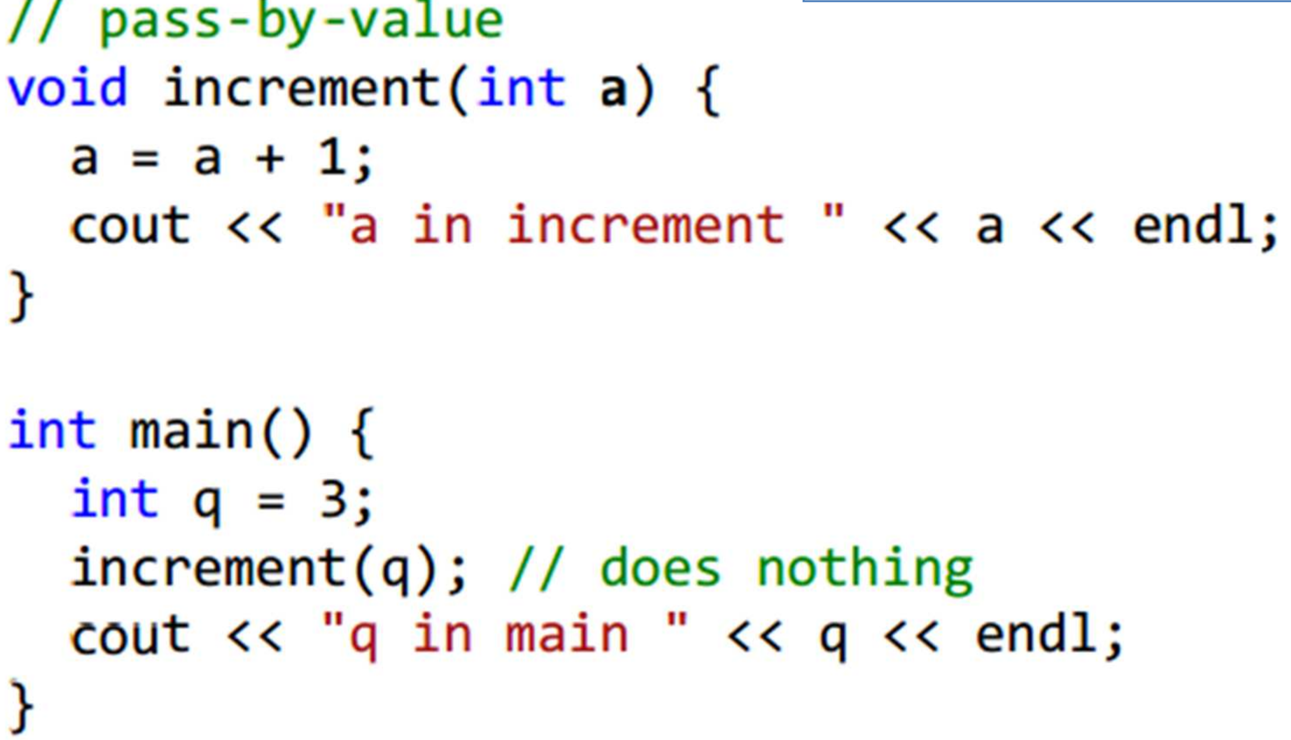


main 函数作用域

q=3

increment 函数作用域

a=4



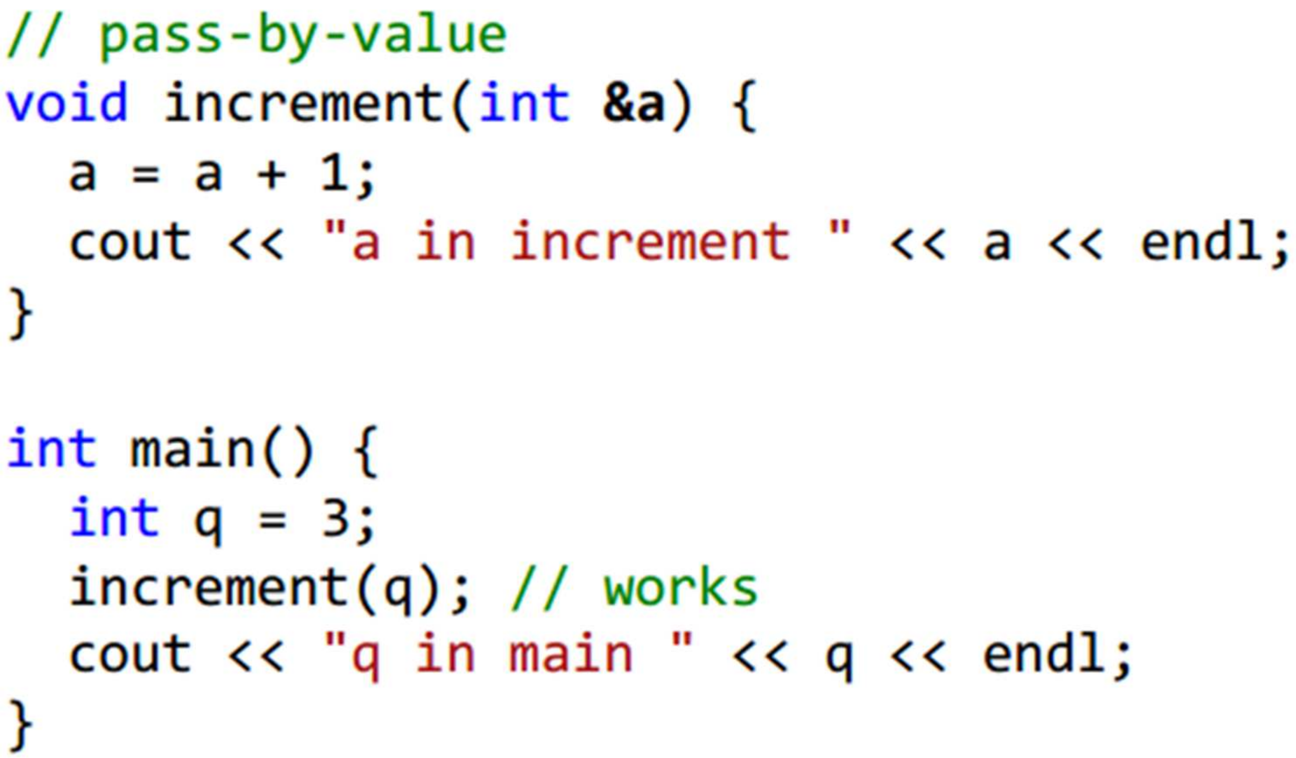
```
// pass-by-value
void increment(int a) {
  a = a + 1;
  cout << "a in increment " << a << endl;
}

int main() {
  int q = 3;
  increment(q); // does nothing
  cout << "q in main " << q << endl;
}
```

输出结果

```
a in increment 4
q in main 3
```

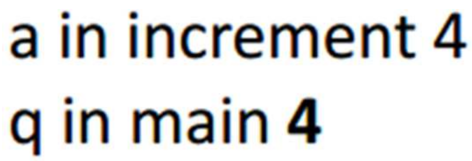
传值(value)还是传引用(reference)
Pass by value vs by reference

- 如果想修改原来的变量而不是复制原来的变量，可以传递引用(reference)。如用int &a 代替int a



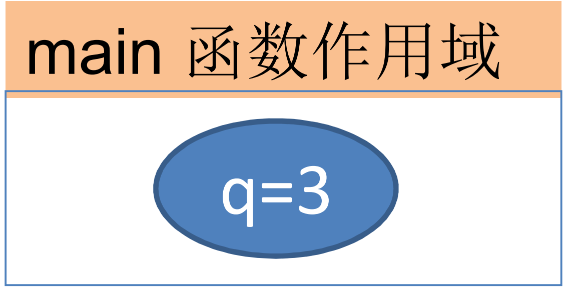
```
// pass-by-value
void increment(int &a) {
  a = a + 1;
  cout << "a in increment " << a << endl;
}

int main() {
  int q = 3;
  increment(q); // works
  cout << "q in main " << q << endl;
}
```

输出结果

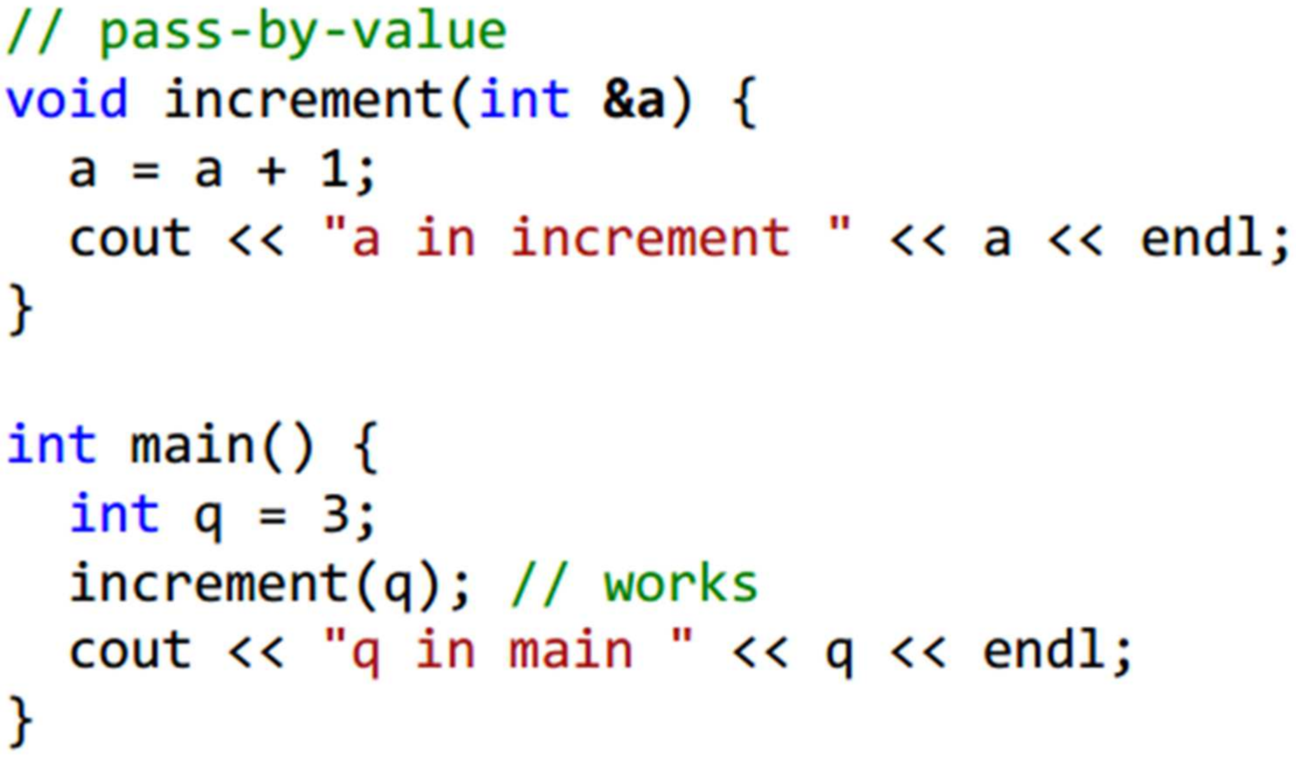


```
a in increment 4
q in main 4
```

传值(value)还是传引用(reference)
Pass by value vs by reference


main 函数作用域

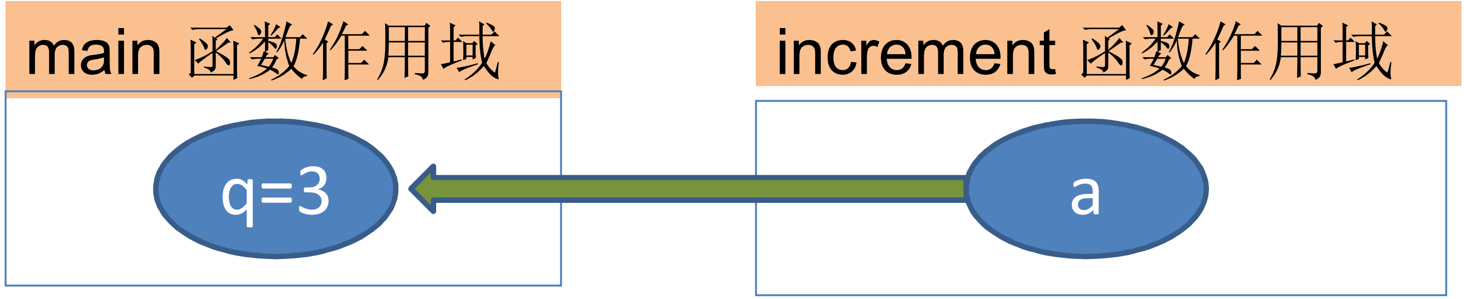
q=3


```
// pass-by-value
void increment(int &a) {
  a = a + 1;
  cout << "a in increment " << a << endl;
}

int main() {
  int q = 3;
  increment(q); // works
  cout << "q in main " << q << endl;
}
```

输出结果

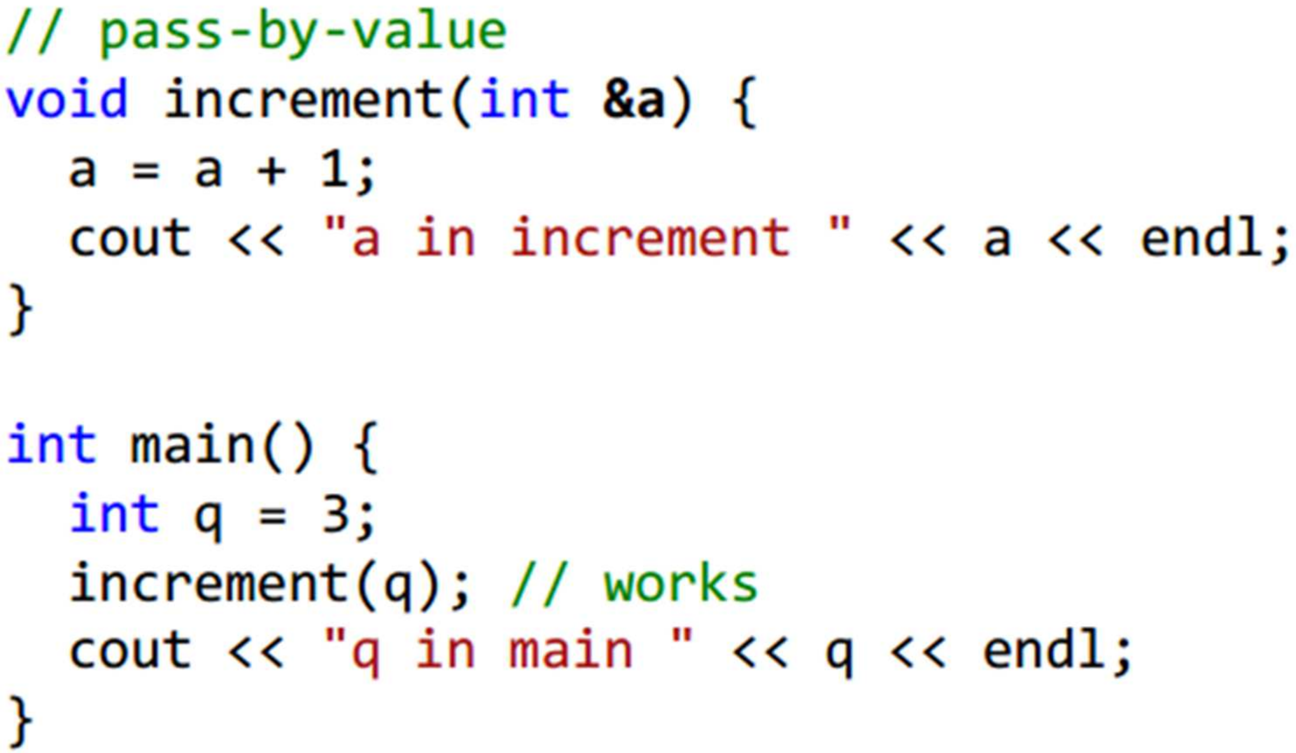
```
a in increment 4
q in main 4
```

传值(value)还是传引用(reference)
Pass by value vs by reference



main 函数作用域

increment 函数作用域



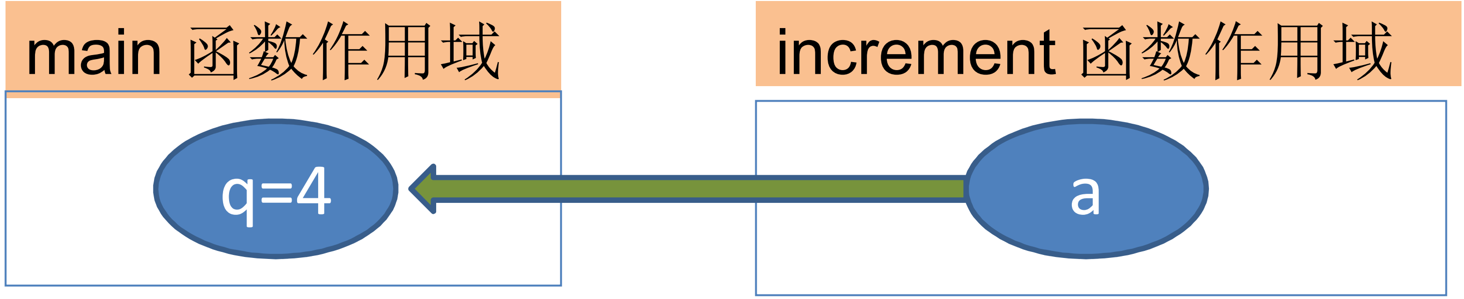
```
// pass-by-value
void increment(int &a) {
  a = a + 1;
  cout << "a in increment " << a << endl;
}

int main() {
  int q = 3;
  increment(q); // works
  cout << "q in main " << q << endl;
}
```

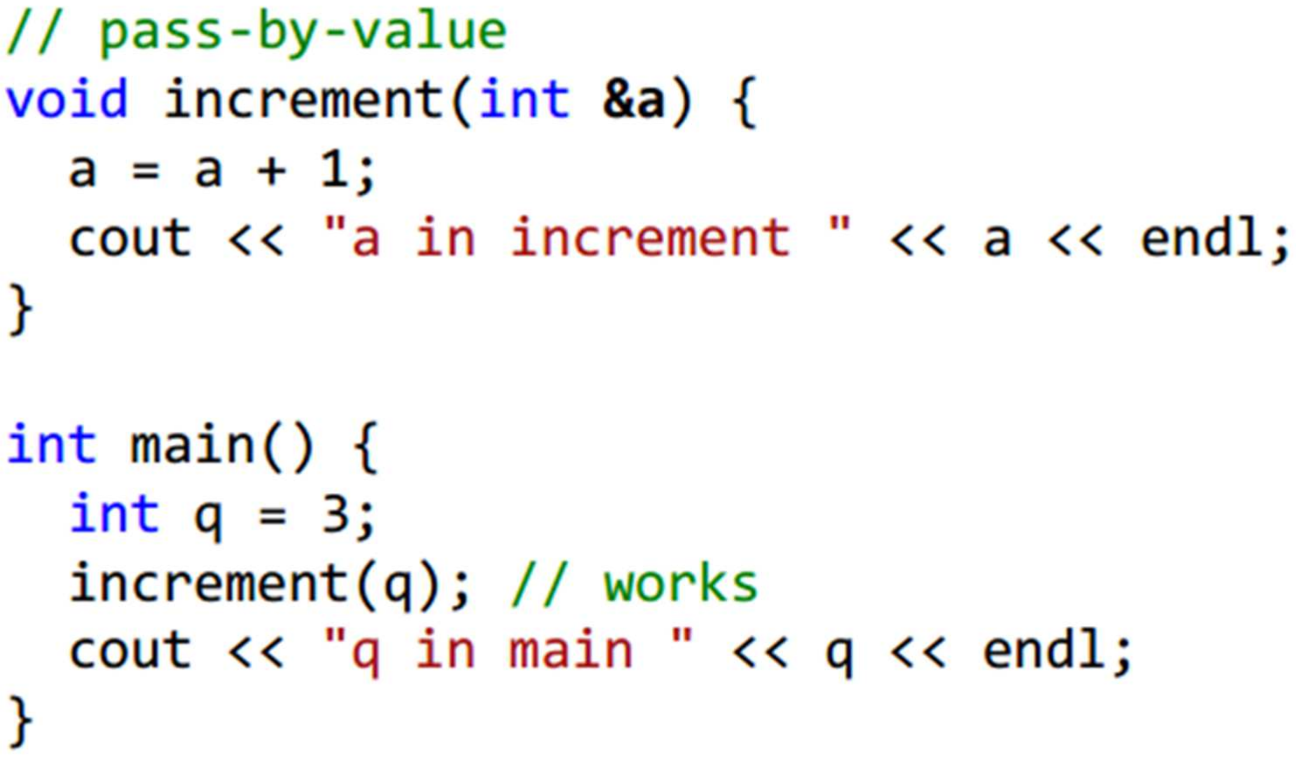
输出结果

```
a in increment 4
q in main 4
```

传值(value)还是传引用(reference)

Pass by value vs by reference



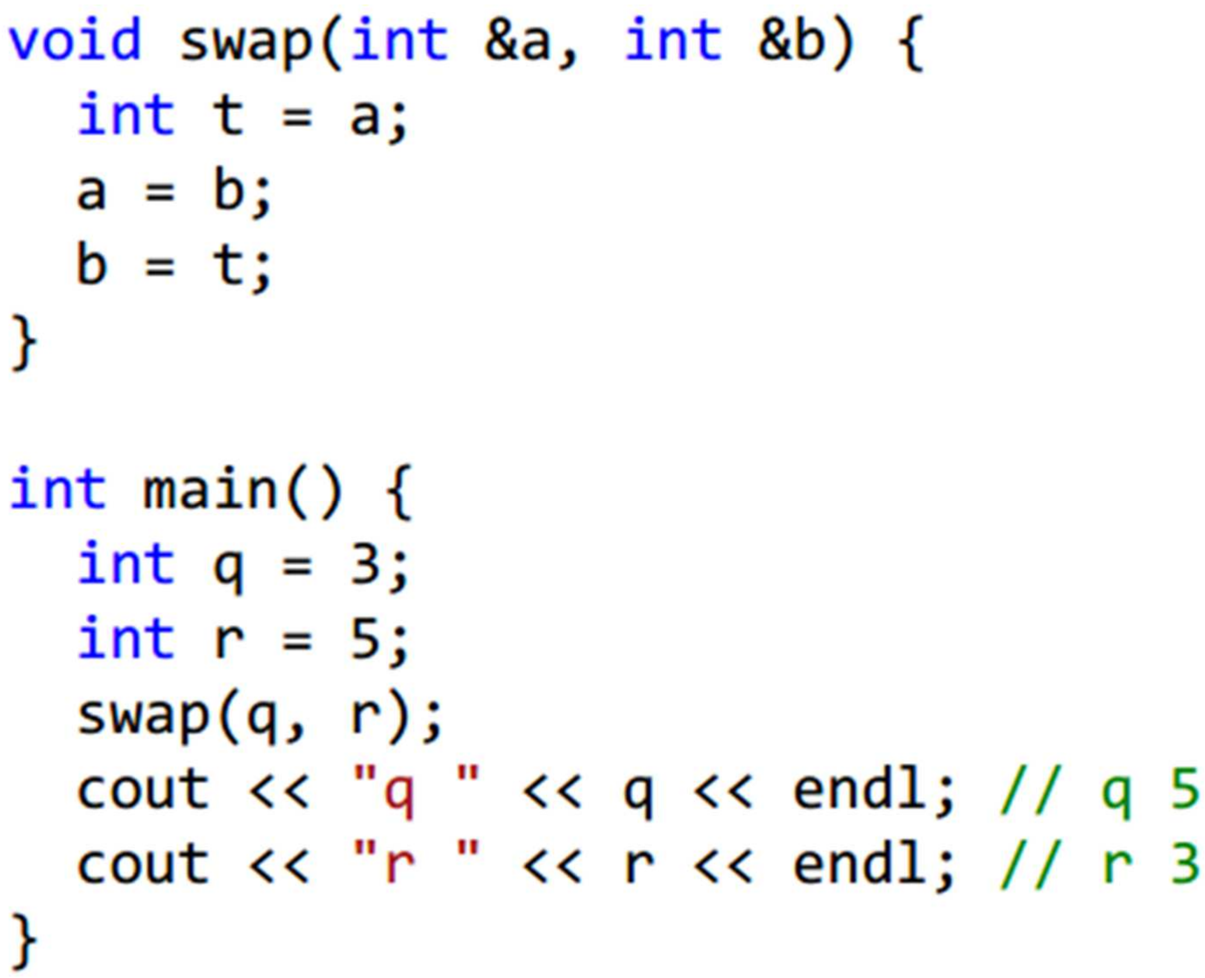


```
// pass-by-value
void increment(int &a) {
  a = a + 1;
  cout << "a in increment " << a << endl;
}

int main() {
  int q = 3;
  increment(q); // works
  cout << "q in main " << q << endl;
}
```

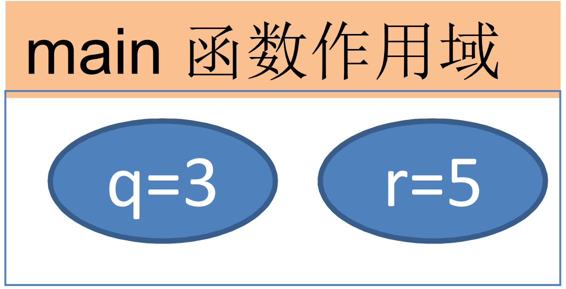
输出结果

```
a in increment 4
q in main 4
```

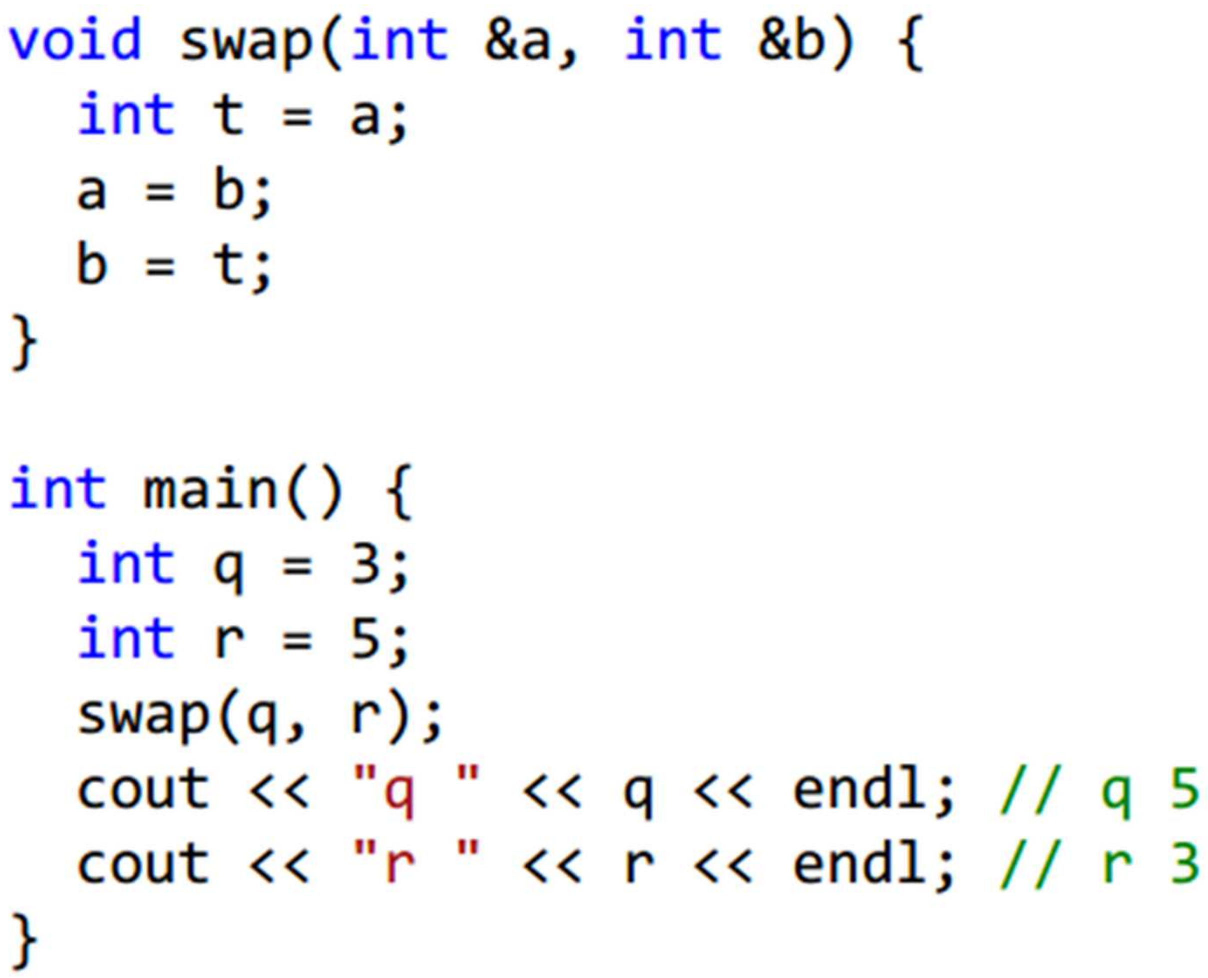
实现swap函数：交换两个变量



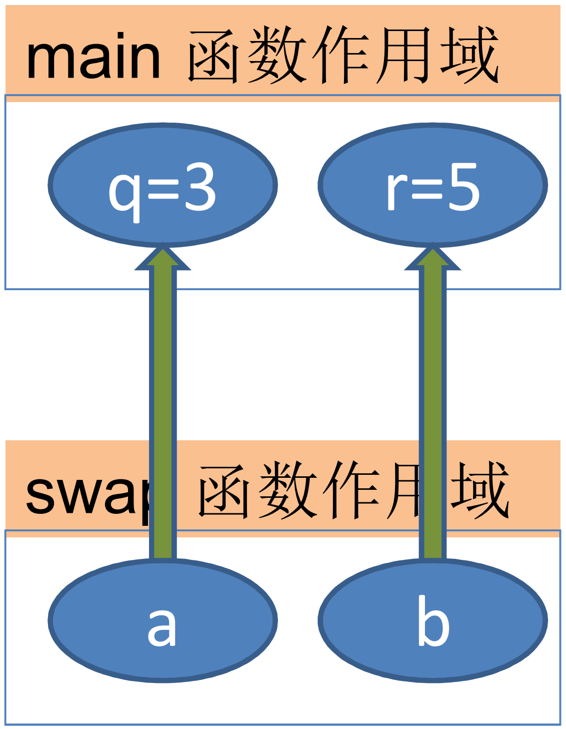
```cpp
void swap(int &a, int &b) {
  int t = a;
  a = b;
  b = t;
}

int main() {
  int q = 3;
  int r = 5;
  swap(q, r);
  cout << "q " << q << endl; // q 5
  cout << "r " << r << endl; // r 3
}
```



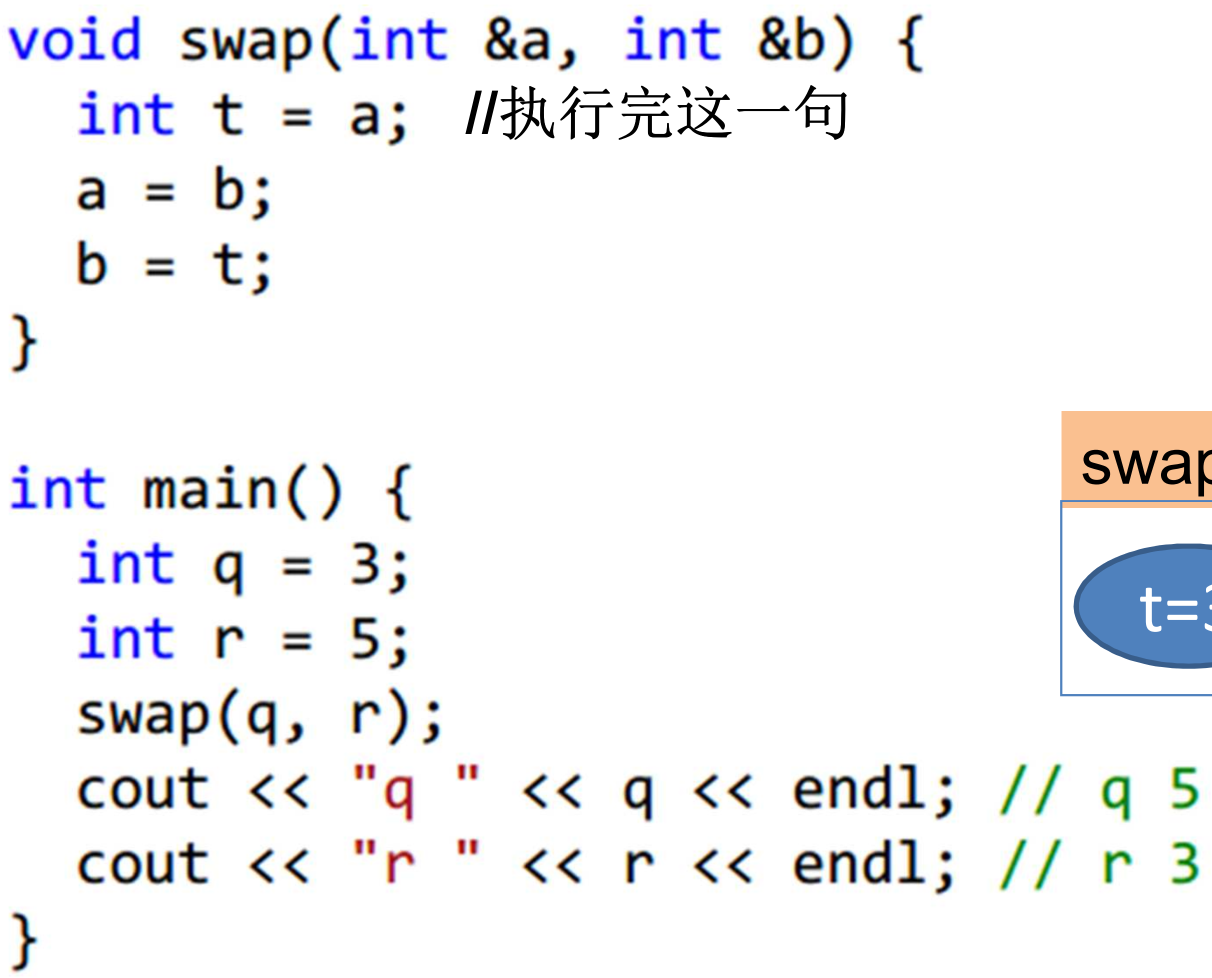
实现swap函数：交换两个变量



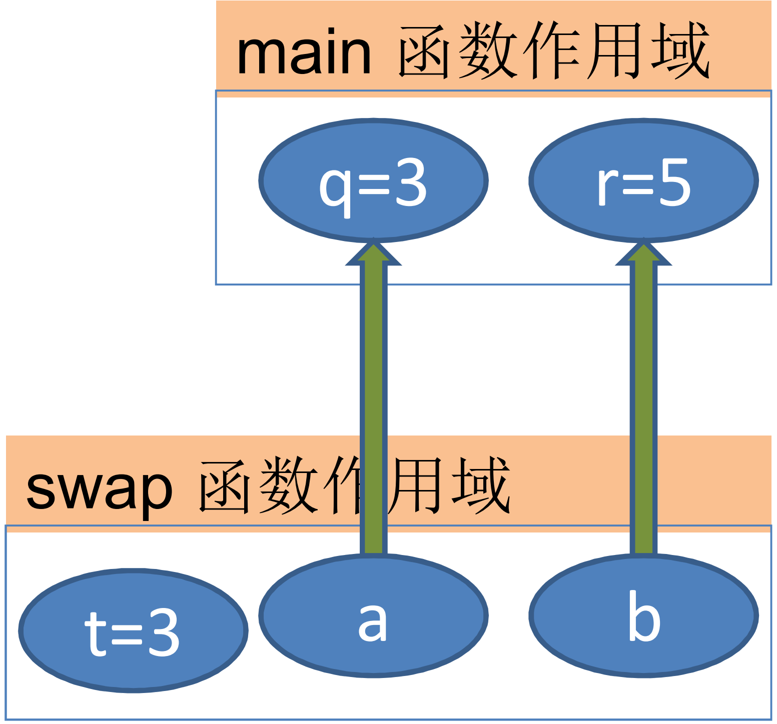
```
void swap(int &a, int &b) {
  int t = a;
  a = b;
  b = t;
}

int main() {
  int q = 3;
  int r = 5;
  swap(q, r);
  cout << "q " << q << endl; // q 5
  cout << "r " << r << endl; // r 3
}
```



实现swap函数：交换两个变量



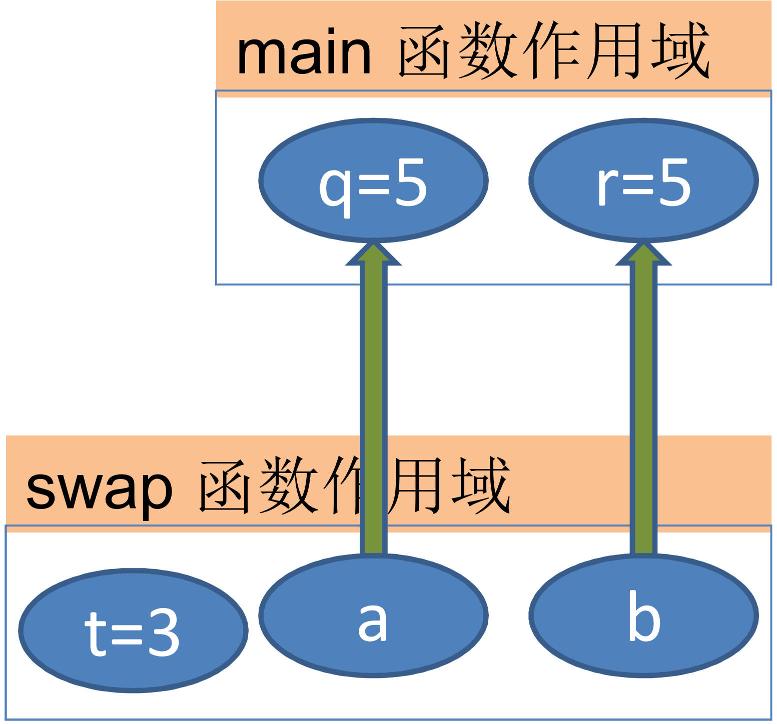
```
void swap(int &a, int &b) {
  int t = a;  //执行完这一句
  a = b;
  b = t;
}

int main() {
  int q = 3;
  int r = 5;
  swap(q, r);
  cout << "q " << q << endl; // q 5
  cout << "r " << r << endl; // r 3
}
```



实现swap函数：交换两个变量

```
void swap(int &a, int &b) {
  int t = a;
  a = b;
  b = t;
}

int main() {
  int q = 3;
  int r = 5;
  swap(q, r);
  cout << "q " << q << endl; // q 5
  cout << "r " << r << endl; // r 3
}
```

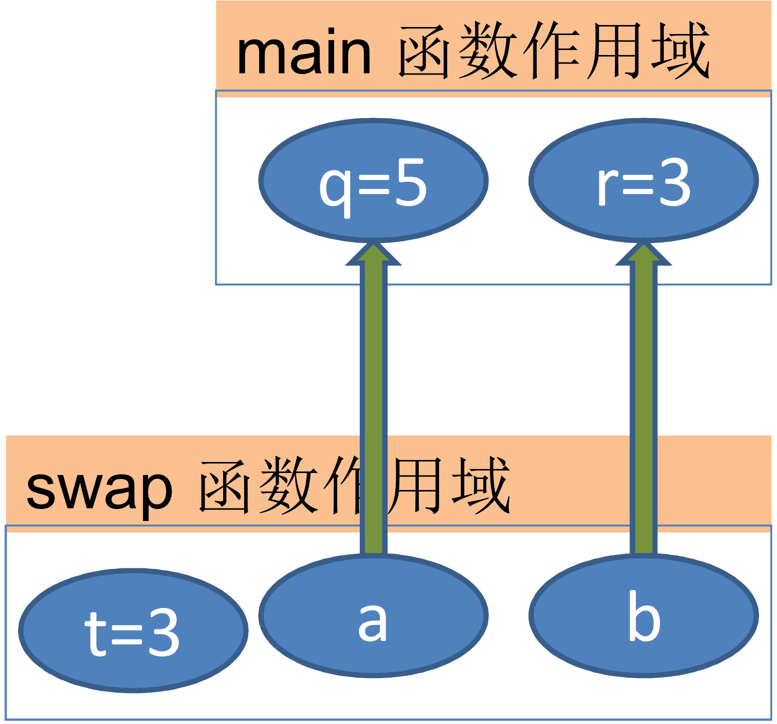
//执行完这一句



实现swap函数：交换两个变量

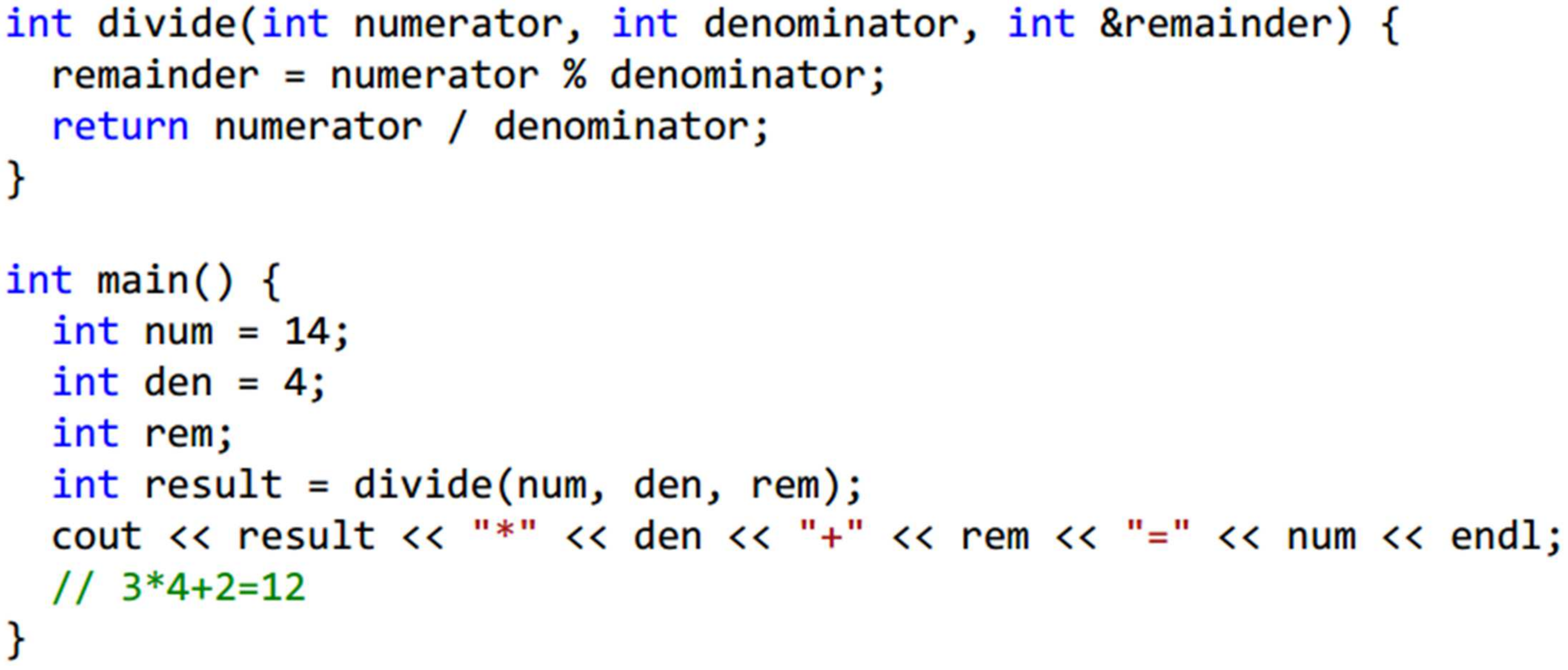
```
void swap(int &a, int &b) {
  int t = a;
  a = b;
  b = t;
}

int main() {
  int q = 3;
  int r = 5;
  swap(q, r);
  cout << "q " << q << endl; // q 5
  cout << "r " << r << endl; // r 3
}
```

//执行完这一句



返回多个值

- Returnn语句只能返回一个值，引用参数可用于输出结果。



```
int divide(int numerator, int denominator, int &remainder) {
  remainder = numerator % denominator;
  return numerator / denominator;
}

int main() {
  int num = 14;
  int den = 4;
  int rem;
  int result = divide(num, den, rem);
  cout << result << "*" << den << "+" << rem << "=" << num << endl;
  // 3*4+2=12
}
```

默认参数值Default values in parameters

- 默认参数一律靠右

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

int divide (int a, int b=2){
  int r = a/b;
  return (r);
}

int main (){
  cout << divide (12) << '\n';
  cout << divide (20,4) << '\n';
  return 0;
}
```

内联函数Inline Functions

- 指示编译器内联展开，可避免函数调用开销；
- 对于包含循环的或复杂代码，编译器仍作为普通函数。

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

inline string concatenate (const string& a,
           const string& b){
    return a+b;
}
int main (){
  string s= “Li”, w = “Wang”;
  string full = concatenate(s,w);
  return 0;
}
```

函数的Declaring声明与定义 defining

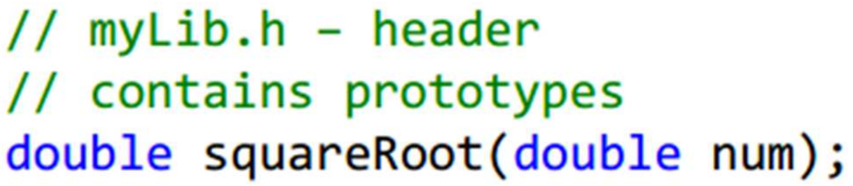
- 函数使用前须声明Declaring functions

```
void swap(int &a, int &b); //使用前先声明

int main(){
    int x = 3, y = 4;
    swap(x,y);
    std::cout<<x<<“ “<<y<<“\n”;
    return 0;
}
void swap(int &a, int &b) {
    int t = a; a = b;  b = t;
}
```

库(Librarys)

- 库的发布包含：1）头文件，其中有函数原型的说明；2）二进制的.dll 或.so文件包含了(编译过的)函数实现。

  -好处：你不需要共享你的.cpp源代码文件！

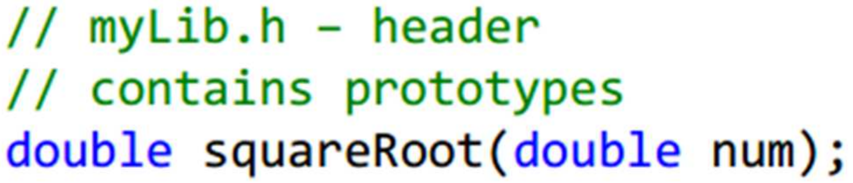


```
// myLib.h - header
// contains prototypes
double squareRoot(double num);
```

库(Librarys)

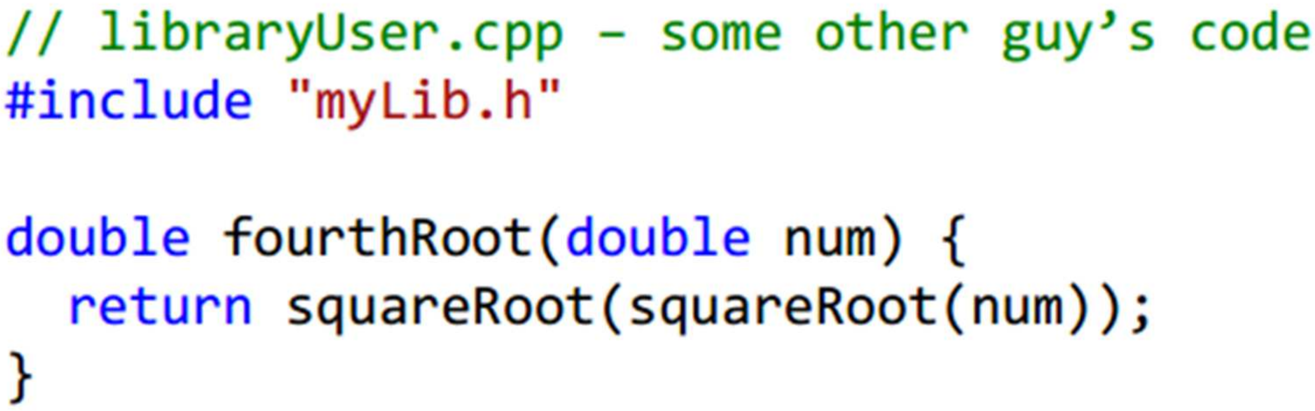
- 库的发布包含：1）头文件，其中有函数原型的说明；2）二进制的.dll 或.so文件包含了(编译过的)函数实现。

-好处：你不需要共享你的.cpp源代码文件！



```
// myLib.h - header
// contains prototypes
double squareRoot(double num);
```

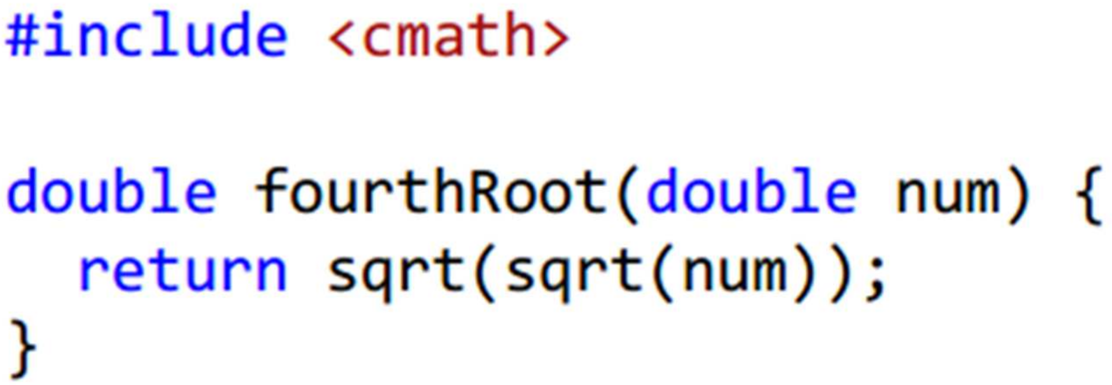
myLib.dll



```
// libraryUser.cpp - some other guy’s code
#include "myLib.h"

double fourthRoot(double num) {
  return squareRoot(squareRoot(num));
}
```

库(Librarys)- cmath

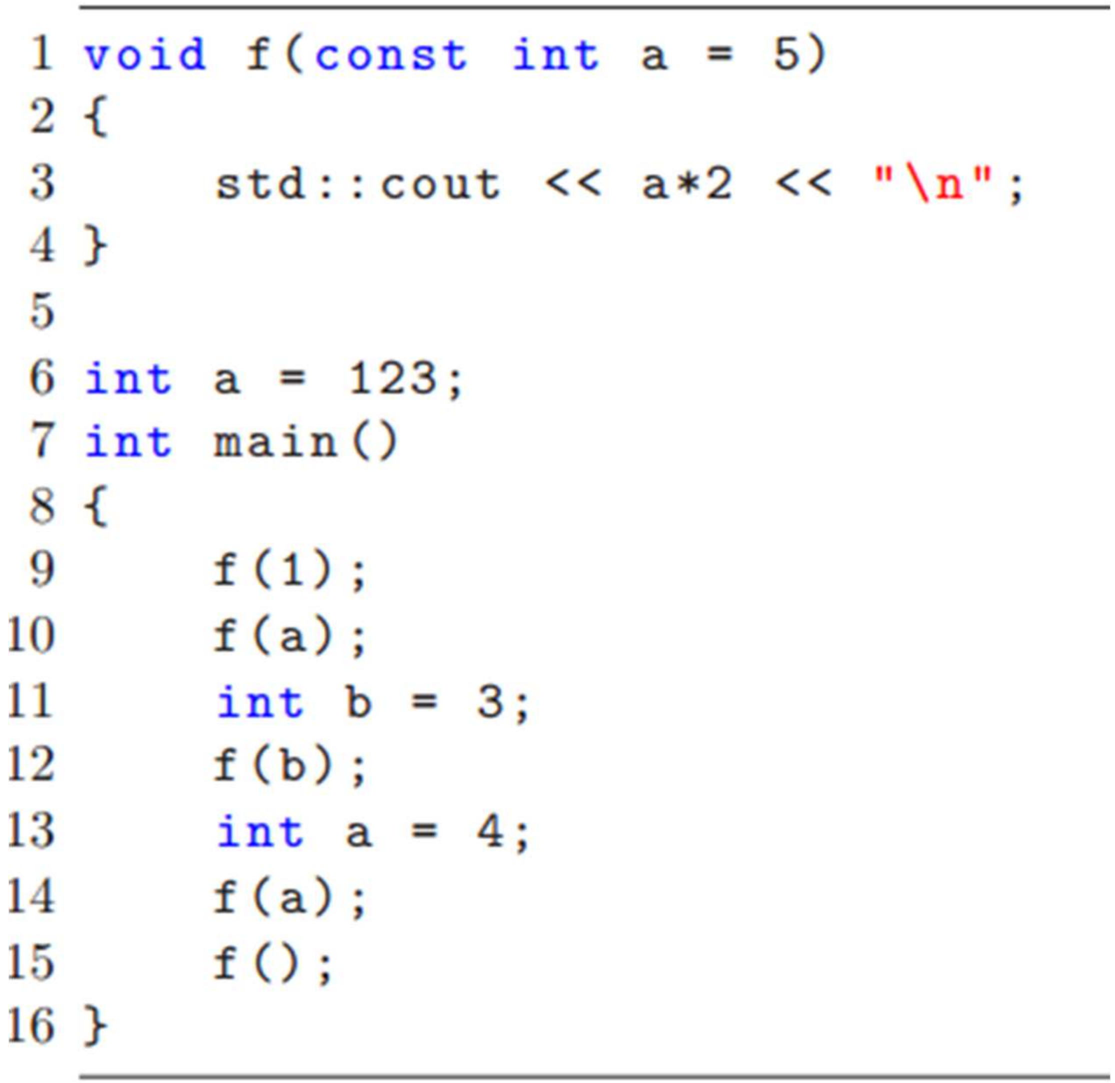
- 我们根本不需要自己实现之前的raiseToPower 和squareRoot函数，因为作为标准库的一部分cmath库已经实现了**pow** 和**sqrt**函数。我们只需要调用它们！



```
#include <cmath>

double fourthRoot(double num) {
  return sqrt(sqrt(num));
}
```

作业

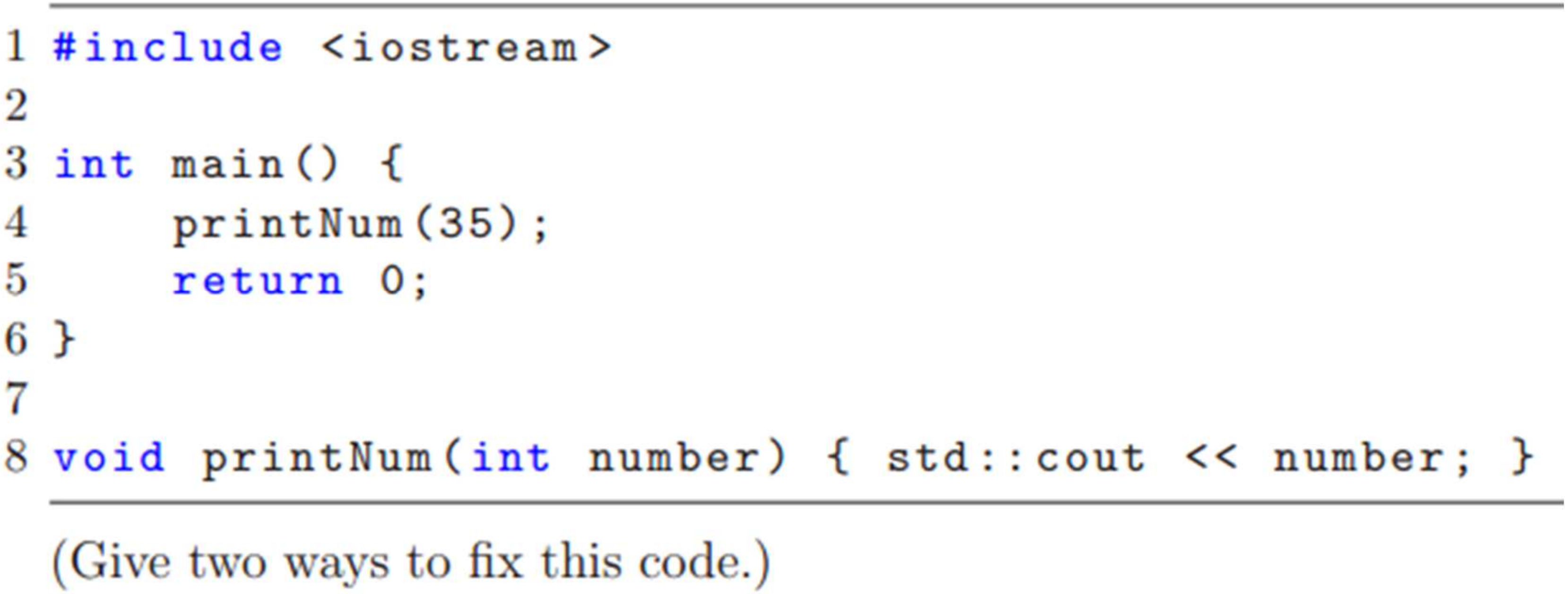
- 1. 不运行程序，说出程序输出结果



```
void f(const int a = 5)
{
    std::cout << a*2 << "\n";
}

int a = 123;
int main()
{
    f(1);
    f(a);
    int b = 3;
    f(b);
    int a = 4;
    f(a);
    f();
}
```

- 2. 指出下列程序中的错误

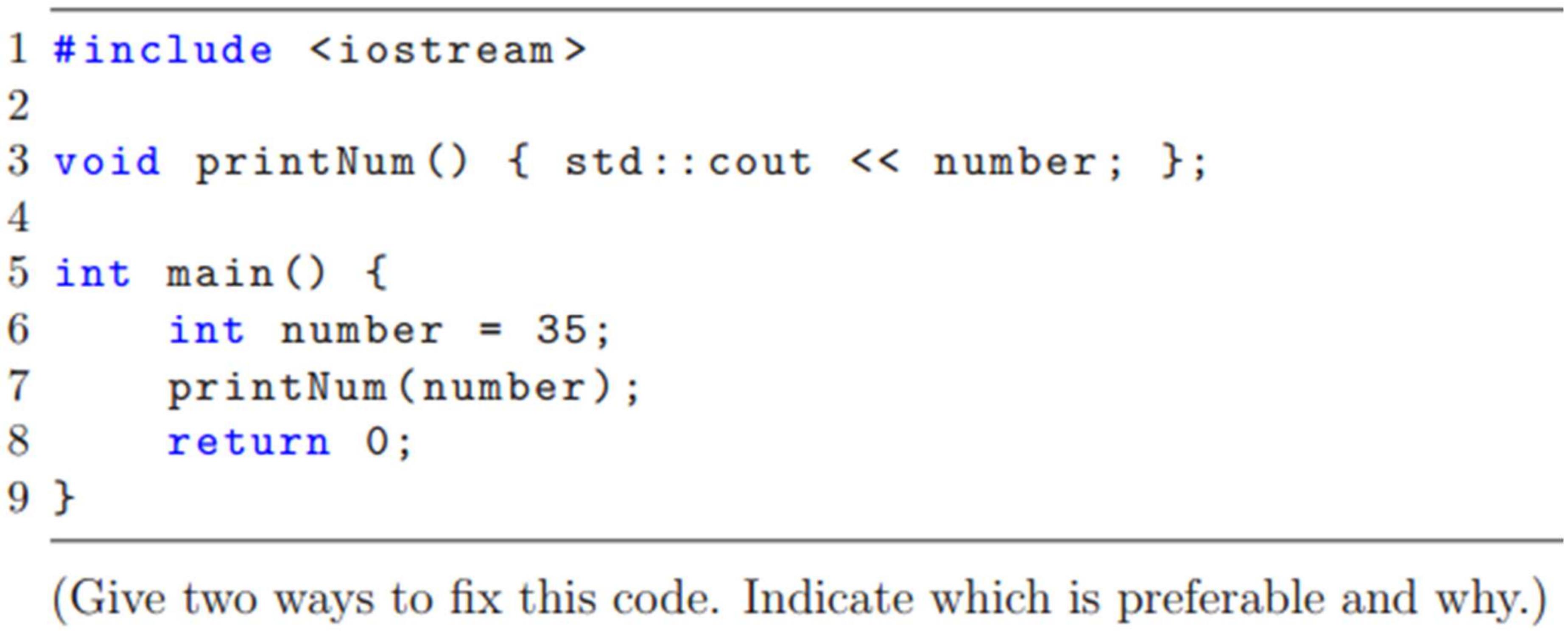
1）



```
#include <iostream>

int main() {
    printNum(35);
    return 0;
}

void printNum(int number) { std::cout << number; }
```

(Give two ways to fix this code.)

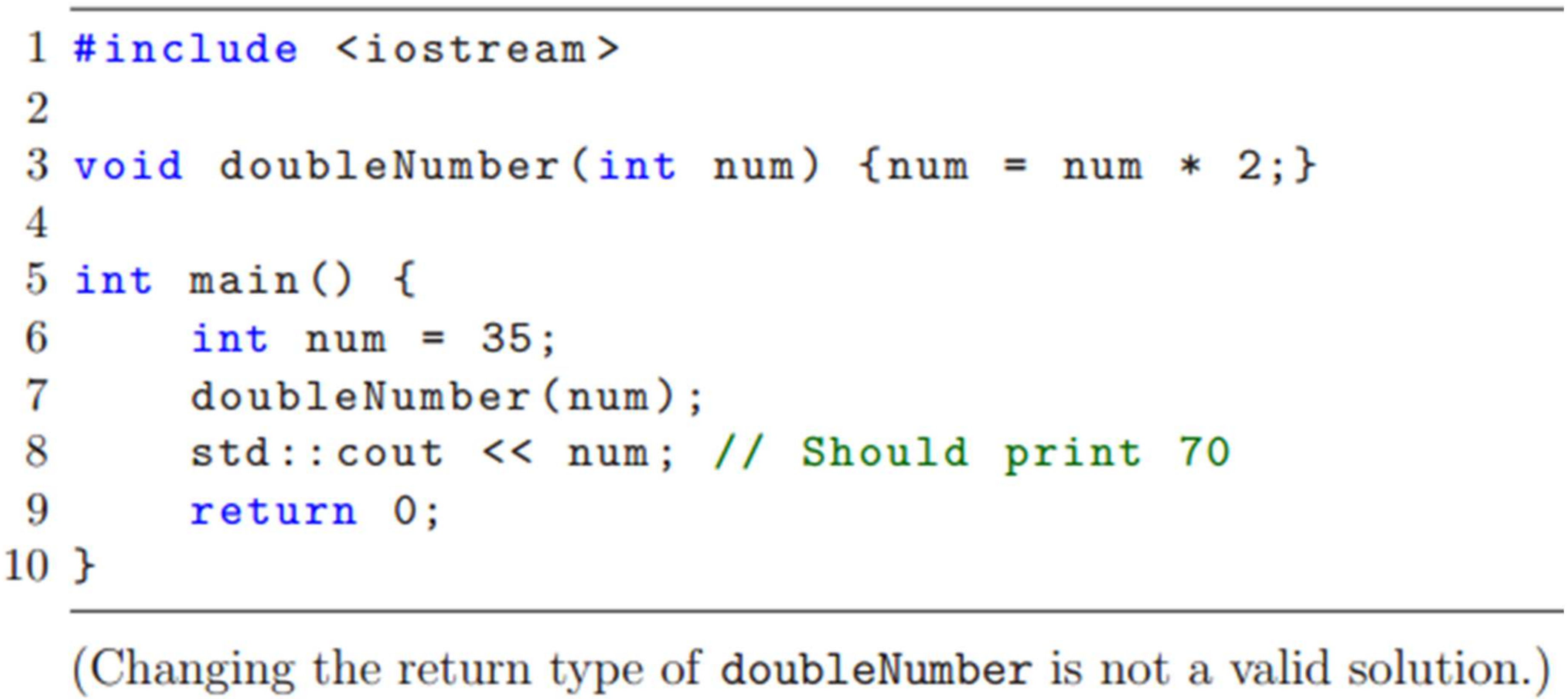
2）



```cpp
#include <iostream>

void printNum() { std::cout << number; };

int main() {
    int number = 35;
    printNum(number);
    return 0;
}
```

(Give two ways to fix this code. Indicate which is preferable and why.)

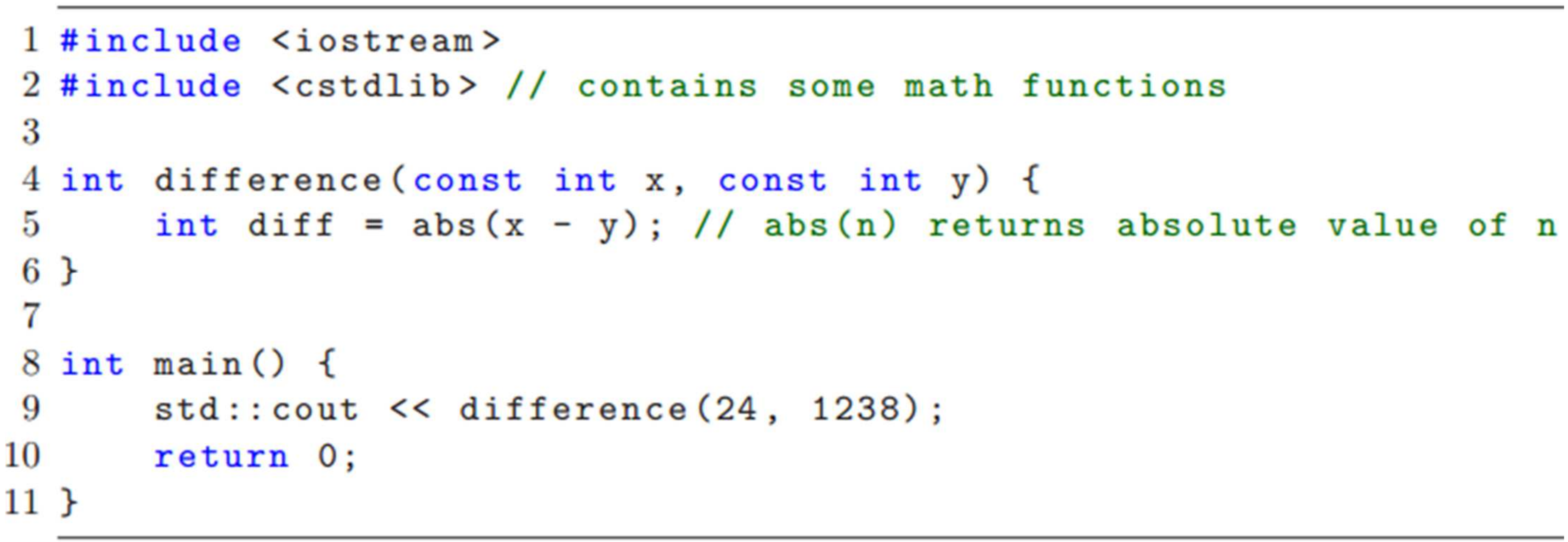
3）



```cpp
#include <iostream>

void doubleNumber(int num) {num = num * 2;}

int main() {
    int num = 35;
    doubleNumber(num);
    std::cout << num; // Should print 70
    return 0;
}
```

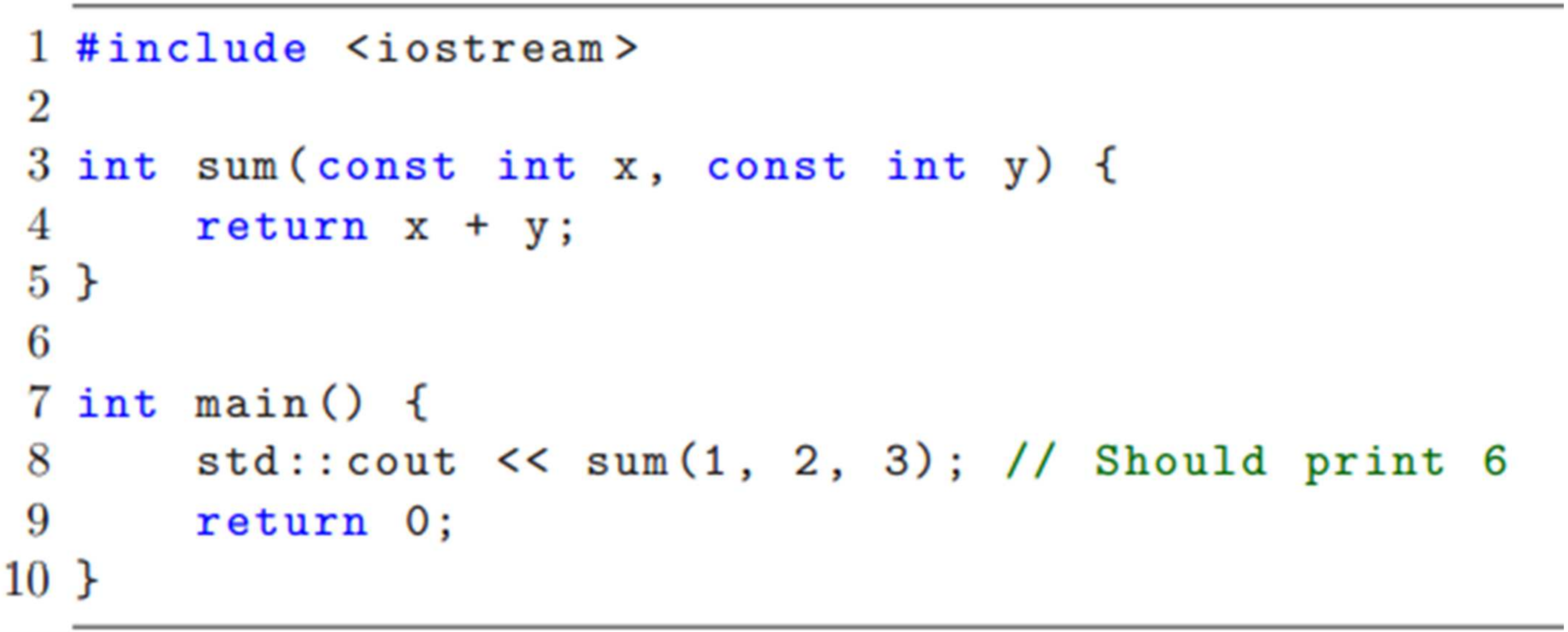
(Changing the return type of `doubleNumber` is not a valid solution.)

4）



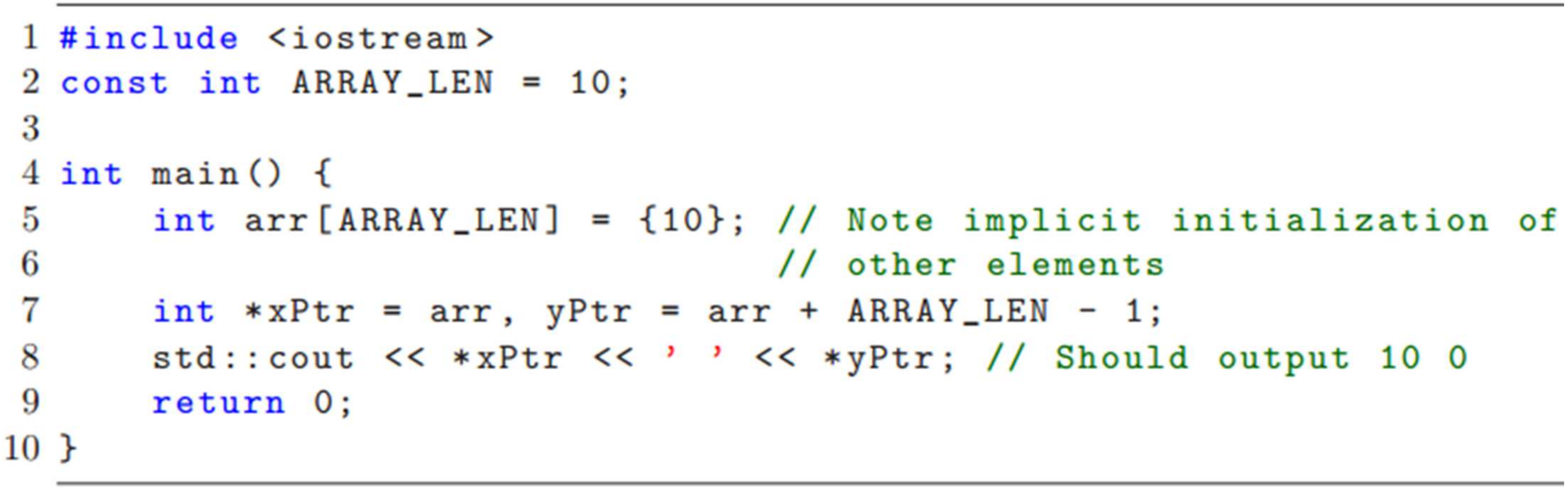
```cpp
#include <iostream>
#include <cstdlib> // contains some math functions

int difference(const int x, const int y) {
    int diff = abs(x - y); // abs(n) returns absolute value of n
}

int main() {
    std::cout << difference(24, 1238);
    return 0;
}
```

5）



```cpp
#include <iostream>

int sum(const int x, const int y) {
    return x + y;
}

int main() {
    std::cout << sum(1, 2, 3); // Should print 6
    return 0;
}
```

6）



```cpp
#include <iostream>
const int ARRAY_LEN = 10;

int main() {
    int arr[ARRAY_LEN] = {10}; // Note implicit initialization of
                               // other elements
    int *xPtr = arr, yPtr = arr + ARRAY_LEN - 1;
    std::cout << *xPtr << ' ' << *yPtr; // Should output 10 0
    return 0;
}
```

- 3. 编写一个函数计算一个已知半径的圆的面积。函数参数采用引用参数和返回值两种方式返回结果。格式如下：

```
void circleArea( double r, double& A);
double   circleArea(double r);
 int main(){
    double r = 23.5, A;
    //….
    std::cout<<“the area of the circle is :”<<A <<“\n”;
    return 0;
}
```