C++ & Python Program Design -- Basics

Functions

Junjie Cao @ DLUT

Summer 2022

https://github.com/jjcao-school/c

Variables

Variables变量 – Review 回顾

- An expression表达式:
  - 4 + 2

- We might want to **give a value a name** so we can **refer to it later**. We do this using variables.
- A variable is a **named location in memory**
  - x = 5; //a statement

- All computers have memory, called RAM (random access memory), that is available for programs to use.
- When a variable is defined, a piece of that memory is set aside for the variable.

Declaration声明

- An integer variable is a variable that holds an integer value.
- In order to define a variable, we generally use a **declaration statement**.
  - int x;

- We must tell the compiler what **type** x will be so that it knows how much **memory to reserve** for it and what kinds of **operations may** be **performed** on it.
- When this statement is executed by the CPU, a piece of memory from RAM will be set aside (called **instantiation实例化**).

- For the sake of example, let's say that the variable **x is assigned memory location 140**.
- Whenever the program sees the variable x in an expression or statement, it knows that it should look in memory location 140 to get the value.

Initialization初始化

- **initialization** of x, where we specify an initial value for it. This introduces **assignment operator**: =
  - x=5;
- When the CPU executes this statement, it translates this to “put the value of 5 in memory location 140”.

- Later in our program, we could print that value to the screen using std::cout:
  - std::cout << x; // prints the value of x (memory location 140) to the console

l-values and r-values 右值

- variables are a type of l-value
- An **l-value** is a value that has an **address** (in **memory**).
- The name l-value came about because l-values are the only values that **can be on the left side of an assignment statement**.
  - When we do an assignment, the left hand side of the assignment operator must be an l-value.
  - Consequently, a statement like 5 = 6; will cause a compile error, because 5 is not an l-value.
  - The value of 5 has no memory, and thus nothing can be assigned to it. 5 means 5, and its value can not be reassigned.

- When an l-value has a value assigned to it, the current value at that memory address is overwritten.

l-values and r-values

- An **r-value** refers to any value that can be assigned to an l-value.

- r-values are always evaluated to produce a single value.

- Examples of r-values are
  - single numbers (such as 5, which evaluates to 5),
  - variables (such as x, which evaluates to whatever value was last assigned to it),
  - expressions (such as 2 + x, which evaluates to the value of x plus 2).

l-values and r-values

```
1  int y;       // define y as an integer variable
2  y = 4;       // 4 evaluates to 4, which is then assigned to y
3  y = 2 + 5;   // 2 + 5 evaluates to 7, which is then assigned to y
4
5  int x;       // define x as an integer variable
6  x = y;       // y evaluates to 7 (from before), which is then assigned to x.
7  x = x;       // x evaluates to 7, which is then assigned to x (useless!)
8  x = x + 1;   // x + 1 evaluates to 8, which is then assigned to x.
```

- In last statement, x is being used in two different contexts.
  - On the left side of the assignment operator, “x” is being used as an l-value (variable with an address).
  - On the right side of the assignment operator, x is being used as an r-value, and will be evaluated to produce a value (in this case, 7).
  - When C++ evaluates the above statement, it evaluates as:
  - x = 7 + 1;
- Notes
  - **l-values must have address, r-values will be evaluated to produce a value**

l-values and r-values

- Practical application (best practice)
- Example
  - while (x=1) y++;
  - while (x==1) y++;

- if (1==x) //instead of x==1 When placing the r-value on the left, mistyping the == operator as the = operator triggers a compilation error.

- if (1=x) // error: "L-value required“ Although this isn't the most intelligible error message, it certainly catches the bug.

Initialization vs assignment

- new programmers often get mixed up
- After a variable is defined, a value may be **assigned** to it via the assignment operator (the = sign):

```
1 int x; // this is a variable definition
2 x = 5; // assign the value 5 to variable x
```

- C++ will let you both define a variable AND give it an initial value in the same step: **initialization**.

```
1 int x = 5; // initialize variable x with the value 5
```

- A variable can only be initialized when it is defined.

- we'll see cases in future lessons where some types of variables require an initialization value, or **disallow assignment**.
- For these reasons, it's useful to **make the distinction now**.

Uninitialized variables

- Unlike some programming languages, C/C++ does not initialize variables to a given value (such as zero) automatically (for performance reasons).
  - Thus when a variable is assigned to a memory location by the compiler, the default value of that variable is whatever **garbage** happens to already be in that memory location!

- A variable that has not been assigned a value is called an **uninitialized variable**.
  - Note: Some compilers, such as Visual Studio, *will* initialize the contents of memory when you're using a debug build configuration. This will not happen when using a release build configuration.

Uninitialized variables

- Uninitialized variables can lead to unexpected results:

```
1  // #include "stdafx.h" // Uncomment if Visual Studio user
2  #include <iostream>
3
4  int main()
5  {
6      // define an integer variable named x
7      int x;
8
9      // print the value of x to the screen (dangerous, because x is uninitialized)
10     std::cout << x;
11
12     return 0;
13 }
```

- In this case, the computer will assign some unused memory to x
- What value will it print? The answer is “who knows!”, and the answer may change every time you run the program.

Uninitialized variables

- Using uninitialized variables is one of the most common mistakes that novice programmers make,
  - Unfortunately, it can also be one of the most challenging to debug (because the program may run fine anyway if the uninitialized value happened to get assigned to a spot of memory that had a reasonable value in it, like 0).
  - Fortunately, most modern compilers will print warnings at compile-time.
    - vc2005projectstesttesttest.cpp(11) : warning C4700: uninitialized local variable 'x' used

- ***Rule: Initialize your variables.***
  - A good rule of thumb is to initialize your variables.
  - This ensures that your variable will always have a consistent value, making it easier to debug if something goes wrong somewhere else.

Quiz

```
1  int x = 5;
2  x = x - 2;
3  std::cout << x << std::endl; // #1
4
5  int y = x;
6  std::cout << y << std::endl; // #2
7
8  // x + y is an r-value in this context, so evaluate their values
9  std::cout << x + y << std::endl; // #3
10
11 std::cout << x << std::endl; // #4
12
13 int z;
14 std::cout << z << std::endl; // #5
```

#6 What is undefined behavior?

functions and return values

Function call

```
4   // Definition of function doPrint()
5   void doPrint() // doPrint() is the called function in this example
6   {
7       std::cout << "In doPrint()" << std::endl;
8   }
11  int main()
12  {
13      std::cout << "Starting main()" << std::endl;
14      doPrint(); // Interrupt main() by making a function call to doPrin
    t().  main() is the caller.
15      std::cout << "Ending main()" << std::endl;
```

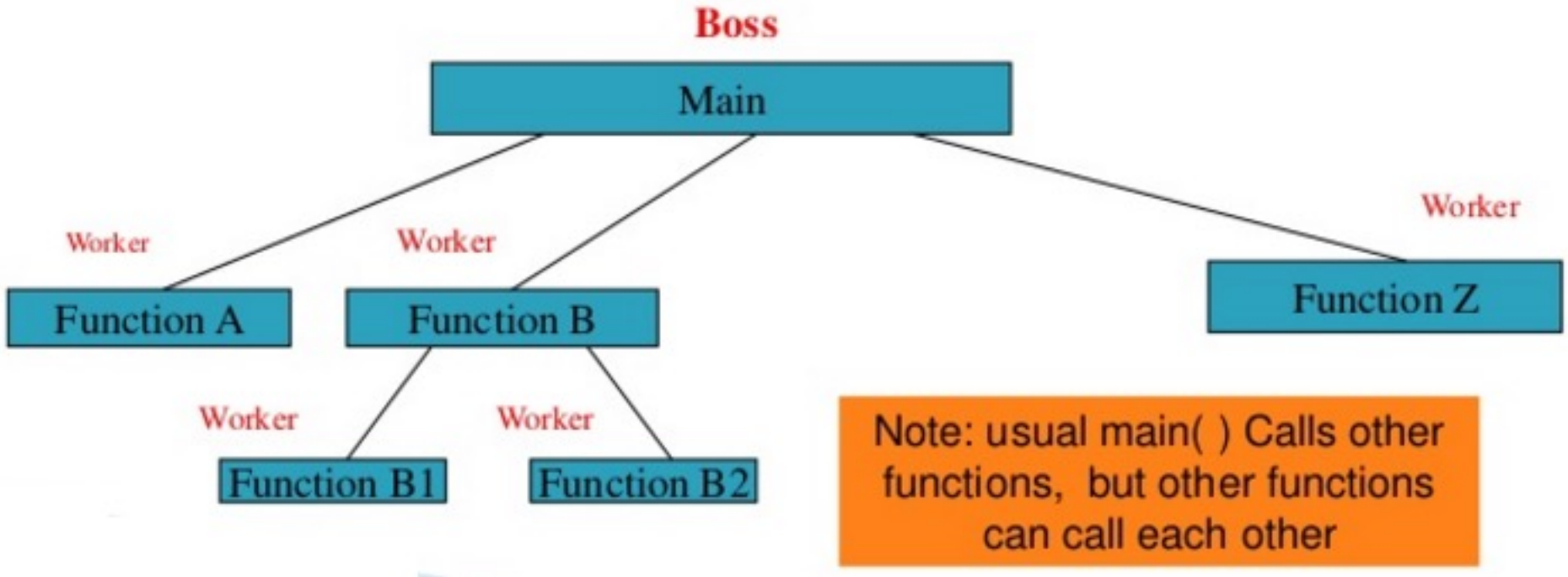
Starting main()
In doPrint()
Ending main()

- **main(): caller**
- **doPrint(): callee or called function.**

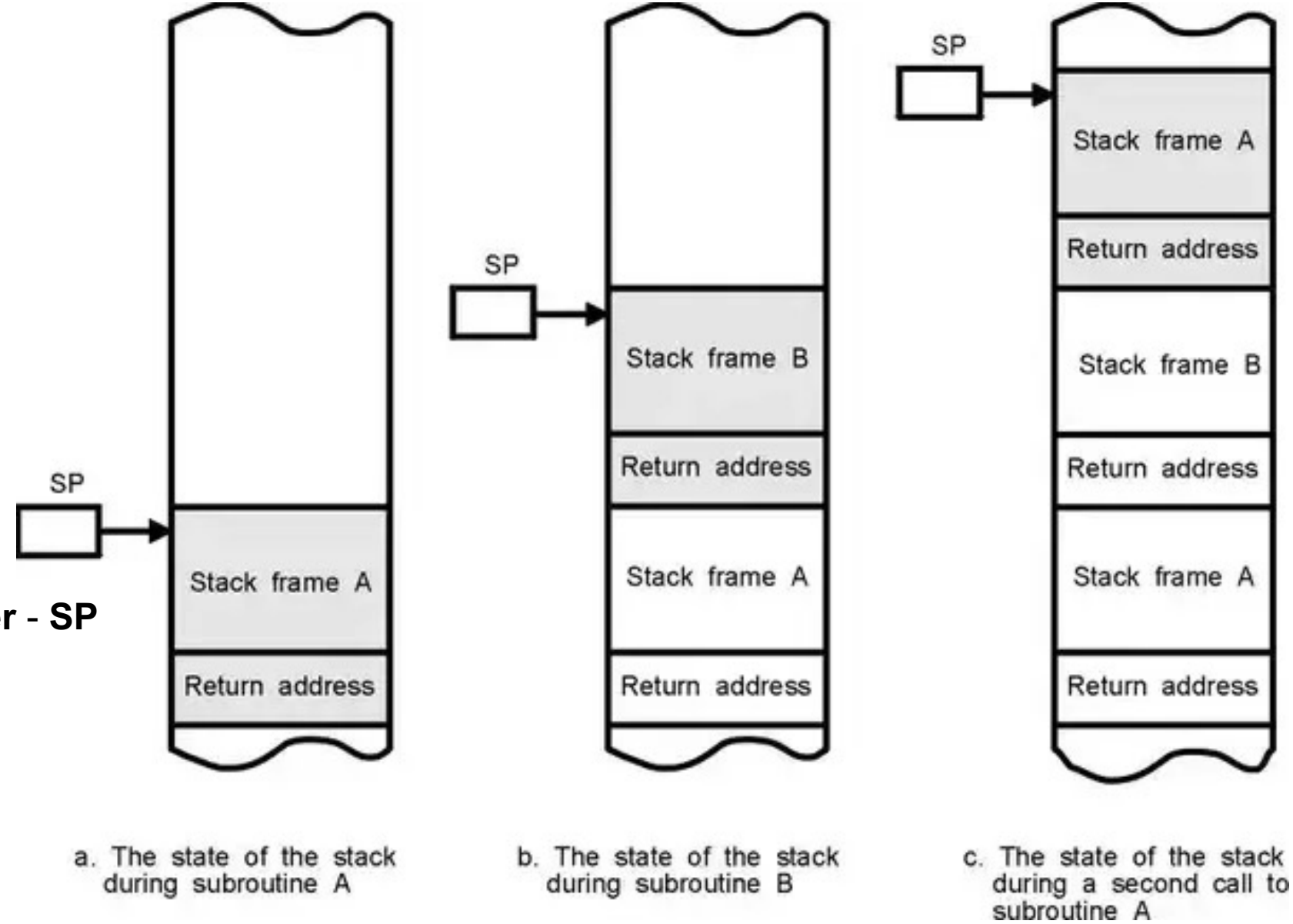
```
def doPrint():
    print("In doPrint()")

if __name__ == "__main__":
    print("Starting main()")
    doPrint()
    print("Ending main()")
```

Function

- A **function** is a **reusable** sequence of statements designed to do a particular job.
- A **function call** is an expression that tells the CPU to interrupt the current function and execute another function.



What happens when a function is called in C++?



stack pointer - SP

a. The state of the stack during subroutine A

b. The state of the stack during subroutine B

c. The state of the stack during a second call to subroutine A

In this image, function A is called, then A function calls B, then B function again calls A. In turn, this is a stack frame 堆栈帧 when both functions call each other and stack grows. (Image source : Google)

Reusing functions

- Any function can call another function!
- The same function can be called multiple times, which is useful if you need to do
- main() isn't the only function that can call other functions.

```
1  //#include <stdafx.h> // Visual Studio users need to uncomment this lin
   e
2  #include <iostream>
3
4  void printA()
5  {
6      std::cout << "A" << std::endl;
7  }
8
9  void printB()
10 {
11     std::cout << "B" << std::endl;
12 }
13
14 // function printAB() calls both printA() and printB()
15 void printAB()
16 {
17     printA();
18     printB();
19 }
20
21 // Definition of main()
22 int main()
23 {
24     std::cout << "Starting main()" << std::endl;
25     printAB();
26     std::cout << "Ending main()" << std::endl;
27     return 0;
28 }
```

Nested functions

- Functions can not be defined inside other functions (called nesting) in C++. While Python not.

```cpp
#include <iostream>

int main()
{
    int foo() // this function is nested inside main(), which is illega
l.
    {
        std::cout << "foo!";
        return 0;
    }

    foo();
    return 0;
}
```

```python
def ret1():# ok
    def nested_fun():
        return 2, 3
    nested_fun()
    return 2
```

```cpp
#include <iostream>

int foo() // no longer inside of main()
{
    std::cout << "foo!";
    return 0;
}

int main()
{
    foo();
    return 0;
}
```

Return values of Python functions

```
def ret1():
    return 2
def ret2():
    return 2, ‘ok’
if __name__ == "__main__":
    print(type(ret1()))
    print(type(ret2()))
    print(ret2())
```

<class 'int'>
<class 'tuple’> 元组
(2, 'ok')

Return values

- A return type of **void** means the function does not return a value.
- A return type of **int** means the function returns a **integer** value to the caller.

```
1  // void means the function does not return a value to the caller
2  void returnNothing()
3  {
4      // This function does not return a value so no return statement is
   needed
5  }
6
7  // int means the function returns an integer value to the caller
8  int return5()
9  {
10     return 5; // this function returns an integer, so a return statemen
   t is needed
11 }
```

```
15  int main()
16  {
17      std::cout << return5() << std::endl; // prints 5
18      std::cout << return5() + 2 << std::endl; // prints 7
19
20      returnNothing(); // okay: function returnNothing() is called, no va
    lue is returned
21      return5(); // okay: function return5() is called, return value is d
    iscarded
22
23      std::cout << returnNothing(); // This line will not compile.  You'l
    l need to comment it out to continue.
24
25      return 0;
26  }
```

- “Can my function return multiple values using a return statement?”.
  - No.
  - Functions can only return a **single** value using a return statement.
  - However, there are ways to work around the issue, ….

Returning of main

- When the program is executed, the operating system makes a function call to main().
- Execution then jumps to the top of main.
- The statements in main are executed sequentially.
- Finally, main returns a integer value (usually 0) back to the operating system.

Returning to main

- Why return a value back to the operating system?
  - This value is called a **status code**, and it tells the operating system (and any other programs that called yours) whether your program executed successfully or not.
  - By consensus, a return value of 0 means success, and a positive return value means failure.

Function parameters 形参 & arguments 实参

Function parameters 形参

- A parameter is the variable which is part of the function's signature签名 (function declaration声明/definition定义).

```
15  // This function has two integer parameters, one named x
    y
16  // The caller will supply the value of both x and y
17  int add(int x, int y)
18  {
19      return x + y;
20  }
```

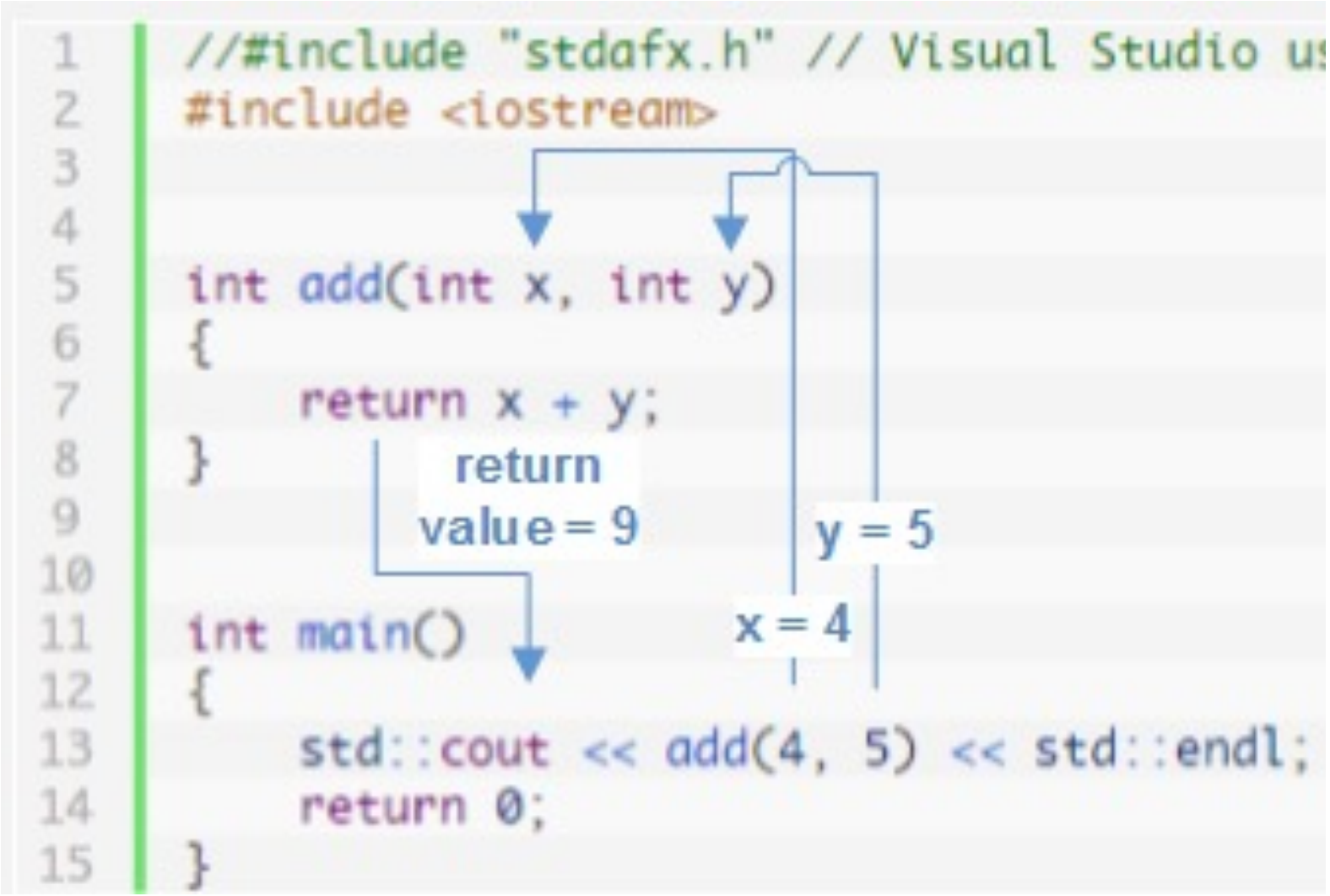
Function argument 实参

- An **argument** is a value that is passed *from* the caller *to* the function when a function call is made:

```
1  printValue(6); // 6 is the argument passed to function printValue()
2  add(2, 3); // 2 and 3 are the arguments passed to function add()
```

How parameters, arguments & return values work together

- When a function is called, all of the **parameters** of the function are created as **variables**
- the value of each of the **arguments** is *copied* into the matching parameter. This process is called **pass by value**.
- By using both parameters and a return value, we can create functions that take data as input, do some calculation with it, and return the value to the caller.



How parameters and return values work together

```
#include <iostream>

int add(int x, int y)
{
    return x + y;
}

int multiply(int z, int w)
{
    return z * w;
}

int main()
{
    using namespace std;
    cout << add(4, 5) << endl; // within add(), x=4, y=5, so x+y=9
    cout << multiply(2, 3) << endl; // within multiply(), z=2, w=3, so z*w=6

    // We can pass the value of expressions
    cout << add(1 + 2, 3 * 4) << endl; // within add(), x=3, y=12, so x+y=15

    // We can pass the value of variables
    int a = 5;
    cout << add(a, a) << endl; // evaluates (5 + 5)

    cout << add(1, multiply(2, 3)) << endl; // evaluates 1 + (2 * 3)
    cout << add(1, add(2, 3)) << endl; // evaluates 1 + (2 + 3)

    return 0;
}
```

Pass by reference引用 vs value值

```
void swapv( int a, int b){
    int tmp;
    tmp = a; a = b; b = tmp;}
void swapr( int & a, int & b){
    int tmp;
    tmp = a; a = b; b = tmp;}

int n1(1), n2(2);
swapv(n1,n2);
cout << n1 << ", " << n2 << endl;
swapr(n1,n2);
cout << n1 << ", " << n2 << endl;
```

- All parameters (arguments) in the Python language are passed by object reference.

Pass mutable or immutable objects

```
def swap2(a, b):
    tmp = a
    a = b
    b = tmp
def swap1(a):
    tmp = a[0]
    a[0] = a[1]
    a[1] = tmp
```

```
n1 =1; n2=2
swap2(n1,n2)
print("{}, {}".format(n1, n2))
li1 = [n1,n2]
swap1(li1)
print(li1)
```

1, 2
[2, 1]

- if object is **immutable(not modifiable)** than the modified value is not available outside the function.
  - *int, float, complex, string, tuple, frozen set [note: immutable version of set], bytes.*
- if object is **mutable (modifiable)** than modified value is available outside the function.
  - *list, dict, set, byte array*

Function Arguments

- Required arguments
- Default arguments
- Keyword arguments
- Variable-length arguments

```
def printinfo( name, age=50 ):
    print("name: {}; age: {}".format(name, age))

printinfo( age=50, name="miki" )
printinfo( "miki", age=50 )
#printinfo( age=50, "miki" )
```

Variable-length arguments

```
def printinfo_varl( arg1, *vartuple ):
    #"This prints a variable passed arguments"
    print("Output is: ")
    print(arg1)
    for var in vartuple:
        print(var)

printinfo( 10 )
printinfo( 70, 60, 50 )
```

Anonymous Functions in Python

- syntax of *lambda* functions
  - lambda [arg1 [,arg2,.....argn]]:expression

```
sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

Conclusion

- **Parameters** are the key mechanism by which functions can be written in a **reusable** way,
  - as it allows them to perform tasks without knowing the specific input values ahead of time.
  - Those **input values** are passed in as **arguments** by the caller.

- **Return values** allow a function to return a value back to the caller.

Quiz

1) What's wrong with this program fragment?

```
1  void multiply(int x, int y)
2  {
3      return x * y;
4  }
5
6  int main()
7  {
8      std::cout << multiply(4, 5) << std::endl;
9      return 0;
10 }
```

Quiz

2) What two things are wrong with this program fragment?

```
1  int multiply(int x, int y)
2  {
3      int product = x * y;
4  }
5
6  int main()
7  {
8      std::cout << multiply(4) << std::endl;
9      return 0;
10 }
```

Quiz

3) What value does the following program print?

```
1  #include <iostream>
2
3  int add(int x, int y, int z)
4  {
5      return x + y + z;
6  }
7
8  int multiply(int x, int y)
9  {
10     return x * y;
11 }
12
13 int main()
14 {
15     std::cout << multiply(add(1, 2, 3), 4) << std::endl;
16     return 0;
17 }
```

Quiz

4) Write a function called doubleNumber() that takes one integer parameter and returns twice the value passed in.

5) Write a complete program that reads an integer from the user (using cin, doubles it using the doubleNumber() function, and then prints the doubled value out to the console.

Scope范围 of Variables

All variables in a program may not be accessible at all locations in that program.

- Depends on where you have declared a variable.
  - Global variables全局变量
  - Local variables局部变量

```
total = 0; # This is global variable.
def sum( arg1, arg2 ): # Add both the parameters and return them."
    total = arg1 + arg2 # Here total is local variable.
    print("Inside the function local total : {}”.format(total))
    return total
# Now you can call sum function
sum( 10, 20 )
print("Outside the function global total : {}”.format(total))
```

When is an instantiated实例化的 variable destroyed?

- A variable's **scope** determines who can **see** & **use** the variable

- function parameters & variables declared inside function body have **local scope**.
  - That is, those variables can only be seen and used within the function that declares them.
  - Local variables are **created** at the point of definition, and **destroyed** when they go out of scope.

When is an instantiated variable destroyed?

```
1  #include <iostream>
2
3  int add(int x, int y) // x and y are created here
4  {
5      // x and y are visible/usable within this function only
6      return x + y;
7  } // x and y go out of scope and are destroyed here
8
9  int main()
10 {
11     int a = 5; // a is created and initialized here
12     int b = 6; // b is created and initialized here
13     // a and b are usable within this function only
14     std::cout << add(a, b) << std::endl; // calls function add() with
   x=a and y=b
15     return 0;
16 } // a and b go out of scope and are destroyed here
```

- Note that if function add() were to be called twice, parameters x and y would be created and destroyed twice -- once for each call. In a program with lots of functions, variables are created and destroyed often.

Local scope prevents naming collisions

- each function doesn't need to care what other functions, including caller, name their variables.

```
1  #include <iostream>
2
3  int add(int x, int y) // add's x is created here
4  {
5      return x + y;
6  } // add's x goes out of scope and is destroyed here
7
8  int main()
9  {
10     int x = 5; // main's x is created here
11     int y = 6;
12     std::cout << add(x, y) << std::endl; // the value from main's x is
   copied into add's x
13     return 0;
14 } // main's x goes out of scope and is destroyed here
```

What does the following program print?

```
1  #include <iostream>
2
3  void doIt(int x)
4  {
5      x = 3;
6      int y = 4;
7      std::cout << "doIt: x = " << x << " y = " << y << std::endl;
8  }
9
10 int main()
11 {
12     int x = 1;
13     int y = 2;
14     std::cout << "main: x = " << x << " y = " << y << std::endl;
15     doIt(x);
16     std::cout << "main: x = " << x << " y = " << y << std::endl;
17     return 0;
18 }
```

Why functions are useful, and how to use them effectively

Why functions are useful, & how to use them effectively

- **Organization组织**: all code inside main() => complicated => divide & conquer 分而治之
- **Abstraction 抽象/封装**
  - In order to use a function, you only need to know its name, inputs, outputs. You don't need to know how it works. This is super-useful for making other people's code accessible (such as everything in the standard library).
- **Testing 测试**
  - functions are self-contained, once be tested to ensure it works, don't need to test it again unless it is changed
- **Reusability**: 可重用性
  - called multiple times from within the program and other program.
  - This avoids duplicated code
  - minimizes the probability of copy/paste errors.
- **Extensibility可扩展性**
  - When we need to extend our program to handle a case it didn't handle before, functions allow us to make the change in one place and have that change take effect every time the function is called. 动态库dynamic libary，补丁patch

When to write functions

- Codes that appear more than once.
- A function should generally perform one (and only one) task.
- Codes for accomplish a task are too long
- When a function becomes too long, too complicated, or hard to understand, it should be split into multiple sub-functions.
- This is called **refactoring重构**.

function – no more than 1 page

- Typically, when learning C++, you will write a lot of programs that involve 3 subtasks:
  1. Reading inputs from the user
  2. Calculating a value from the inputs
  3. Printing the calculated value

- For trivial programs (e.g. less than 20 lines of code), some or all of these can be done in main().
- However, for longer programs (or just for practice) each of these is a good candidate for an individual function

function – no more than 1 page

- New programmers often **combine calculating a value and printing** the calculated value into a single function.

- **this violates the "one task" rule of thumb for functions.**

- A function that calculates a value should return the value to the caller and let the caller decide what to do with the calculated value (such as call another function to print the value).

下一小节：
内联函数、函数重载和函数缺省参数

内联函数

- 函数调用是有时间开销的。如果函数本身只有几条语句，执行非常快，而且函数被反复执行很多次，相比之下调用函数所产生的这个开销就会显得比较大。

- 为了减少函数调用的开销，引入了内联函数机制。编译器处理对内联函数的调用语句时，是将整个函数的代码插入到调用语句处，而不会产生调用函数的语句。

内联函数

```
inline int Max(int a,int b)
{
    if( a > b)
        return a;
    return b;
}
```

函数重载 function overloading

- 多个函数，名字相同，然而参数个数或参数类 型不相同，这叫做函数的重载。
- 以下三个函数是重载关系：

```
int Max(double f1,double f2) {    }
int Max(int n1,int n2) {   }
int Max(int n1,int n2,int n3) {   }
```

- 函数重载使得函数命名变得简单。
- 编译器根据调用语句的中的实参的个数和类型判断应该调用哪个函数。

函数重载

(1) int Max(double f1,double f2) { }
(2) int Max(int n1,int n2) { }
(3) int Max(int n1,int n2,int n3) { }

```
Max(3.4,2.5);      //调用 (1)
Max(2,4);          //调用 (2)
Max(1,2,3);        //调用 (3)
Max(3,2.4);        //error,二义性
```

No function overloading in Python

- accomplishing the same task in Python requires a different technique.

```
def myfunct(n, m=None):
    if m is None:
        print("1 parameter: " + str(n))
    else:
        print("2 parameters: " + str(n), end="")
        print(" and ", str(m))

myfunct(4)
myfunct(5, 6)
```

缺省参数Default parameters in C++ & Python

- C++中，定义函数的时候可以让最右边的连续若干个参数有缺省值，那么调用函数的时候，若相应位置不写参数，参数就是缺省值。

```
void func( int x1, int x2 = 2, int x3 = 3) {  }

func(10 ) ; //等效于 func(10,2,3)
func(10,8) ; //等效于 func(10,8,3)
func(10, , 8) ; //不行,只能最右边的连续若干个参数缺省
```

函数的缺省参数：

- 函数参数可缺省的目的在于提高程序的可扩充性。

- 即如果某个写好的函数要添加新的参数，而原先那些调用该函数的语句，未必需要使用新增的参数，那么为了避免对原先那些函数调用语句的修改，就可以使用缺省参数。