Subprograms and Functions

pm_jat@daiict.ac.in

Subprograms

- Subprograms are fundamental building blocks of programs and important concept in programming
- Divide and conquer has been one of the important software design strategy, and subprograms are just outcome of that
- Abstraction plays an important role in while having this decomposition
- Abstraction, in context of software development is, seeing *user's* view of software component while hiding implementation details

Subprograms

- Two Fundamental *abstraction* techniques in software Design-
  - Functional Abstraction, and
  - Data Abstraction
- Functional abstraction, also known as procedural abstraction or sometimes process abstraction; has been extensively used in program design, particularly for imperative programming.
- Data abstraction has been the basis of Object Orientated Programming
- The way abstraction is made, differentiates Procedural approach and Object Oriented program design

Subprograms

- Through Functional Abstraction, we identify number of sub-programs to have functionality to be accomplished by the program
- Typically we have hierarchy of sub-programs
- Functional Abstraction
  - – lets us view subprogram as a black box, doing specific computing task;
  - – specify input to it and output from it
- Subprograms is though of as a relatively small unit of computation, doing a specific task, and can be accomplished by a small number of programming statements

Subprograms

- Subprogram, have input and output. Inputs is usually through parameters to the subprogram while output could be through parameters or as a *return*
- Conventionally, there are two categories of subprograms- procedures and functions

- Subprograms having return are called functions; and others are called procedures

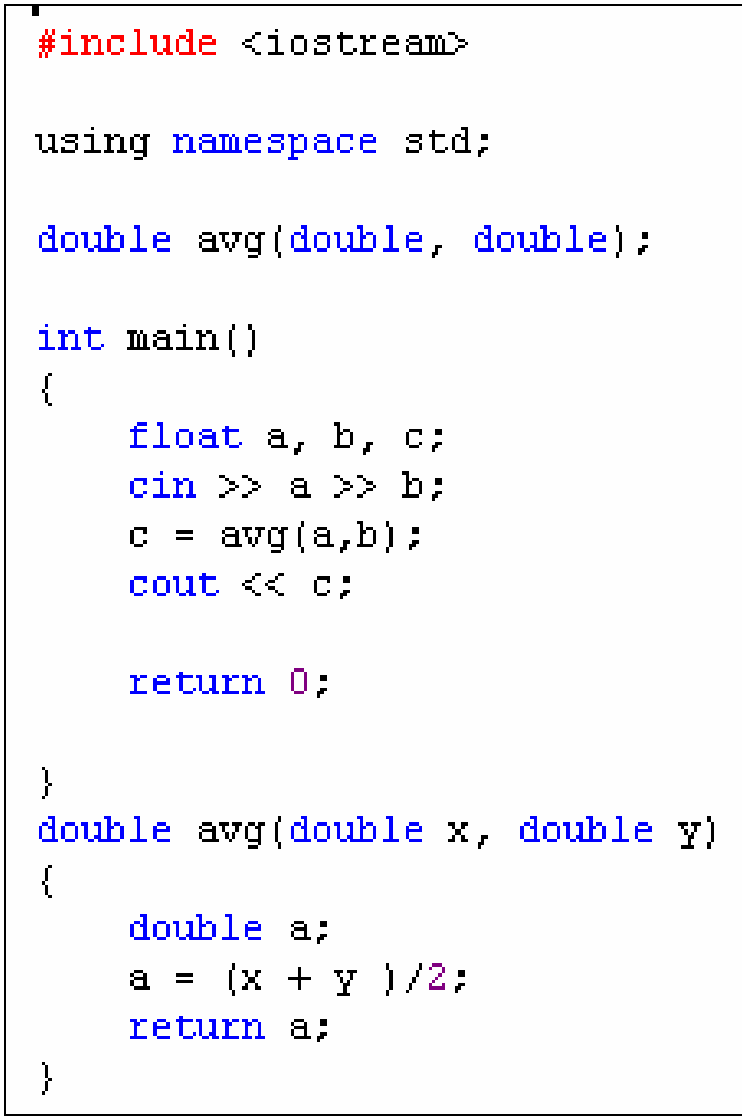
- C/C++/Java have only functions

Subprogram: Basic Definitions

- Subprogram Definition: describes *interface* and the *actions* of the subprogram
- Subprogram call is the explicit request that the called subprogram is executed
- Subprogram header: Which is first part of definition, defines its name, Input (parameters), and outputs and their types
- Subprogram Interface: Also called signature of subprogram, includes name of subprogram, list of parameters with their type, and output(s) and their types. For example prototypes in C/C++

C/C++ Subprograms

- C/C++ supports function subprograms
- C++ functions provides good “data hiding” mechanism .. It is very much like black box
- Local variables are visible within the function; there is no way you can access local variables of one function into another function- while execution in function avg, no variable of function main can be accessed
- Parameters are means by which function share data
- Though global variables is issue, may have side effects; can be avoided by having next level proper “encapsulation”



```
#include <iostream>

using namespace std;

double avg(double, double);

int main()
{
    float a, b, c;
    cin >> a >> b;
    c = avg(a,b);
    cout << c;

    return 0;

}
double avg(double x, double y)
{
    double a;
    a = (x + y )/2;
    return a;
}
```

Function Prototypes in C/C++

- What is purpose prototypes serve in C/C++ ?
- What information does it contain?
- Why do we need to declare it or include a header file?
- Why main function does not have a prototype?

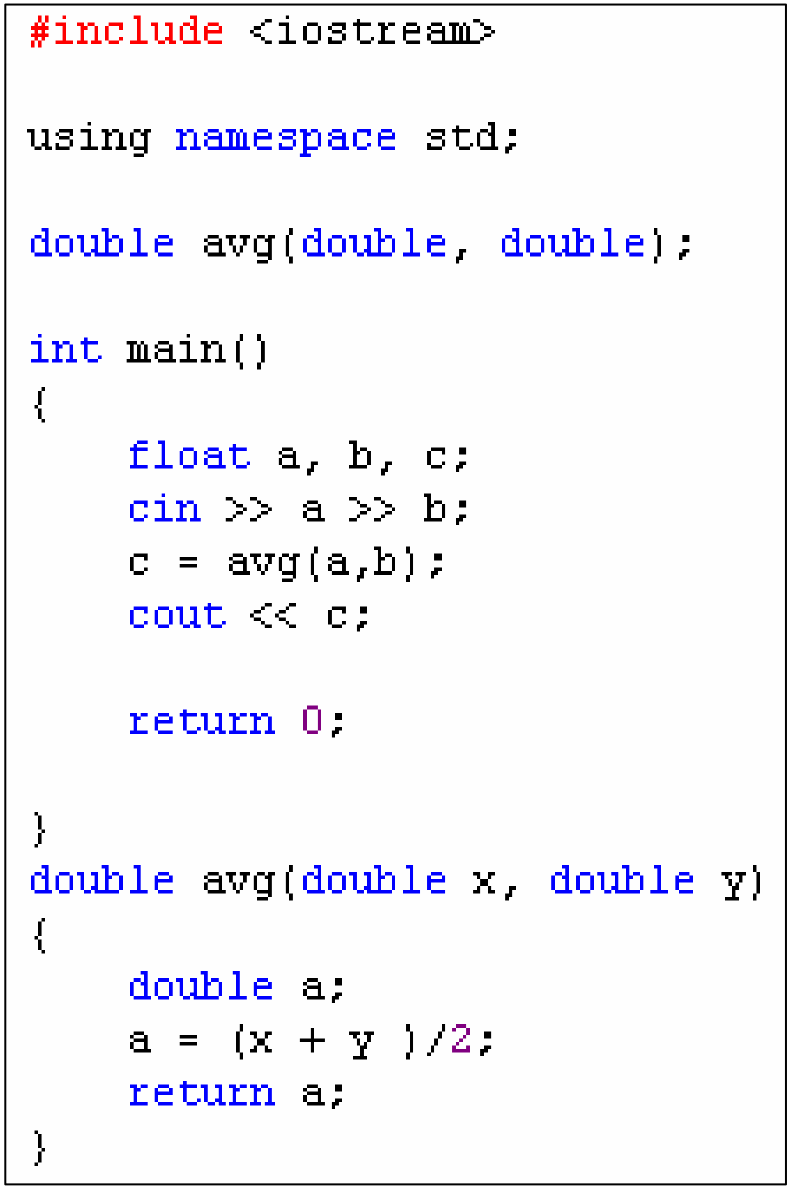
Function Prototypes in C/C++

- What is purpose prototypes serve in C/C++ ?
  - Know signature of function, for both programmer and compiler
- What information does it contain?
  - Name of function, its parameter list and their type, return type
- Why do we need to declare it or include a header file?
  - So that compiler know signature of function and can have type checking
- Why main function does not have a prototype?
  - Whole definition of function is right there !!

Formal Parameters and Actual Parameters

- When you define function, you have formal parameter
- When you make a call to function you pass actual parameters
- Binding of formal and actual parameters are mostly positional, some languages allow to have named binding

Formal Parameters and Actual Parameters C++ Example:



```
#include <iostream>

using namespace std;

double avg(double, double);

int main()
{
    float a, b, c;
    cin >> a >> b;
    c = avg(a,b);
    cout << c;

    return 0;

}
double avg(double x, double y)
{
    double a;
    a = (x + y )/2;
    return a;
}
```

- Here x, and y are formal parameter while a, and b in function main are actual parameters

How C++ functions are executed?

- Supports the function call and return mechanism
- It is like a hierarchy of functions; a C/C++ program composed of a single main function, it makes call to some other function f1, that in turn makes call to another function f11, and so forth to any depth
- Each function/subprogram, on completion of its execution returns control to the function which called it
- During execution of called function, execution of calling function is halted; and when execution on called function is completed, execution of calling program resumes at immediately after the call of the function.

How C++ functions are executed?

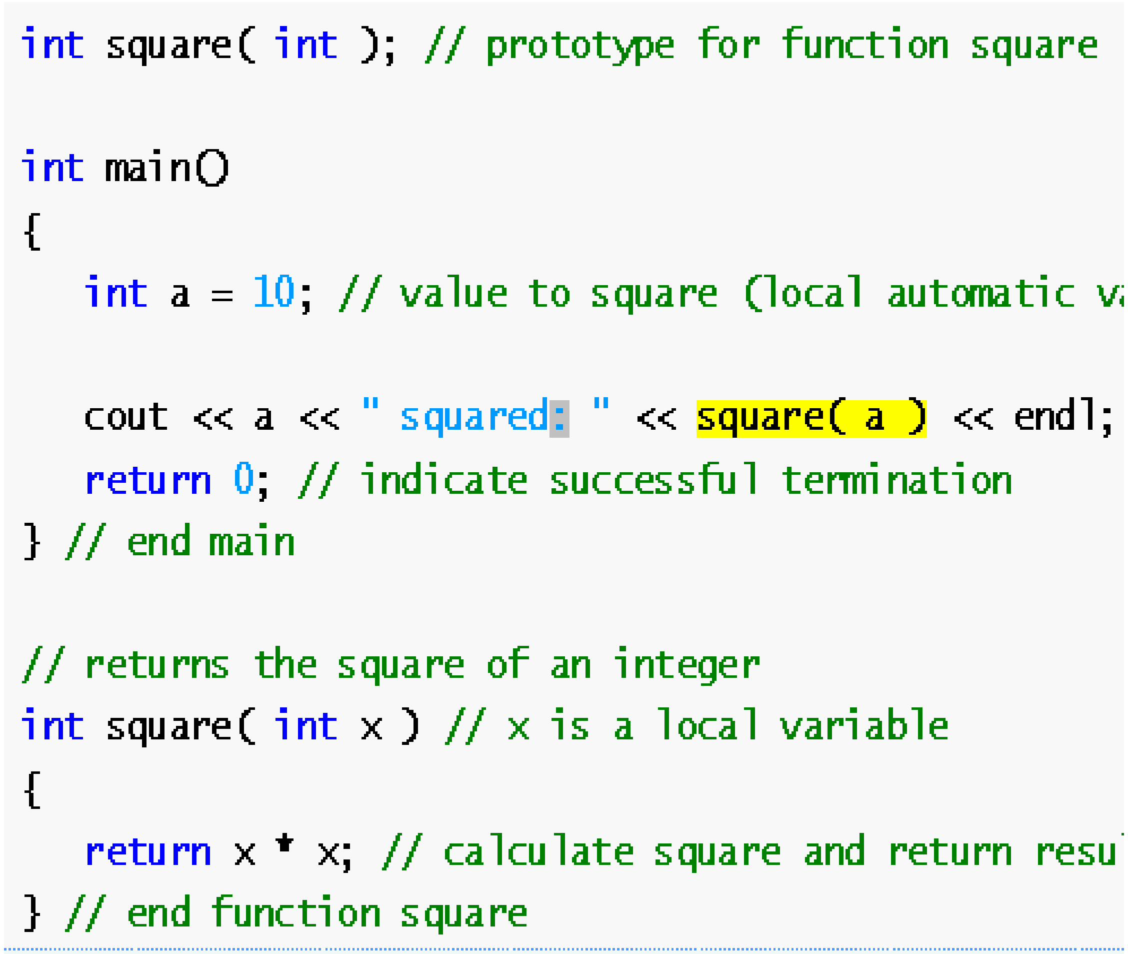
- Each time a function calls another function, an activation record is pushed onto the execution stack

- Activation record
  - Maintains the return address that the called function needs to return to the calling function
  - Contains automatic variables—parameters and any local variables the function declares

Example

- Consider a simple example from deitel and deitel

- Let us see how call and return works



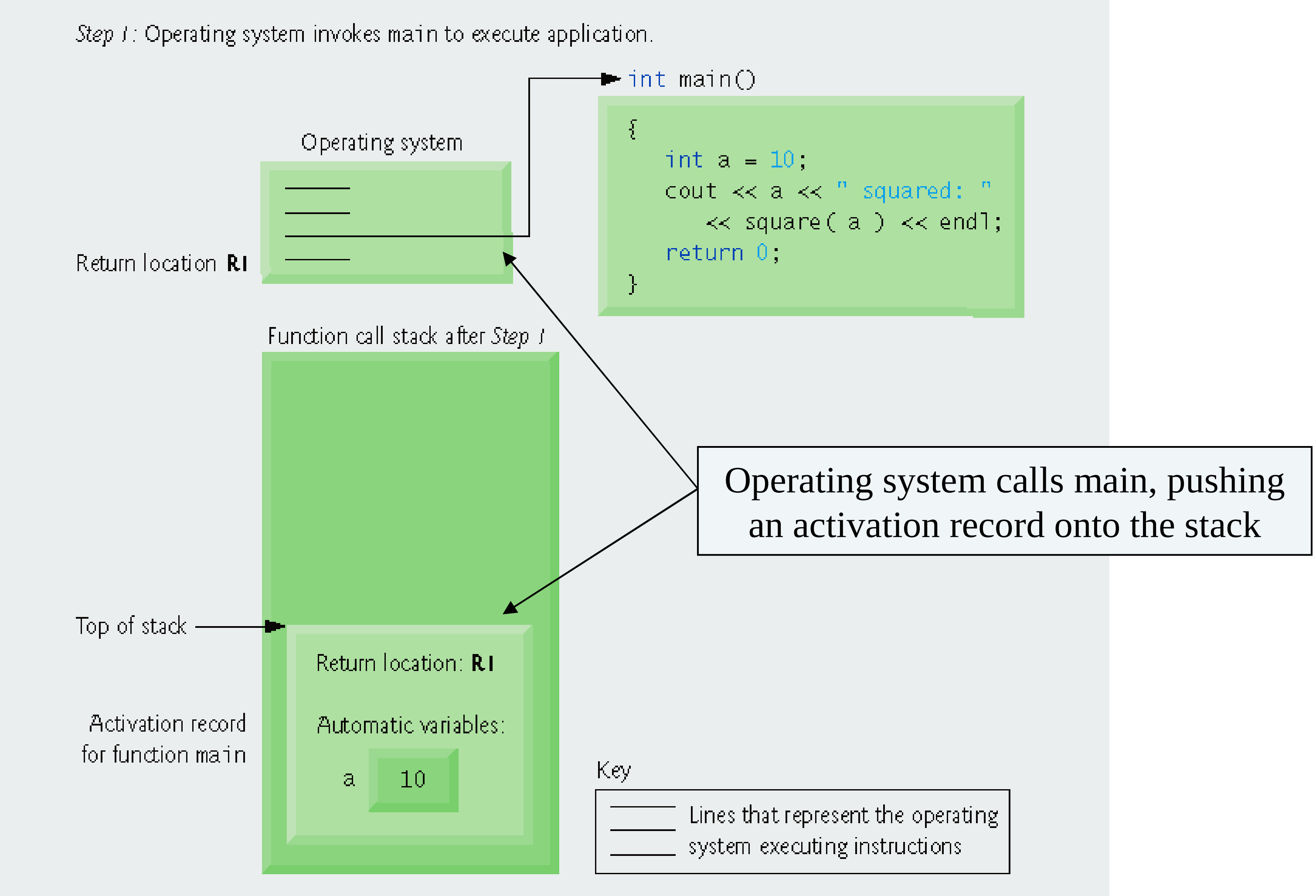
```
int square( int ); // prototype for function square

int main()
{
   int a = 10; // value to square (local automatic va

   cout << a << " squared: " << square( a ) << endl;
   return 0; // indicate successful termination
} // end main

// returns the square of an integer
int square( int x ) // x is a local variable
{
   return x * x; // calculate square and return resu
} // end function square
```

Function call stack after the operating system invokes `main` to execute the application.

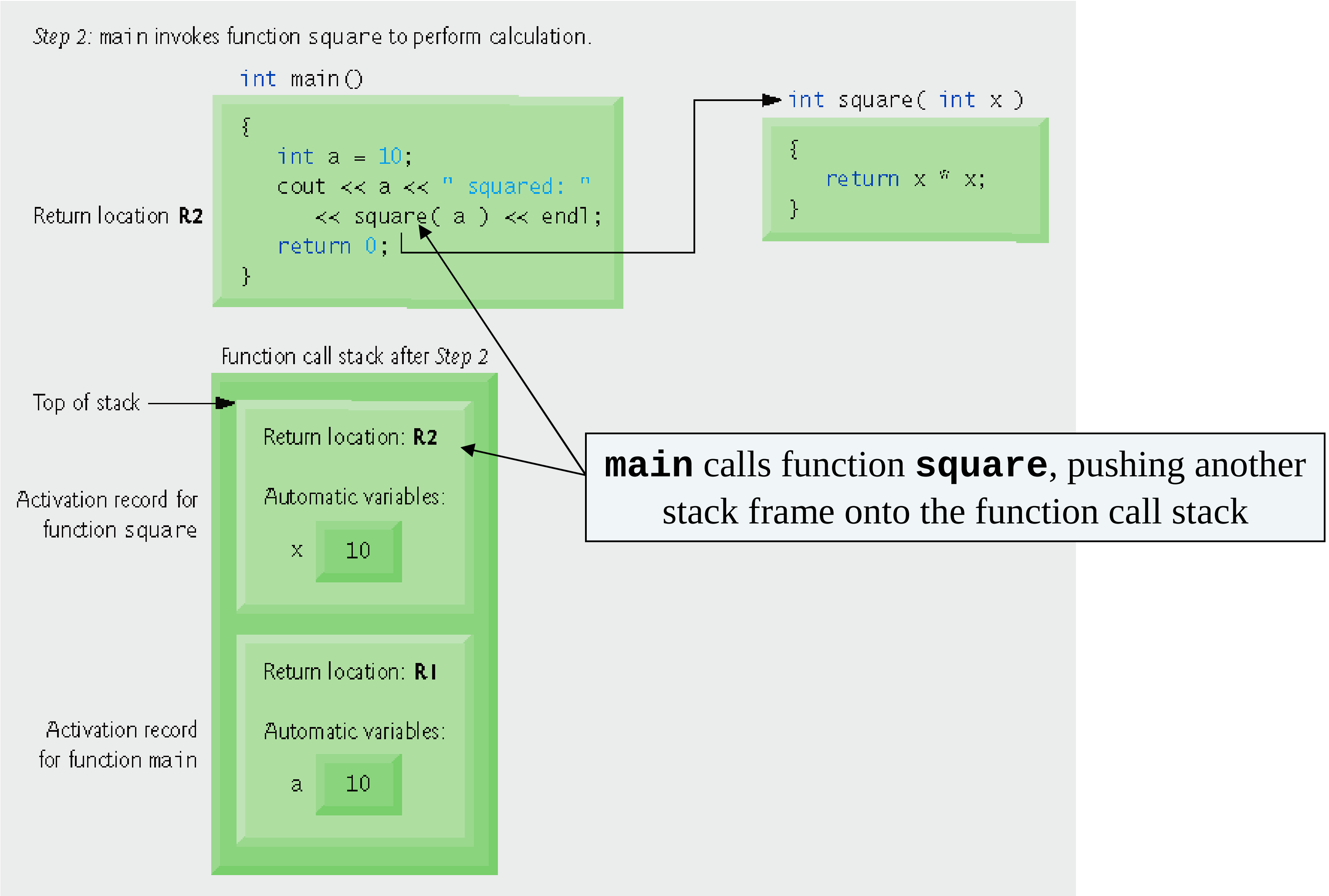


Step 1: Operating system invokes main to execute application.
Operating system
Return location R1
Function call stack after *Step 1*
Top of stack
Activation record for function main
Return location: R1
Automatic variables:
a
10
Key
Lines that represent the operating system executing instructions

```
int main()
{
   int a = 10;
   cout << a << " squared: "
      << square( a ) << endl;
   return 0;
}
```

Operating system calls main, pushing an activation record onto the stack

Function call stack after `main` invokes function `square` to perform the calculation.



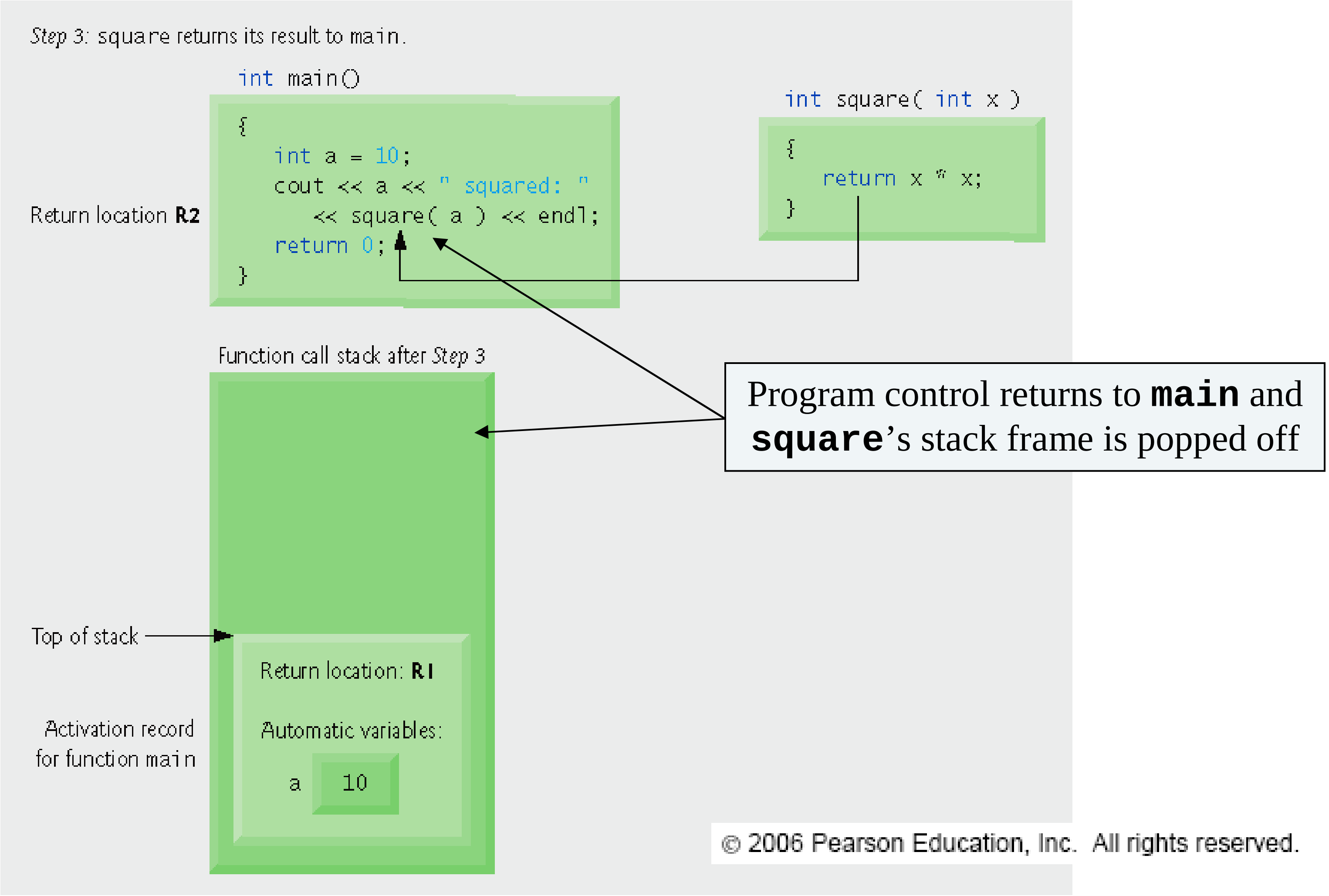
Step 2: main invokes function square to perform calculation.

```
int main()
{
   int a = 10;
   cout << a << " squared: "
      << square( a ) << endl;
   return 0;
}
```

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

`main` calls function **`square`**, pushing another stack frame onto the function call stack

Function call stack after function `square` returns to `main`.



Step 3: `square` returns its result to `main`.

```
int main()
{
   int a = 10;
   cout << a << " squared: "
      << square( a ) << endl;
   return 0;
}
```

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

Program control returns to **`main`** and **`square`**’s stack frame is popped off

Parameter Passing Methods

- Parameter Passing Methods are ways in which parameters are transmitted to and/or from called program
- Formal Parameters are characterized by one of three distinct semantic models:
  - They can receive data from the corresponding actual parameter
  - They can transmit data to the actual parameter
  - They can do both
- These three semantic models are called **in mode**, **out mode**, and **inout mode**, respectively

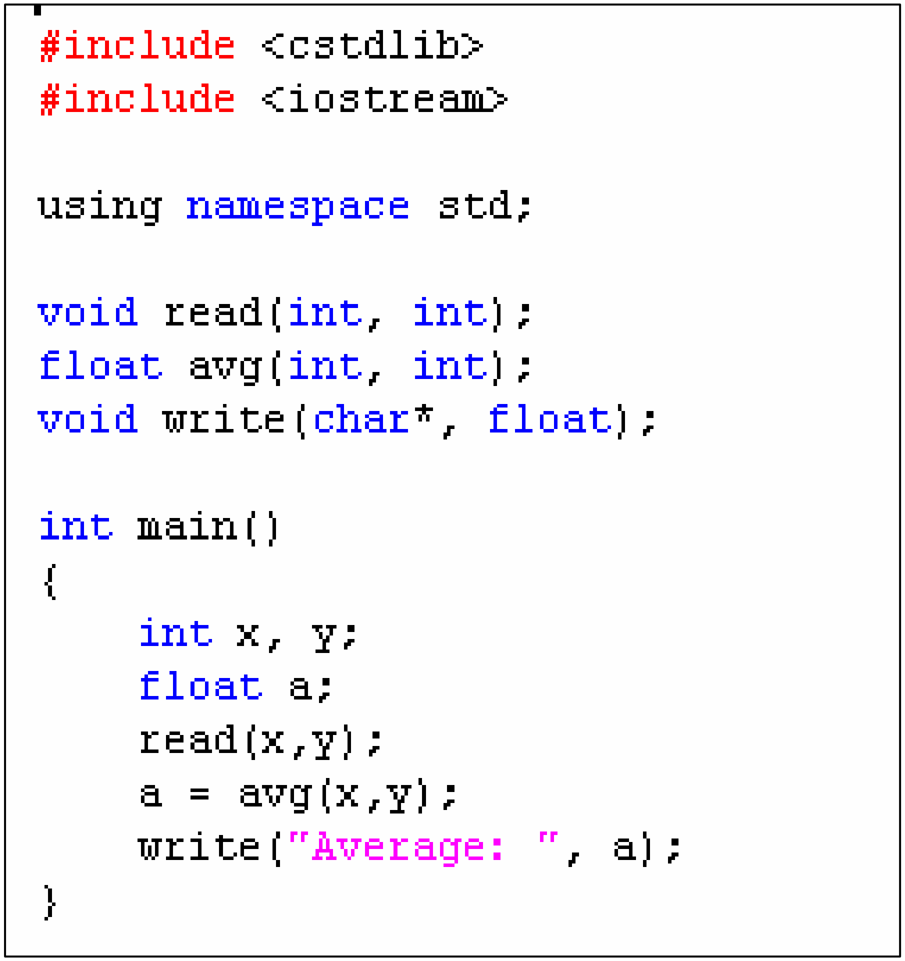
Example

- For example, consider following function calls

  ```
  read(x, y);
  a = pow(x,y)
  doubleIt(a);
  swap(x,y);
  ```

- What semantic model of these functions?
- read function: x, y are **out mode**
- pow function: x,y are in mode
- doubleMe function: a inout mode
- swap function: x, y inout mode

C++ Example
Will this program work as expected?



```
#include <cstdlib>
#include <iostream>

using namespace std;

void read(int, int);
float avg(int, int);
void write(char*, float);

int main()
{
    int x, y;
    float a;
    read(x,y);
    a = avg(x,y);
    write("Average: ", a);
}
```

No
Because parameters of read function are in “in mode”

Where as should be in “out mode”

How do we so say so?

Thanks