Module 5 Guidance Notes

Functions

ENGG1340
Computer Programming II

COMP2113
Programming Technologies

Estimated Time of Completion: 2.5 Hours

Outline

- (P. 3 – 7) Top-down design (divide and conquer) approach
- (P. 8 – 27) Pre-defined functions vs. self-defined functions
- (P. 28 – 37) Functions definition, function call & function declaration
- (P. 38 – 51) Flow of control
- (P. 52 – 61) Scope of Variables
- (P. 62 – 79) Parameters passing mechanism
    - Pass-by-value
    - Pass-by-reference

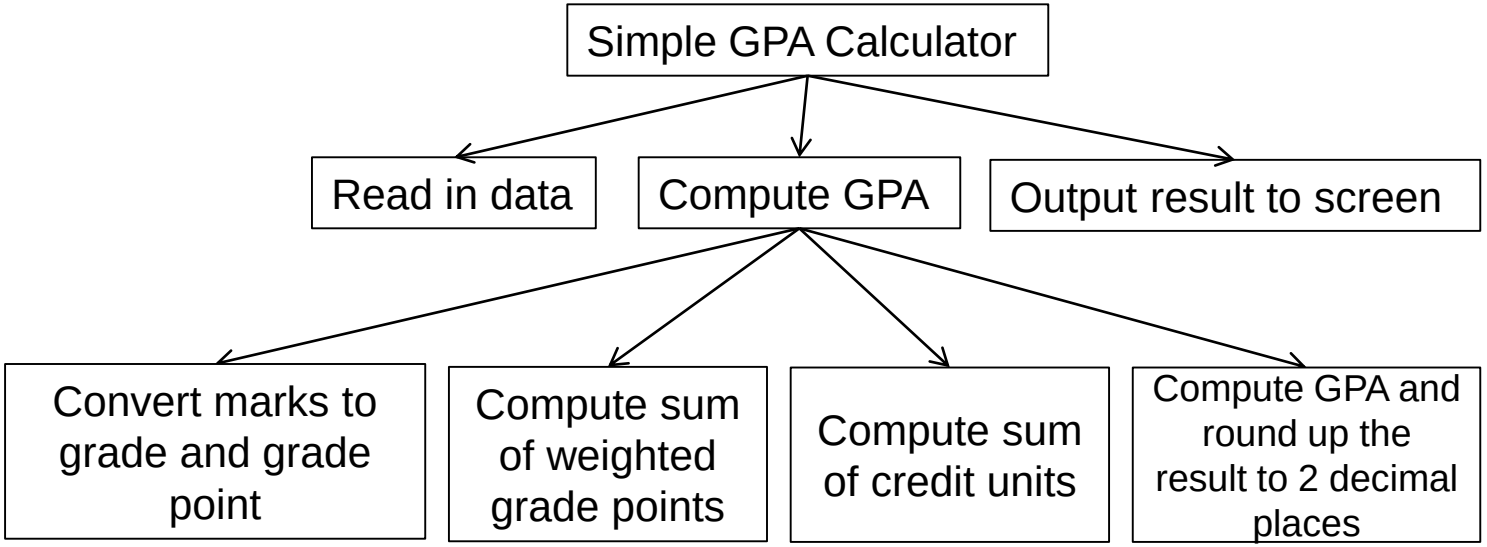
TOP-DOWN DESIGN (DIVIDE AND CONQUER) APPROACH

Top-Down Program Design

- A good way to design a program is to break down the task to be accomplished into a few **sub-tasks**
- Each sub-task can be further decomposed into smaller sub-tasks, and this process is repeated until all sub-tasks are small enough that their implementations become manageable
- This approach is called top-down design (a.k.a. divide and conquer)

Top-Down Design

- Example: Compute the final score for a student



Each module should perform a single, well-defined task

Functions

- Preserving the top-down design structure in a program will make it easier to understand and modify the program, as well as to write, test, and debug the program.
- In C++, sub-tasks are implemented as functions.
  - A function is a group of statements that is executed when it is called from some point of the program.
  - E.g., the main function `main()` in previous examples
- A program is composed of a collection of functions.
- When a program is put into execution, **it always starts at the main function**, which may in turn call other functions.

Advantages of Using Functions

- May focus on a particular task, easy to construct and debug
- Different people can work on different functions simultaneously.
- A function is written once and can be reused multiple times in a program or in different programs.
- Improve readability of a program by reducing the complexity of `main()`

PREDEFINED FUNCTIONS VS. SELF-DEFINED FUNCTIONS

Predefined Functions

- Some computations and operations are so common that they are implemented as pre-defined functions that are shared for use.
- Consider computing the square root of a number. It would be nice if we have a black box function (i.e., we don't care how the computation is done) to help us do the calculation.

e.g.

```
double x = sqrt(5.29);
```

Here, 5.29 is the function input and the function output 2.3 would be stored to x.

- All that you need to know to use a predefined function is:
  - what is required for the computation (i.e., function input); and
  - what is the result of the computation (i.e., function output)

Predefined Functions

- C++ comes with libraries of pre-defined functions that programmers can use in their programs. A library contains
  - Function definitions (i.e., codes containing the body of a function for doing the actual computations) that are stored in separate files and have been **pre-compiled** into object codes for further linking.
  - Function declarations (i.e., what a function accepts as input and returns as output) that are stored in files known as the header files.

Examples

- Some commonly used predefined functions. It means that you can make use of them in your program for a particular task (e.g., computing the square root of a number) without the need of writing that part of the code on your own.

The file containing all the function declarations

Function declarations	Description	Library Header
`double sqrt(double x)`	Square root of x	`cmath`
`double pow(double x, double y)`	x to the power of y (i.e., x^y)	`cmath`
`double fabs(double x)`	Absolute value of x	`cmath`
`double ceil(double x)`	Round up the value of x	`cmath`
`double floor(double x)`	Round down the value of x	`cmath`
`int abs(int x)`	Absolute value of x	`cstdlib`
`int rand()`	A random integer	`cstdlib`

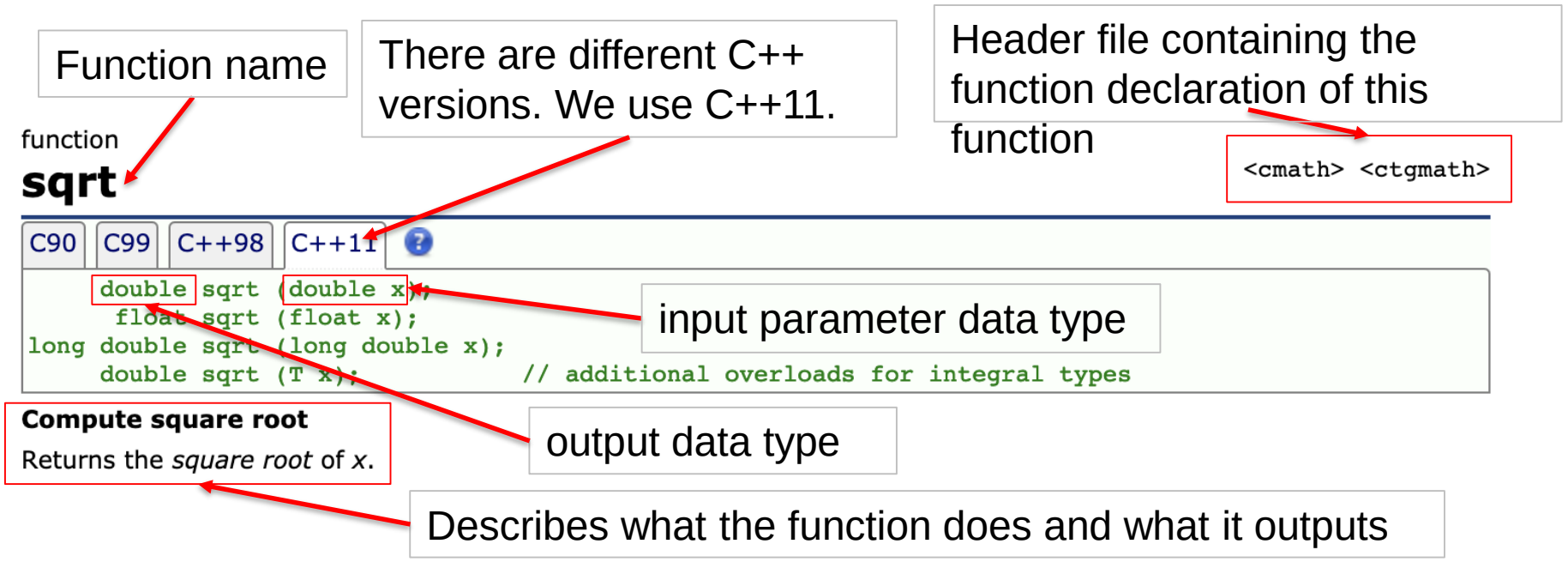
- You can consult the C++ manuals, e.g. www.cplusplus.com, for the details of individual functions.

Using Predefined Functions

- It is very important for a developer to be able to use predefined functions, since there are many libraries out there already well-written (and well-tested) by others for some specific purposes, e.g., math libraries, image handling libraries, linear algebra libraries.
- The function declarations in the header files should tell how we may use each function in the given library, but usually we will go to the library documentation (or manual/reference) to look at the usage details for each function.
- It is therefore also very important for a developer to be able to read and understand library documentation.

Function Reference Example

This is what you can find from the [sqrt() function reference] from cplusplus.com:



Note that a function may accept different data types as inputs (hence there are 4 different function declarations for sqrt()).
This is called function overloading.
In this case, if you provide a double type input to sqrt(), the output returned by the function will be of a double type, if the input to sqrt() is of a float type, then the output would be of a float type.

Using Predefined Functions

- To use a pre-defined function, simply include the corresponding header file using the include directive `#include <...>` at the beginning of the file containing your code
  - e.g., `#include<iostream>` for using `cin`, `cout`, `endl`
  - e.g., `#include<cstdlib>` for using `rand()`, `srand()`
- This step is mandatory so that the compiler can check if the functions are used correctly. Recall that the header contains how the functions should be used.

- To use a predefined function, read its function reference and take note of the input parameters and the return type for a function and make the function call accordingly.

Using Predefined Functions

Example:

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

int main() {
    // Compute the root mean square of 10 input numbers
    int i;
    double n, sq_sum = 0;

    for(i=0; i<10; i++)
    {
        cout << i+1 << ": ";
        cin >> n;
        sq_sum += pow(n, 2.0);
    }

    cout << "The root mean square is " << sqrt(sq_sum/10) << endl;

    return 0;
}
```

Include <cmath> so you can use the pow() and sqrt() functions from the math library later on in the same program file.

A function may accept one or more input parameters. The order and type of each parameter matter. Check the pow() reference page to see what each parameter mean.

Example: Random Number Generation

We need random numbers under many circumstances, e.g., games in which we need to generate random scenarios for the players.

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

int main()
{
  int num = 23;
  int guess;
  bool isGuessed;

  isGuessed = false;

  while (!isGuessed) {
    cout << "Make a guess (0-99)? ";
    cin >> guess;

    if (guess == num) {
      cout << "Correct!" << endl;
      isGuessed = true;
    }
    else if (guess < num)
      cout << "Too small.  Guess again!" << endl;
    else
      cout << "Too large.  Guess again!" << endl;
  }
  return 0;
}
```

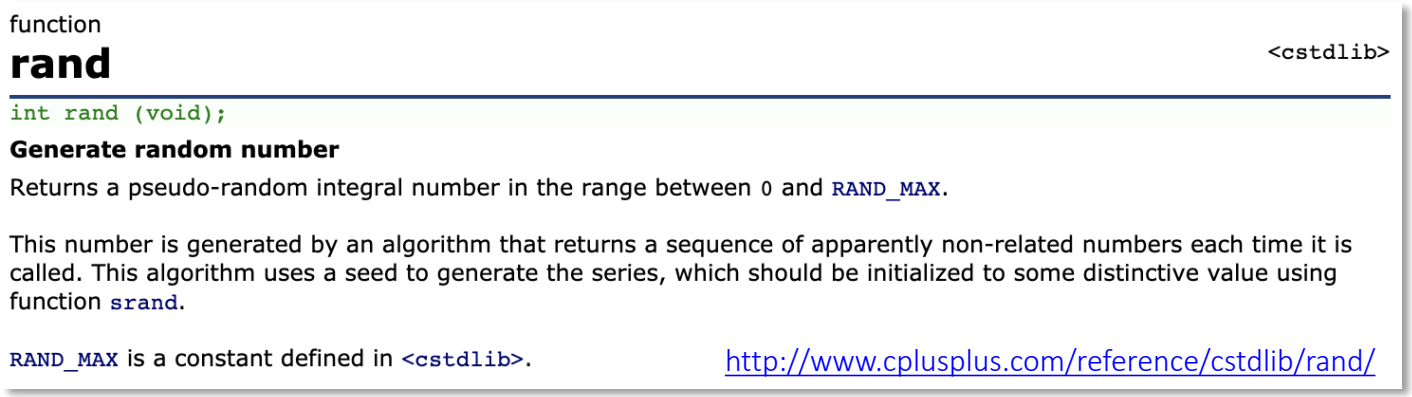
Recall the simple number guessing game in Module 3.

We can see that the answer is always 23 no matter how many times you run the program, as this "answer" is hard-coded into the program. This is not so interesting as a game.

Let's try add in some randomness for this game by making your program generate a different number randomly every time it is run.

Example: Random Number Generation

We need the rand() function in <cstdlib> to generate a random number.



function

rand <cstdlib>

```
int rand (void);
```

Generate random number

Returns a pseudo-random integral number in the range between 0 and RAND_MAX.

This number is generated by an algorithm that returns a sequence of apparently non-related numbers each time it is called. This algorithm uses a seed to generate the series, which should be initialized to some distinctive value using function srand.

RAND_MAX is a constant defined in <cstdlib>.

http://www.cplusplus.com/reference/cstdlib/rand/

Look at what it returns. The function returns a random integer in the range [0, RAND_MAX], and RAND_MAX is a constant defined in <cstdlib>.

So how can we make use of it for generating a random integer in the range of [0, 9]?

Hint: use the modulus operator %, which computes the remainder of a division.

Example: Random Number Generation

Q.1 how to generate a random integer in the range of [0, 9]?

Answer:

```
rand() % 10
```

Q.2 how to generate a random integer in the range of [0, 100]?

Answer:

```
rand() % 101
```

Q.3 how to generate a random integer in the range of [1, 100]?

Answer:

```
rand() % 100 + 1
```

Example: Random Number Generation

Take look at this program. What does it do?

```
#include <iostream>
#include <cstdlib>       // needed for calling rand()
using namespace std;

int main()
{
  for (int i = 0; i < 10; ++i)
    cout << rand() % 100 + 1 << endl;
  return 0;
}
```

It generates and prints 10 random integers in the range [1,100].

Now, try to run the program. Write down the numbers that it generated.

Run the program again. And again. And again. What do you notice?

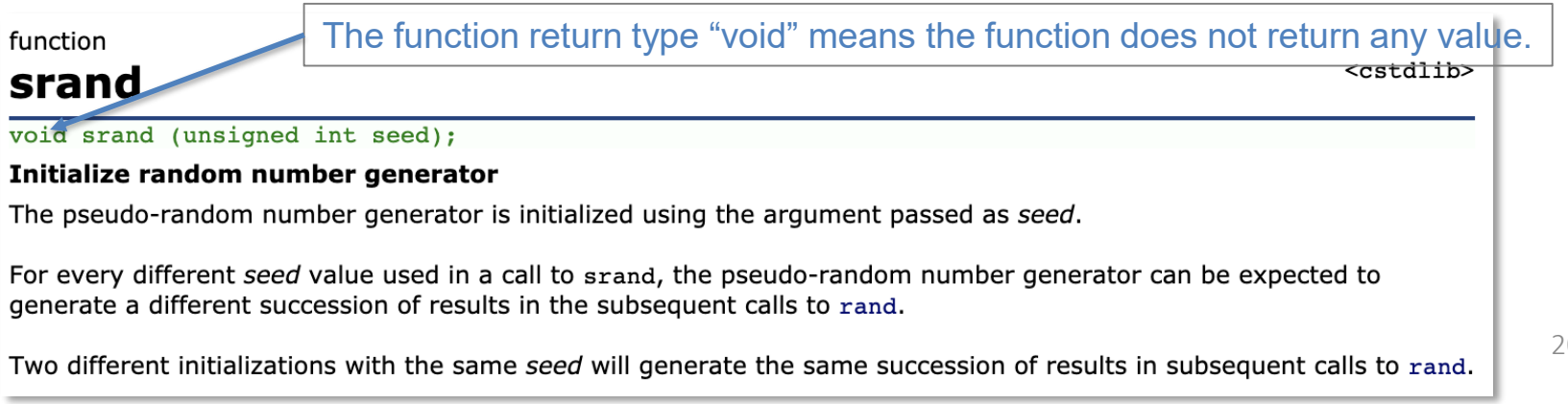
Example: Random Number Generation

The program generates the same 10 numbers for every run. So it's not that "random" after all. Why is that?

The algorithm for generating the sequence of random numbers depends on an initial "seed" number. Given the same seed, the same sequence of numbers will be generated. They are "random" in the sense that the distribution of the numbers in the sequence is random.

This is why when you run your program, it always generates the same sequence because we didn't change the seed.

The seed for the random number generator can be specified by srand() function.



Example: Random Number Generation

So our program should look something like this:

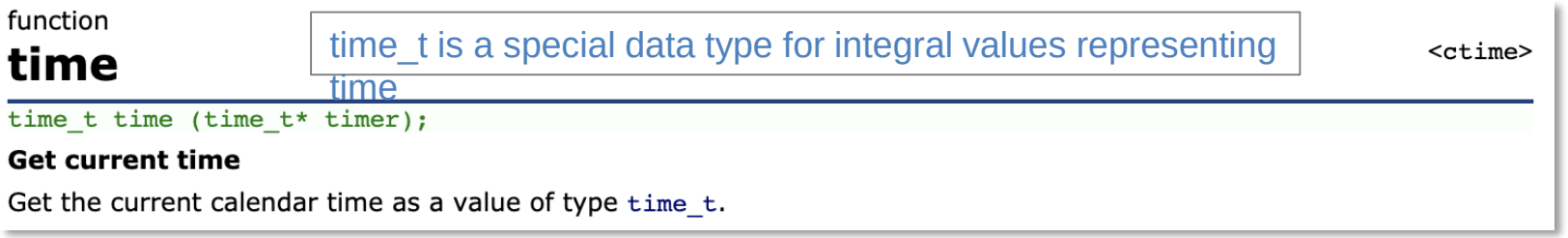
```
#include <iostream>
#include <cstdlib>       // for calling srand(), rand()
using namespace std;

int main()
{
  srand(???);     // initialize the seed for rand()
  for (int i = 0; i < 10; ++i)
    cout << rand() % 100 + 1 << endl;
  return 0;
}
```

Now what should we fill in for ??? ? The srand() takes a number which will be used as the seed for generating number. But we need a different number every time we run the program.

How about taking the current system time as the seed? In this way, we can guarantee a different runtime value for every program run.

Example: Random Number Generation

We need another predefined function to obtain the current system time:



http://www.cplusplus.com/reference/ctime/time/

Go to the function reference page and read the details about the parameters and return value, on how to get the current time using this time() function. The function call `time(NULL)` will return the current time.

Or, look at the example on the reference page for rand() (http://www.cplusplus.com/reference/cstdlib/rand/) and you can see that we can simply initialize the seed for the random number generator with the current time by

```
// initialize random seed
srand(time(NULL));
```

Example: Random Number Generation

Now plug all these into our program for generating 10 random numbers:

```
#include <iostream>
#include <cstdlib>      // for calling srand(), rand()
#include <ctime>        // for calling time()
using namespace std;

int main()
{
  srand(time(NULL)); // initialize the seed for rand()
  for (int i = 0; i < 10; ++i)
    cout << rand() % 100 + 1 << endl;
  return 0;
}
```

Run the program again. And again. And again. We are done!

Example: Random Number Generation

Note: It turns out to be very important that we can specify the seed value. Sometimes we do want to have the same sequence of random numbers to be generated for every run of our program, especially for debugging.

Suppose you have a set of random numbers as input to part of your code, then imagine how difficult it would be to debug your program if in every program run, these numbers are different because the behavior of your program would be different. In this case, you may want to fix the random seed, by supply the same number to `srand()`, such as `srand(0)`.

Now, a quick exercise. Can you modify the guessing game on P.19 so that it will generate a random number (say, from 1 to 50) for the player?

Next, you will start writing your own functions in your program.

Defining Your Own Functions

Suppose you want to have a function which tells which of the two given floating point numbers is larger.

These are the questions that you should ask (& answer):

Q1. What are the input(s) to the functions? What are their data type?

Two floating-point numbers, data type: double

Q2. What is the output of the function? What is its data type?

One floating-point number, data type: double

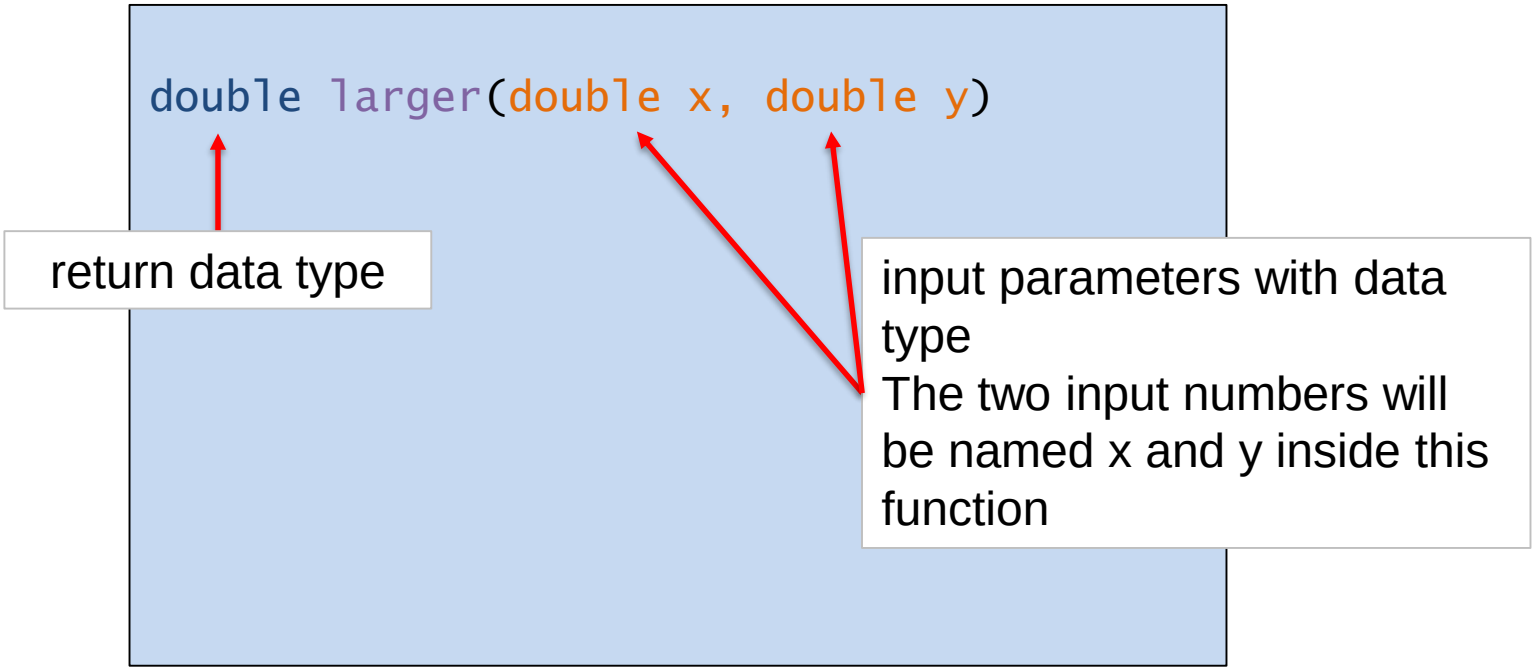
Q3. What should be done inside the function to make it work?

How do you determine which of the two given numbers are larger?

Defining Your Own Functions

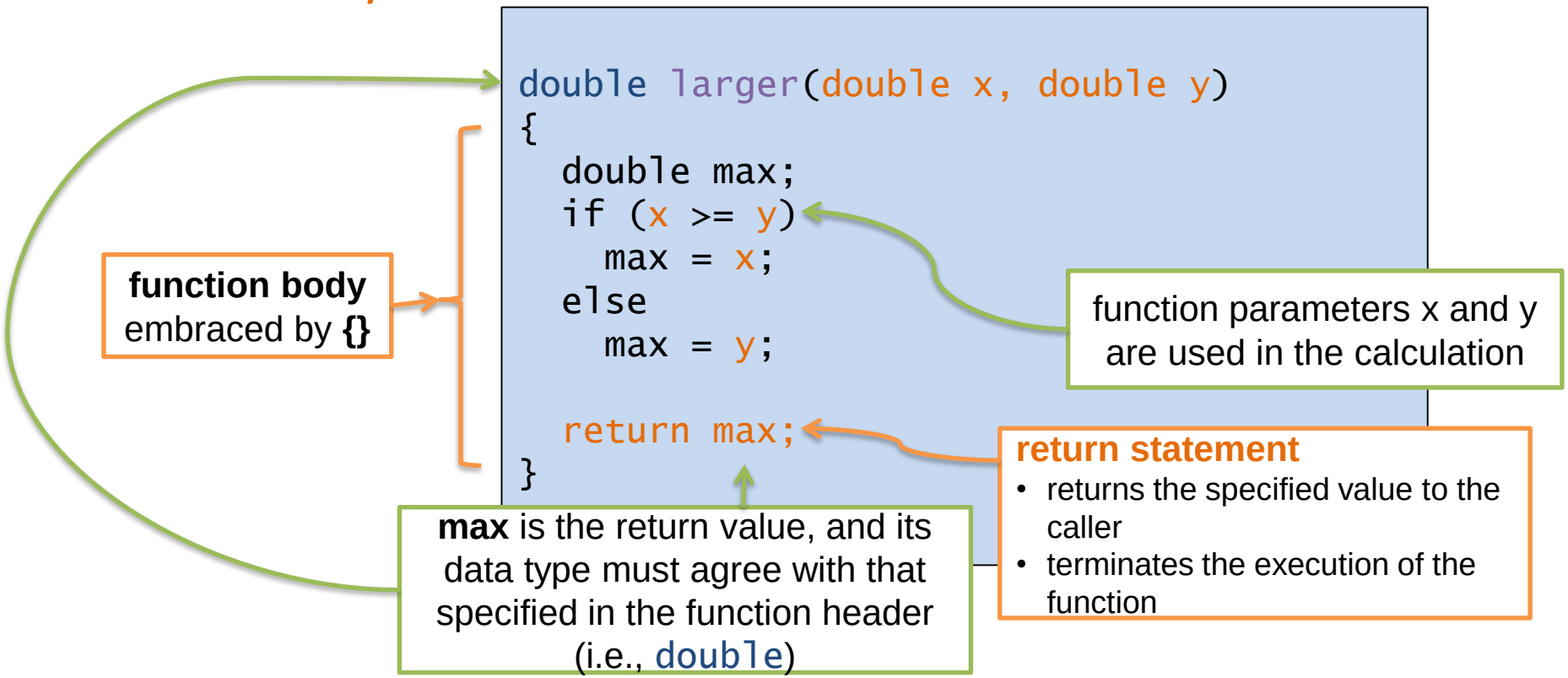
Let's give a name to the function: larger

By answering Q1 & Q2, we can come up with the function header



Defining Your Own Functions

To answer Q3, we need the actual computations inside the function body:

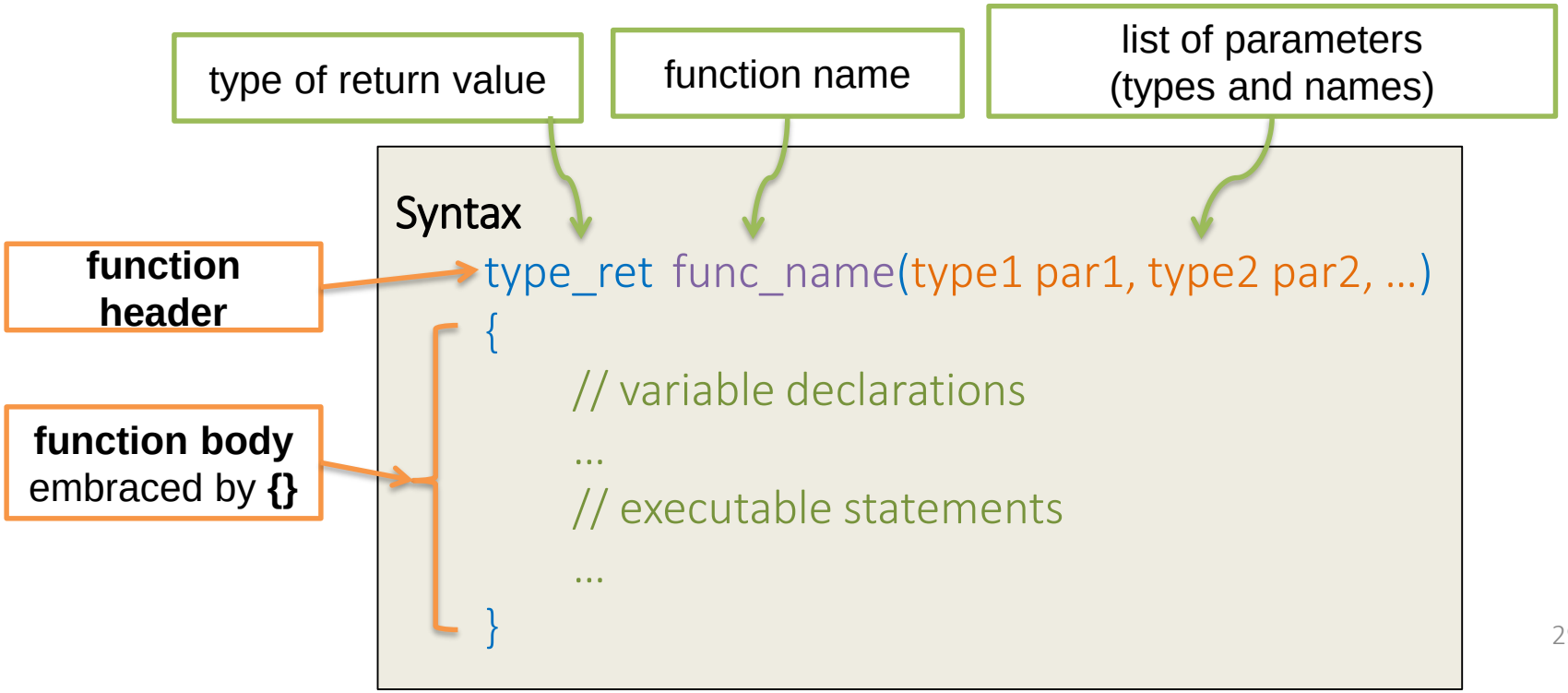


FUNCTION DEFINITION, FUNCTION CALL & FUNCTION DECLARATION

Function Definition

Formally speaking, a function is defined using a function definition which

- Describes how a function computes the value it returns
- Consists of a function header followed by a function body



Void Functions

- In some situations, a function simply carries out some operations and produces no return value.
- A function with no return value is called a void function.
- In this case, the void type specifier, which indicates absence of type, can be used.
- The return statement in a void function does not specify any return value. It is used to return the control to the calling function.
- If a return statement is missing in a void function, the control will be returned to the calling function after the execution of the last statement in the function.

Void Functions

Examples

```
void print_msg(int x)
{
  cout << "This is a void function " << x << endl;
  return;
}
```

A return statement with no return value

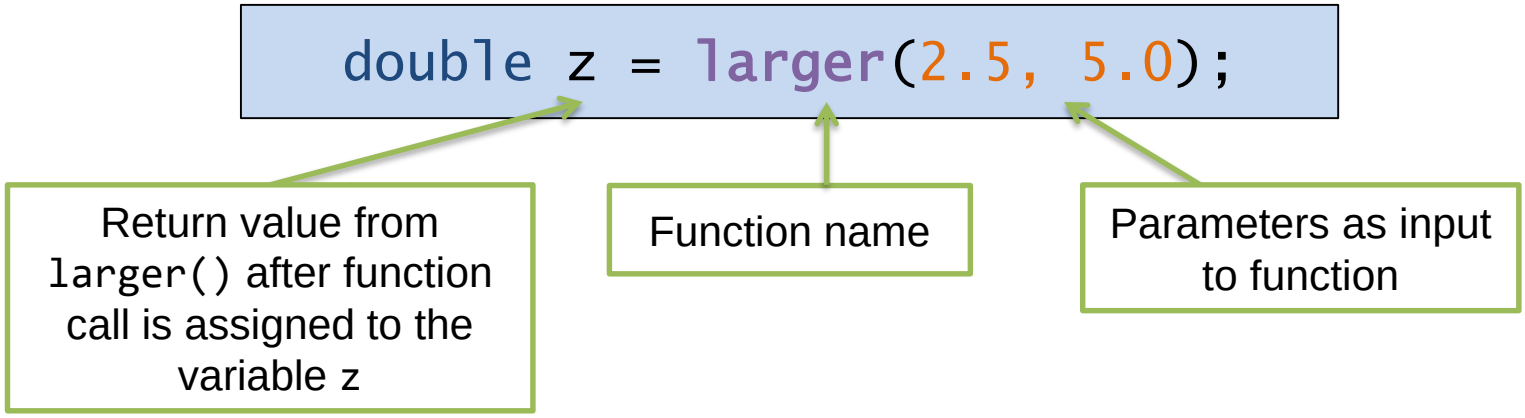
```
void print_msg(int x)
{
  cout << "This is a void function " << x << endl;
}
```

No return statement

Both are OK!

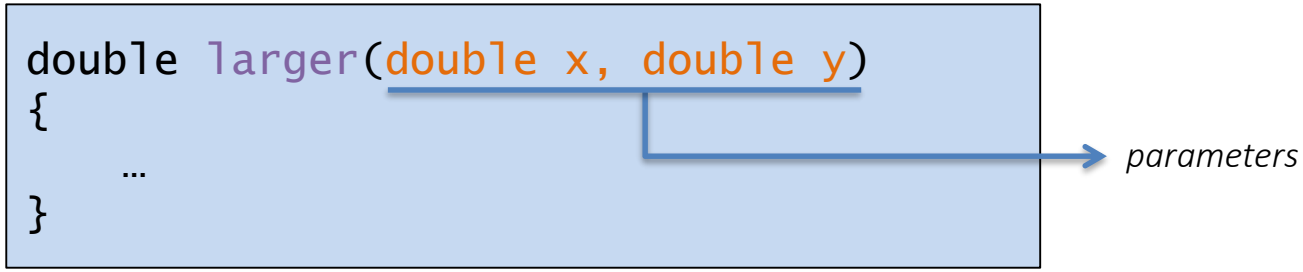
Function Call

- How to call (or invoke) a function?
- Think about how you use the pre-defined function sqrt()?
- A function call (i.e., the process of calling a function) is made using the function name with the necessary parameters
  - A function call is itself an expression, and can be put in any places where an expression is expected.
  - Example:

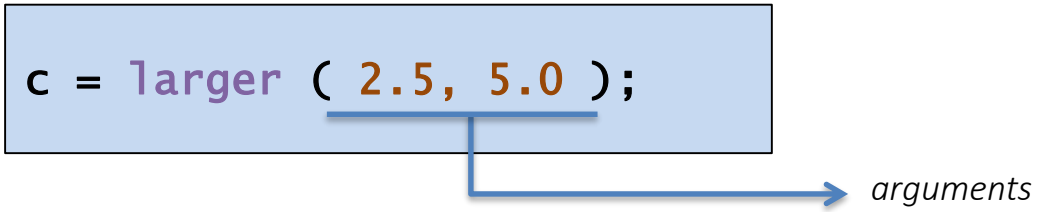


Function Call

- Parameters vs. arguments
  - The parameters used in the function definition are called formal parameters or simply parameters. They are placeholders in the function.



  - The actual values passed to a function in a function call are referred to as actual parameters or arguments. They are the actual values used in the execution of the function to produce the return value.



Function Call

- The arguments used in a function call can be constants, variables, expressions, or even function calls, e.g.,

```
double z1 = larger (2.5, 5.0);                  // constants
double z2 = larger (one, two);                  // variables
double z3 = larger (one - 2, two);              // expressions
double z4 = larger (2.5, larger (3, 5.0) );  // a function
```

- In using expressions as arguments, the expressions will be evaluated to produce a value before the function call is made.
- Since a function call is also an expression, the mechanism of using function calls as arguments is identical to that of using expressions.

Function Declaration

The compiler needs to know about the function prototype (i.e., its name, input parameters, return type) before a function can be used. We can declare a function before defining it.

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

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
    return y;
}

int main()
{
    ….
    c = larger(a, b);
    ….
}
```

One way to do this is to place the function definition before the function call in the source file.

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

double larger(double x, double y);

int main()
{
  ….
  c = larger(a, b);
  ….
}

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
    return y;
}
```

Note the ; here. It is needed since this function declaration is a statement. Compare this with the function header in the example on the left.

Alternatively, the function definition can be placed anywhere in the source file by including a function declaration before the function call.

Function Declaration

- A function declaration is similar to a function header except that it must be followed by a semicolon; and the identifiers in the parameter list can be changed or even omitted. It provides all the information needed in making a function call.

Syntax

```
type_ret func_name(type1 par1, type2 par2, …);
```

or

```
type_ret func_name(type1, type2, …);
```

Function Declaration

Examples:

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

double larger(double p, double q) ;

int main()
{
  ….
  c= larger(a, b);
  ….
}

double larger(double x, double y)
{
  return (x >= y)? x : y;
}
```

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

double larger(double, double) ;

int main()
{
  ….
  c = larger(a, b);
  ….
}

double larger(double x, double y)
{
  return (x >= y)? x : y;
}
```

Can you guess the meaning of `(x >= y)? x : y`?
Hint: It is a value but the value depends on a condition.

FLOW OF CONTROL

Flow of Control

- When a program is put into execution
  - It always starts at the main function no matter where its definition is in the source file.
  - The statements in the main function are executed sequentially from top to bottom and the control is passed from one statement to another.
  - When a function call is encountered, the execution of the current function is suspended.
    - The values of the arguments are copied to the formal parameters of the called function, and the control is passed to the called function.
    - Likewise, the statements in the called function are executed from top to bottom, and the control is passed from one statement to another.
    - When a return statement is encountered, the execution of the function terminates.
    - The control is passed back to the calling function together with the return value.
  - The main function will resume at the calling statement.
  - When a return statement in the main function is encountered, the program ends.

Flow of Control

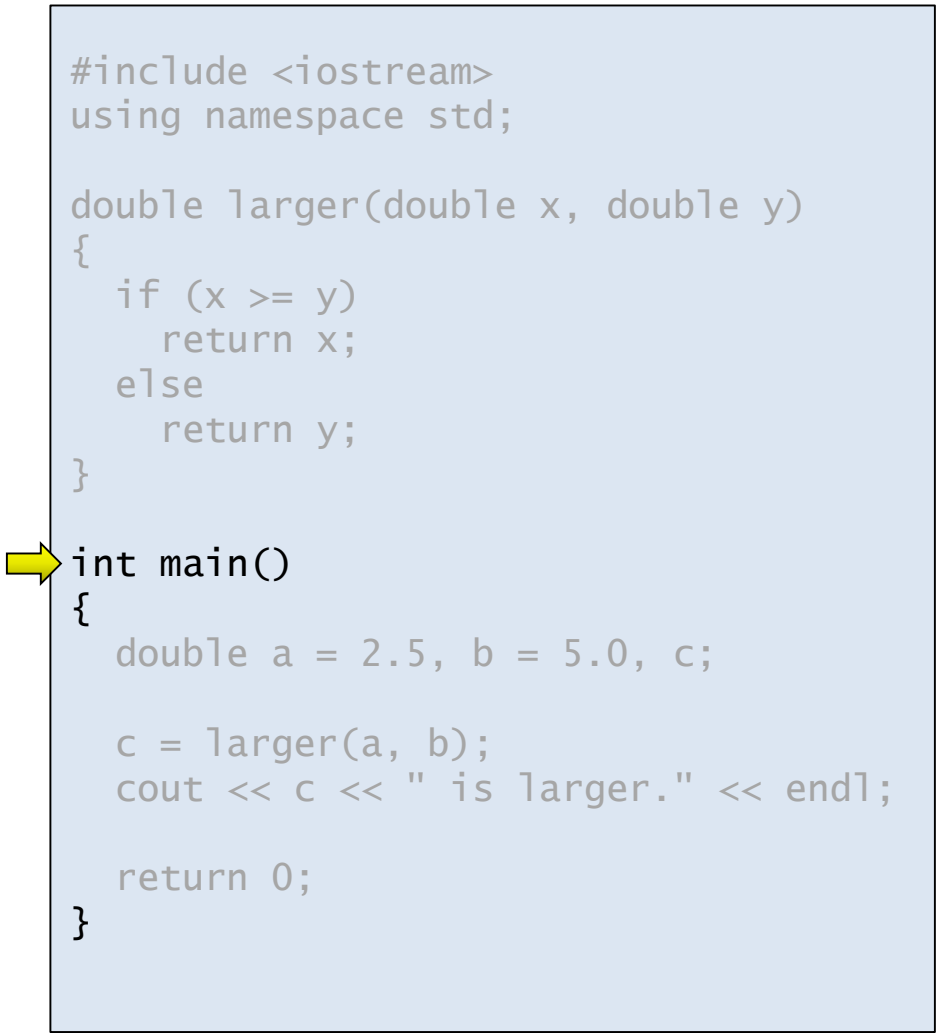
The program on the right consists of two functions:

- main(): controls general logic flow and handles I/O
- larger(): determines the larger of two numbers

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

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
    return y;
}

int main()
{
  double a = 2.5, b = 5.0, c;

  c = larger(a, b);
  cout << c << " is larger." << endl;

  return 0;
}
```

Flow of Control

- When a program is put into execution, it always starts at the main function no matter where its definition is in the source file.



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

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
    return y;
}

int main()
{
  double a = 2.5, b = 5.0, c;

  c = larger(a, b);
  cout << c << " is larger." << endl;

  return 0;
}
```

Flow of Control

- The statements in the main function are executed sequentially from top to bottom.
- The control is passed from one statement to another.

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

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
   return y;
}

int main()
{
  double a = 2.5, b = 5.0, c;

  c = larger(a, b);
  cout << c << " is larger." << endl;

  return 0;
}
```

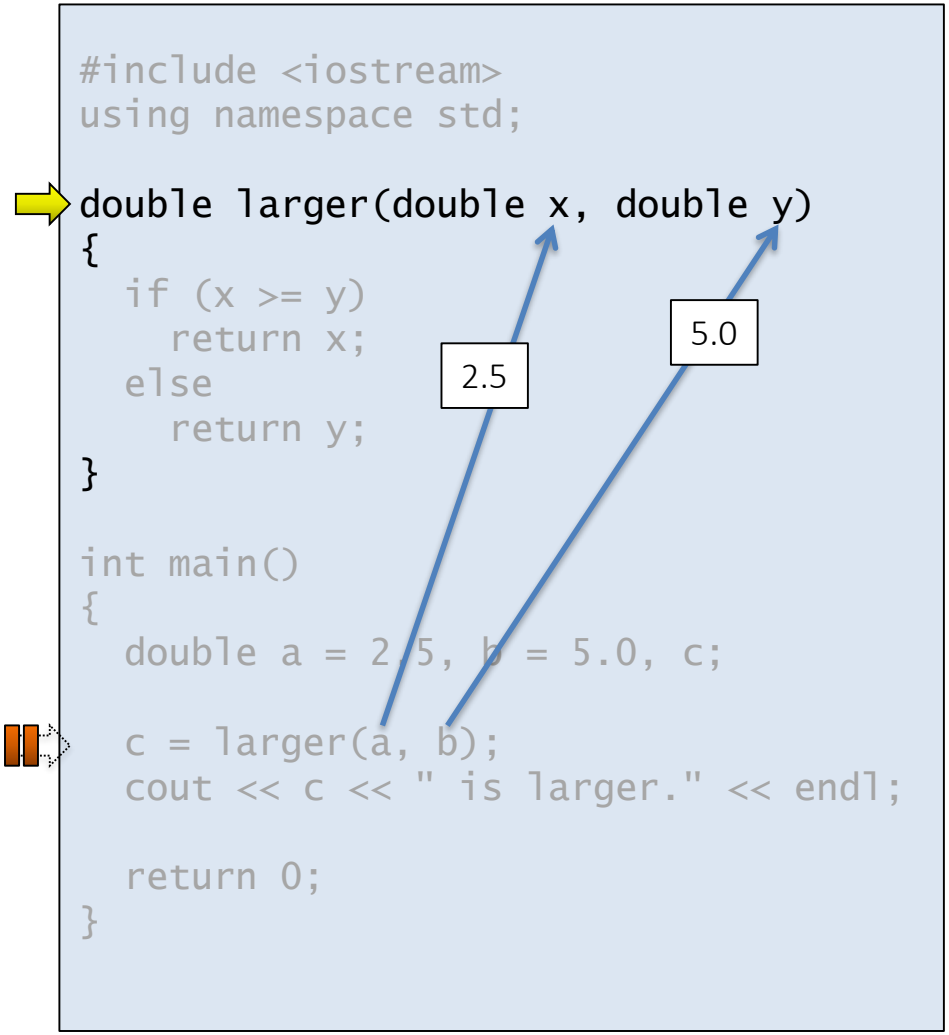
Flow of Control

- When a function call is encountered, the execution of the current function is suspended.

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

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
    return y;
}

int main()
{
  double a = 2.5, b = 5.0, c;

  c = larger(a, b);
  cout << c << " is larger." << endl;

  return 0;
}
```

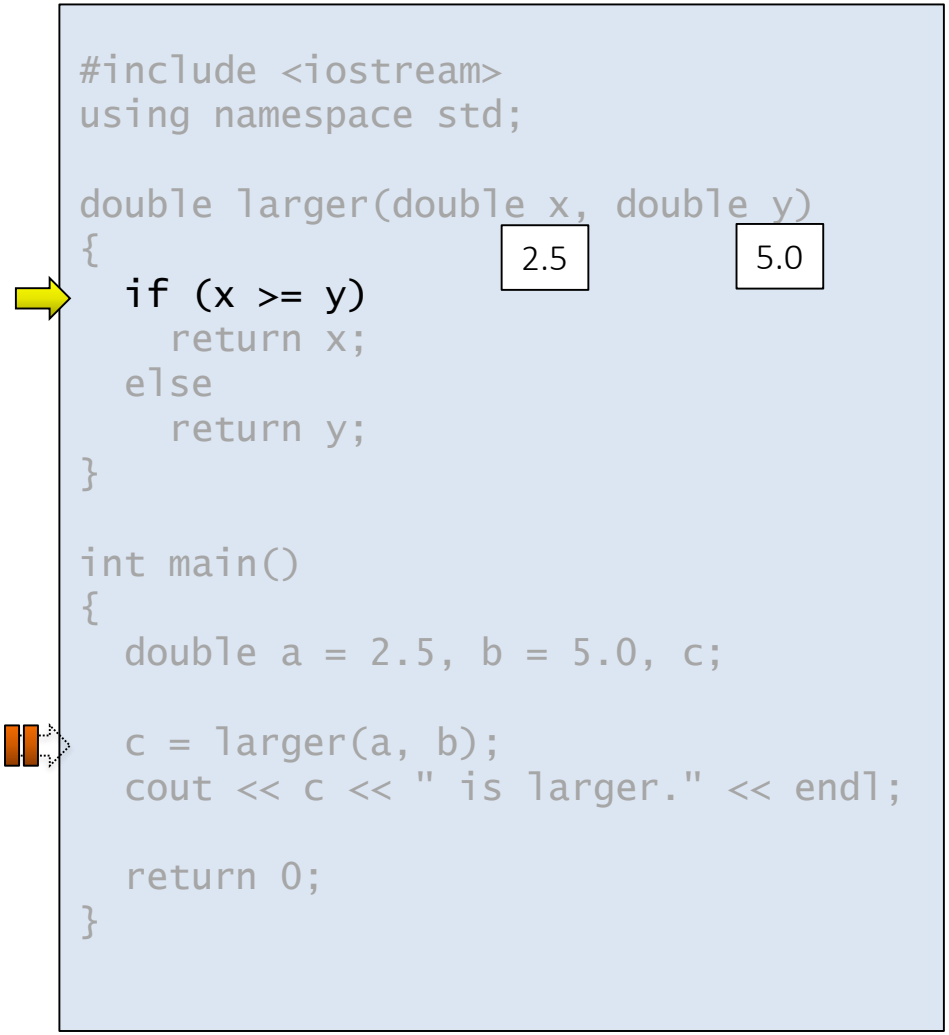
Flow of Control

- The values of the arguments are copied to the formal parameters of the called function.
- The control is passed back to the called function.



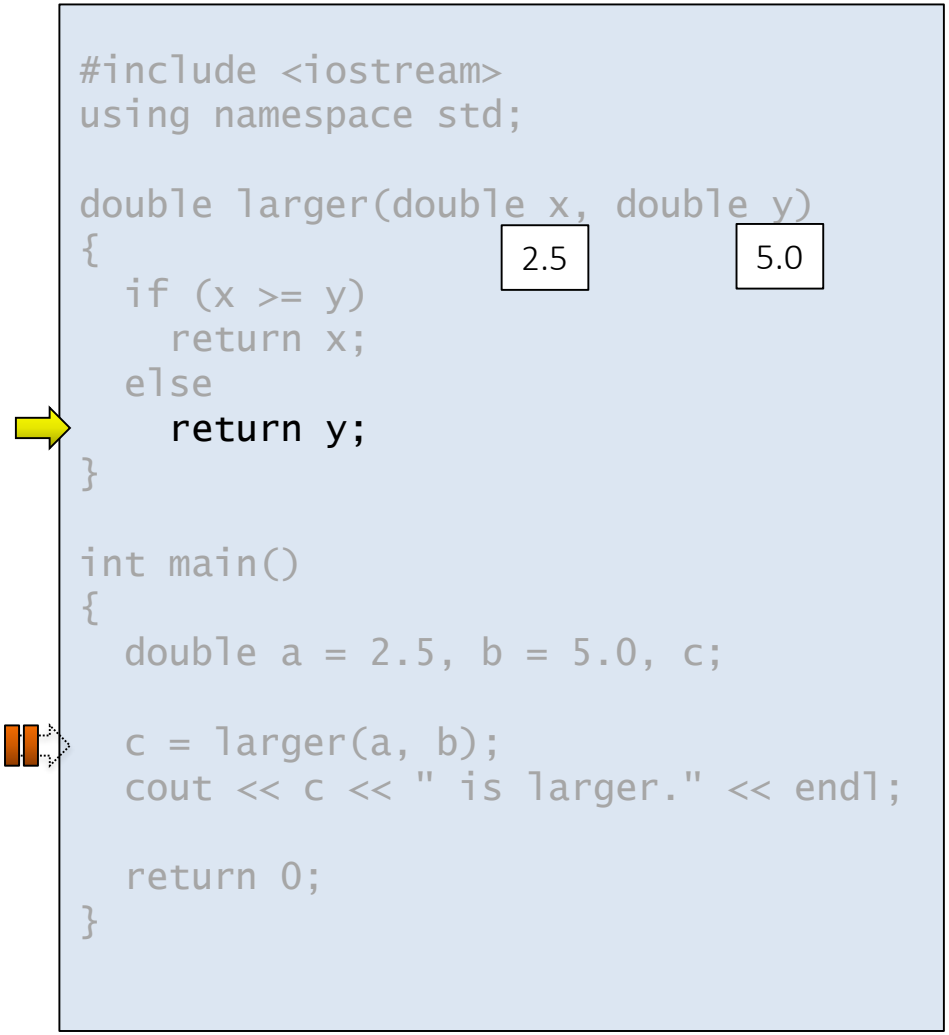
Flow of Control

- Likewise, the statements in the called function are executed from top to bottom
- The control is passed from one statement to another



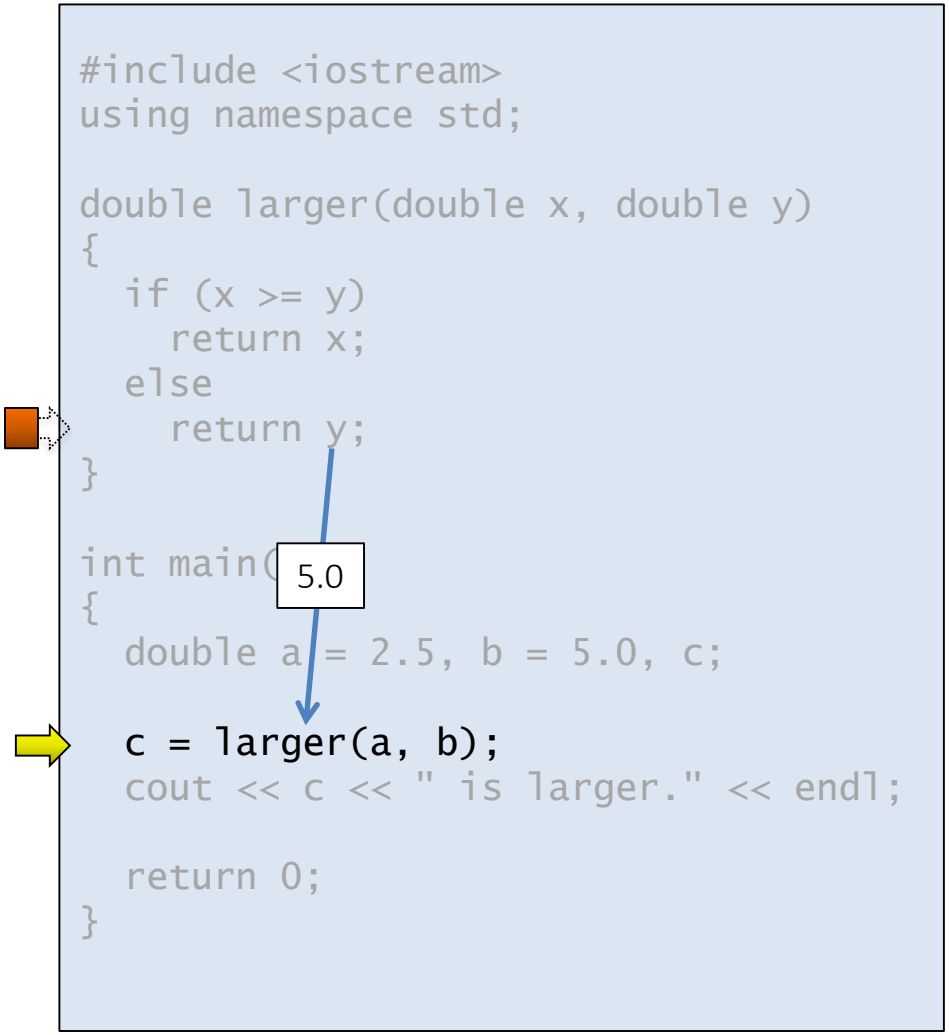
Flow of Control

- When a return statement is encountered, the execution of the function terminates



Flow of Control

- The control is passed back to the calling function together with the return value



Flow of Control

- The main function will resume at the calling statement

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

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
    return y;
}

int main()
{
  double a = 2.5, b = 5.0, c;

  c = larger(a, b);
  cout << c << " is larger." << endl;

  return 0;
}
```

Flow of Control

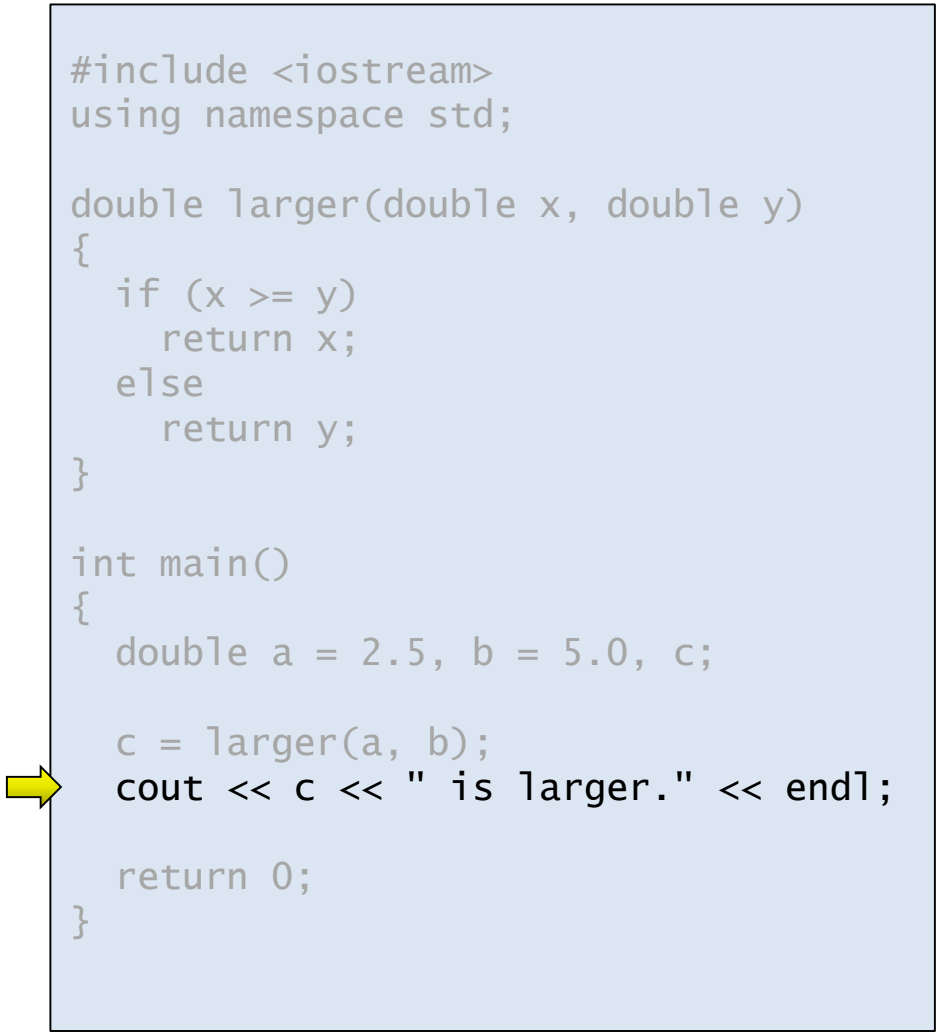
- The statements in the main function are executed sequentially from top to bottom.
- The control is passed from one statement to another.

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

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
    return y;
}

int main()
{
  double a = 2.5, b = 5.0, c;

  c = larger(a, b);
  cout << c << " is larger." << endl;

  return 0;
}
```

c takes the value 5.0 which is the return value of `larger()`

Flow of Control

- The statements in the main function are executed sequentially from top to bottom.
- The control is passed from one statement to another.



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

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
    return y;
}

int main()
{
  double a = 2.5, b = 5.0, c;

  c = larger(a, b);
  cout << c << " is larger." << endl;

  return 0;
}
```

Flow of Control

- When a return statement in the main function is encountered, the program ends.
- The control is passed back to the operating system.

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

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
    return y;
}

int main()
{
  double a = 2.5, b = 5.0, c;

  c = larger(a, b);
  cout << c << " is larger." << endl;

  return 0;
}
```

Think about this: the main body is also a function main(), it is called by the operating system when you run the program.

SCOPE OF VARIABLES

Local Variables

- Variables declared within a function, including formal parameters, are private or local to that particular function, i.e., no other function can have direct access to them.
- Local variables in a function come into existence only when the function is called, and disappear when the function is exited.
  - Do not retain their values from one function call to another
  - Their values must be explicitly set upon each entry
- Local variables declared within the same function must have unique identifiers, whereas local variables of different functions may use the same identifier.

Local Variables

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

double larger(double x, double y)
{
    double max;
    max = (x >= y)? x : y;

    return max;
}

int main()
{
    double a = 2.5, b = 5.0, max;

    max = larger(a, b);
    cout << max << " is larger." << endl;

    return 0;
}
```

local variables of **larger()**:
x, y, max
i.e., these variables are input parameters or variables defined in the function larger(), and therefore can only be seen or used in larger()

local variables of **main()**:
a, b, max
i.e., these variables are defined in the function main(), and therefore can only be seen or used in main()

The local variables **max** of **larger()** and **max** of **main()** are **unrelated**.

Local Variables

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

double larger(double x, double y)
{
   // double max;
   max = (x >= y)? x : y;

   return max;
}

int main()
{
   double a = 2.5, b = 5.0, max;

   max = larger(a, b);
   cout << max << " is larger." << endl;

   return 0;
}
```

There will be a compilation error if we comment out the declaration of **max** in **larger()** because **max** in **main()** is a local variable of **main()** and cannot be seen or used in larger().

Global Variables

- Variables may also be declared outside all functions.
- Such variables are called **global variables** because they can be accessed by all functions, i.e., globally accessible within the file containing the program.
- Global variables remain in existence permanently.
  - Retain their values even after the functions that set their values have returned and exited
  - Can be used instead of arguments to communicate data between functions, **however**:
    - The values of global variables can be changed by any functions.
    - Hard to trace, especially when something goes wrong
    - Not recommended and should be avoided!
- Frequently used as **declared constant** (whose values cannot be changed)

Global Variables

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

double a, b;
const double PI = 3.1415;

double larger()
{
   return (a >= b)? a : b;
}

int main()
{
   cout << "Input two integers: ";
   cin >> a >> b;
   cout << larger() << " is larger." << endl;

   double r;
   cout << "Input radius of a circle: ";
   cin >> r;
   cout << "Area of circle = " << PI * r * r << endl;
   return 0;
}
```

global variables:
a, b, PI

The global constant **PI** can be used throughout the file after its declaration.

Avoid using global variables to communicate data between functions

The variables **a, b** should best be changed into input parameters for the function larger(). Can you do that?

Scopes of Variables

- The scope of a variable is the portion of a program that the variable is well-defined and can be used.
- A variable cannot be accessed beyond its scope.
- The scope of a local / global variable starts from its declaration up to the end of the block / file.
  - A block is delimited by a pair of braces { }.
  - Variables declared in outer blocks can be referred to in an inner block.
- Variables can be declared with the same identifier as long as they have different scopes.
  - Variables in an inner block will hide any identically named variables in outer blocks.

Scopes of Variables

```
double a;
int func(int x, int y)
{
    …
    if (x > y)
    {
        int k;
        …
    }
    int z;
    …
}

double b;
int main()
{
    int x, y, z;
    …
    if (…)
    {
        int x;
        …
    }
    …
}
```

Scope of global variable **a**:
from declaration to end of block
(in this case, end of file; hence scope of **a** is the entire file)

Scope of formal parameters **x, y**:
entire function

Scope of local variable **k**:
from declaration to end of block
(in this case, end of if statement)

Scope of local variable **z**:
from declaration to end of block
(in this case, end of func)

Scopes of Variables

```
double a;
int func(int x, int y)
{
    …
    if (x > y)
    {
        int k;
        …
    }
    int z;
    …
}

double b;
int main()
{
    int x, y, z;
    …
    if (…)
    {
        int x;
        …
    }
    …
}
```

Scope of global variable **b**:
from declaration to end of block
(in this case, end of file)

Scope of local variables **x, y, z**:
from declaration to end of block
(in this case, end of main function)

Scope of local variable **x** in the inner block:
from declaration to end of block
(in this case, end of if statement)

the outer x is hidden within this block

Scopes of Variables

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

int main()
{
   int i = 0;
1  cout << "Outer block: i = " << i << endl;

   {
      int i = 100;
2     cout << "Inner block: i = " << i << endl;
   }

3  cout << "Outer block: i = " << i << endl;
   return 0;
}
```

Screen output

```
Outer block: i = 0
Inner block: i = 100
Outer block: i = 0
```

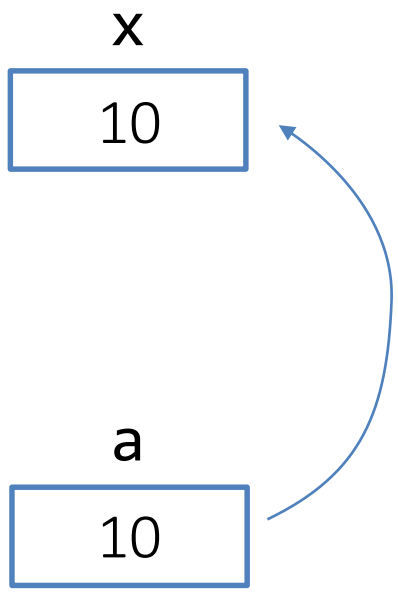
PARAMETER PASSING MECHANISM

Pass-by-Value

- When a function call takes place, the values of the arguments are copied to the formal parameters of the function.
- This mechanism of parameter-passing is known as pass-by-value.
- Recall that formal parameters are local variables.
  - Any changes made to their values are local to the function and will not alter the arguments in the calling function.
  - These variables will disappear when the function exits, only the return value of the function will be passed back to the calling function.

Pass-by-Value

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

// computes the square of an integer
void square( int x )
{
   x *= x;
}

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

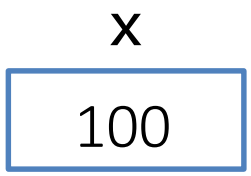


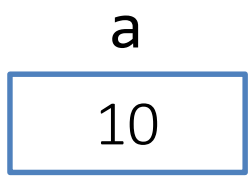
x is a parameter and also a local variable of the square function

Copying of value of actual argument to formal parameter

Pass-by-Value

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

// computes the square of an integer
void square( int x )
{
   x *= x;
}

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

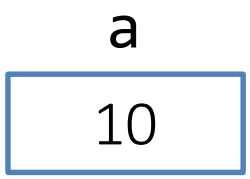




Pass-by-Value

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

// computes the square of an integer
void square( int x )
{
   x *= x;
}

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

Variable **x** disappears (more precisely, the memory location it occupies is released back to the system) upon function completion.



Pass-by-Value

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

void swap(int a, int b)
{
2   cout << "a = " << a << ", b = " << b << endl;
    int temp = a;
    a = b;
    b = temp;
3   cout << "a = " << a << ", b = " << b << endl;
}

int main()
{
    int x = 0, y = 100;
1   cout << "x = " << x << ", y = " << y << endl;
    swap(x, y);
4   cout << "x = " << x << ", y = " << y << endl;
    return 0;
}
```

Suppose we want to swap the values of the variables x and y using the function swap(), what will happen in this program?

Screen output

```
x = 0, y = 100
a = 0, b = 100
a = 100, b = 0
x = 0, y = 100
```

It doesn't work!

Because the variables x and y are passed to swap() using pass-by-value, only the values are transferred to swap(), and swap() can only deal with its local variables a and b.

Pass-by-Reference

- In order to allow a function to modify the arguments (variables) in the calling function, another parameter-passing mechanism known as **pass-by-reference** should be used.
- In pass-by-reference
  - The formal parameters will refer to the same memory cells of the arguments in run-time, and therefore the arguments must be variables.
  - Any changes made to the values of the formal parameters will be reflected in the arguments as they share the same memory cells.

Pass-by-Reference

- To indicate a formal parameter will be passed by reference, an ampersand sign **&** is placed in front of its identifier in the function header and function declaration.

Syntax (function header)

```
type_ret  func_name(type1 &par1, type2 &par2, …)
```

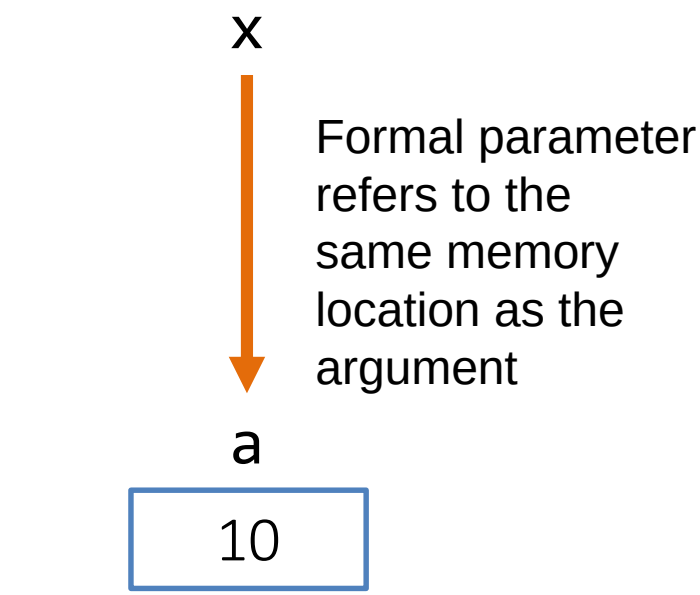
Syntax (function declaration)

```
type_ret  func_name(type1 &par1, type2 &par2, …);
```

Pass-by-Reference

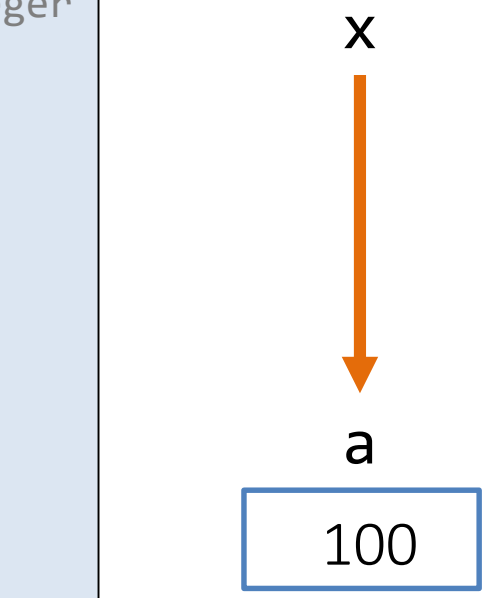
```
#include <iostream>
using namespace std;

// computes the square of an integer
void square( int &x )
{
   x *= x;
}

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

Note the **&** to indicate that the formal parameter x is pass-by-reference.



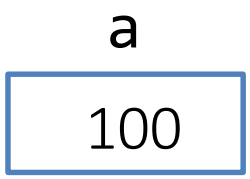
Pass-by-Reference

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

// computes the square of an integer
void square( int &x )
{
   x *= x;
}

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



Pass-by-Reference

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

// computes the square of an integer
void square( int &x )
{
   x *= x;
}

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



Pass-by-Reference

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

void swap(int &a, int &b)
{
2   cout << "a = " << a << ", b = " << b << endl;
    int temp = a;
    a = b;
    b = temp;
3   cout << "a = " << a << ", b = " << b << endl;
}

int main()
{
    int x = 0, y = 100;
1   cout << "x = " << x << ", y = " << y << endl;
    swap(x, y);
4   cout << "x = " << x << ", y = " << y << endl;
    return 0;
}
```

What happens if we use pass-by-reference for the swap function?

Screen output

```
x = 0, y = 100
a = 0, b = 100
a = 100, b = 0
x = 100, y = 0
```

The formal parameters a and b in swap() refer to the memory locations of the arguments x and y, respectively.

Pass-by-Reference vs. Value-Returning Function

- Call by Reference: modify the values of the actual parameters in the calling function
- Value-Returning Function: returning a value that can be used by the calling function

Call by Value

```
int squareByValue( int number )
{
      return number *= number;
}
```

Caller's argument not modified, return result by **return value**

Call by Reference

```
void squareByReference( int &number )
{
      number *= number;
}
```

Caller's argument modified, result stores in the **reference parameter**

Pass-by-Reference vs. Value-Returning Function

```
int squareByValue( int );
void squareByReference( int & );

int main()
{
     int x = 2;
     int z = 4;

     cout << "x = " << x << " before squareByValue\n";
     cout << "Value returned by squareByValue: "
          << squareByValue( x ) << endl;
     cout << "x = " << x << " after squareByValue\n" << endl;

     cout << "z = " << z << " before squareByReference" << endl;
     squareByReference( z );
     cout << "z = " << z << " after squareByReference" << endl;

     return 0;
}
```

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

Screen output

Return value of squareByValue() is used by the cout expresssion.

Result of computation by squareByReference() is updated in z.

Pass-by-Reference vs. Value-Returning Function

- Good programming style:
  - If a function needs to return more than one value, use a void function with reference parameters to return the values

```
const double CONVERSION = 2.54;
const int INCHES_IN_FOOT = 12;
const int CENTIMETERS_IN_METER = 100;

void metersAndCentTofeetAndInches(int mt, int ct, int& f, int& in)
{
     int centimeters;
     centimeters = mt * CENTIMETERS_IN_METER + ct;
     in = (int) (centimeters / CONVERSION);
     f = in / INCHES_IN_FOOT;
     in = in % INCHES_IN_FOOT;
}
```

f and in are the computation results.
Think about how the calling functions can call this function and access the results through the arguments after function call.

Quick Exercise 1

What's the output of the following program?

Try to dry run (i.e., trace manually without using the computer to run) the program to obtain the result. Then run it on your computer to check the result.

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

void figureMeOut(int &x, int y, int &z) {
    cout << x << ' ' << y << ' ' << z << endl;
    x = 1;
    y = 2;
    z = 3;
    cout << x << ' ' << y << ' ' << z << endl;
}

int main() {
    int a=10, b=20, c=30;
    figureMeOut(a, b, c);
    cout << a << ' ' << b << ' ' << c << endl;
}
```

Answer to Quick Exercise 1

Screen output:

```
10 20 30
1 2 3
1 20 3
```

We are happy to help you!

"If you face any problems in understanding the materials,
please feel free to contact me, our TAs or student TAs.
We are very happy to help you!
We wish you enjoy learning programming in this class ☺."