FUNDAMENTALS OF STRUCTURED PROGRAMMING

Lecture 7

Functions II (User-Defined)

Course Coordinator: Prof. Zaki Taha Fayed

Presented by: Dr. Sally Saad

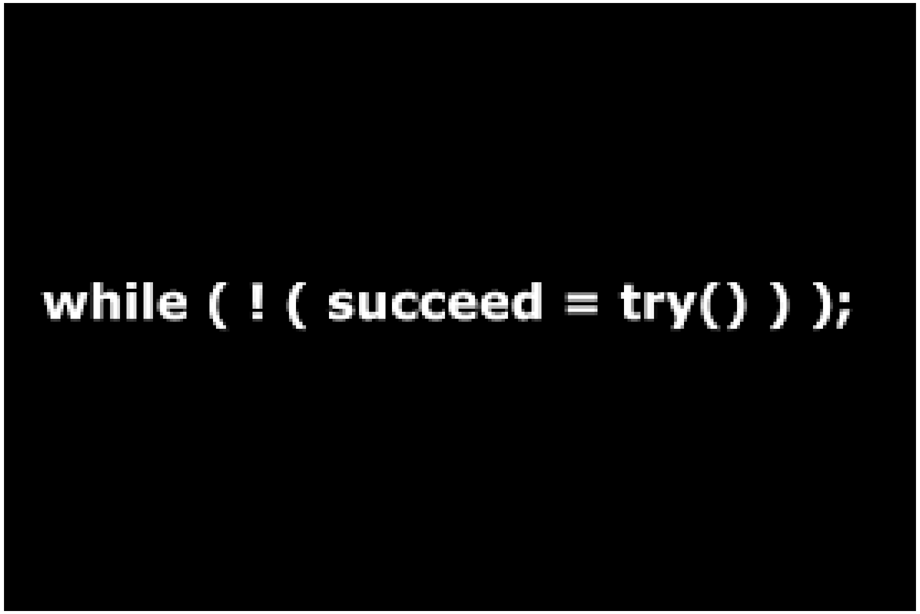
SallySaad@gmail.com

DropBox folder link

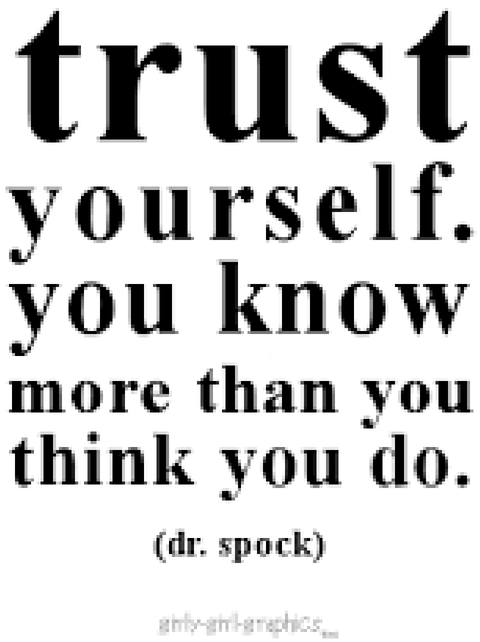
https://www.dropbox.com/sh/85vnrgkfqgrzhwn/AABdwKLJZqZs26a7u-y0AFwia?dl=0

Credits to Dr.Salma Hamdy for content preparation

Quotes of the Day!



```
while ( ! ( succeed = try() ) );
```



trust yourself. you know more than you think you do.

(dr. spock)

AGENDA

- Top Down Design
- User/Programmer-Defined functions
- Example
- Bonus Task (Back Again to Employees Project ☺)
- Projects Startup and Delivery
- Summary to functions.
- File Stream (Self-Study for projects)

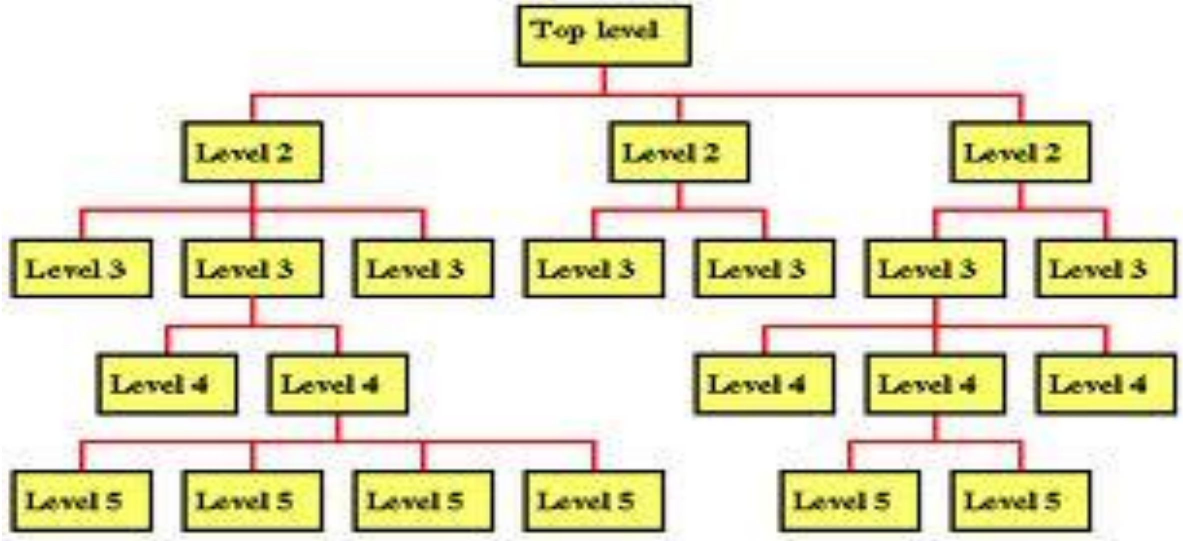
1. Top-down Design

- If we look at a big problem as a whole, it may seem hard to solve because it is so complex.

- Complex problems can be solved using top-down design, where:
  - We break the problem into parts.
  - Then break the parts into parts.
  - Soon, each of the parts will be easy to do.
- Also known as **stepwise** design.

1. Top-down Design – (cont.)

- Also called step-wise refinement.
- A methodology of information processing, problem solving, and knowledge representation, that starts at the highest level of a design concept and proceeds towards the lowest level.



1. Top-down Design – (cont.)

- Start with the broad project specification in mind and put that information in a centralized location.
- Then progress from this information to the individual parts.

1. Top-down Design – (cont.)

Example

- Problem: we want to automate sending letters for students who have passed their absence limits.

1. Top-down Design – (cont.)

Example

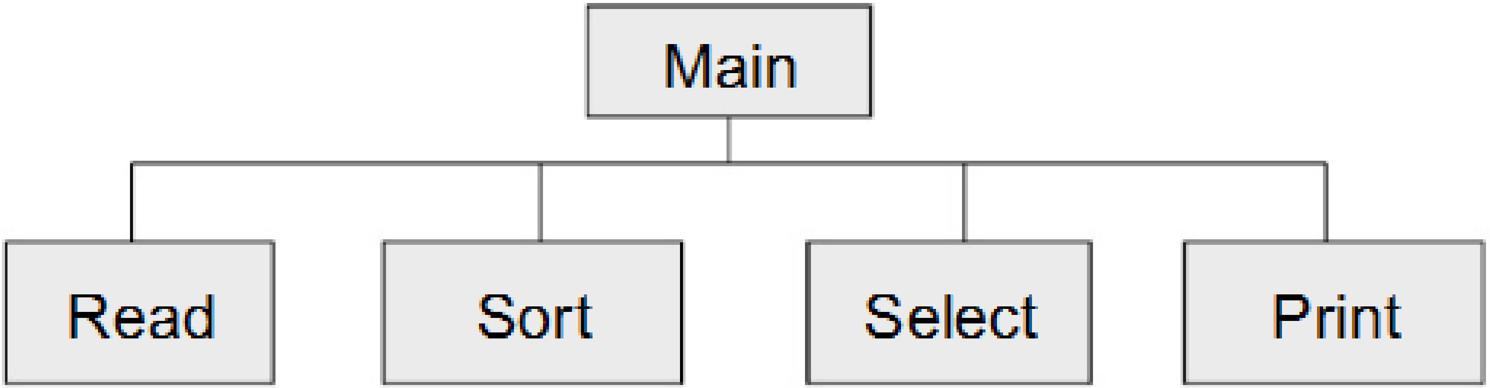
- **Possible solution:**

1 Get students list from files

2 Sort according to absence times

3 Select those with more than 3 times

4 Print a letter for each student



1. Top-down Design – (cont.)

Advantages

- Breaking the problem into parts helps us to clarify what needs to be done.
- At each step of refinement, the new parts become less complicated and, therefore, easier to figure out.
- Parts of the solution may turn out to be reusable.
- Breaking the problem into parts allows more than one person to work on the solution.

1. Top-down Design – (cont.)

Disadvantages

- Not enough details because of abstraction.
- Too many break downs.
- When to stop?

1. Top-down Design – (cont.)

- Should any of these steps be broken down further? Possibly.
- How do I know? Ask yourself whether or not you could easily write the algorithm for the step. If not, break it down again.
- When you are comfortable with the breakdown, write the pseudo code for each of the steps (modules) in the hierarchy.
- Typically, each module will be coded as a separate function.

1. Top-down Design – (cont.)

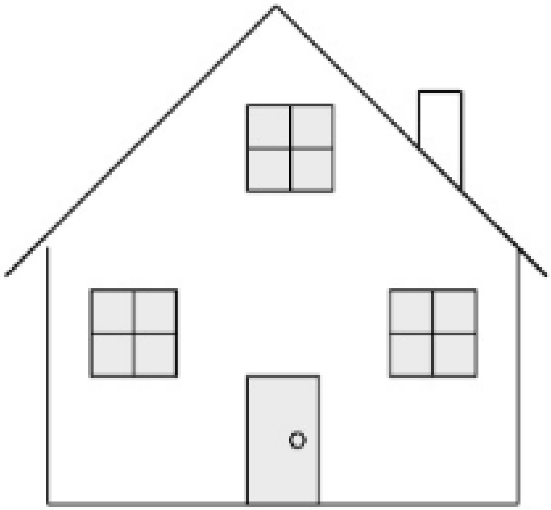
- Experience has shown that the best way to develop and maintain a large program is to construct it from small, simple pieces, or components → functions.

- This technique is called divide and conquer.

1. Top-down Design – (cont.)

Example

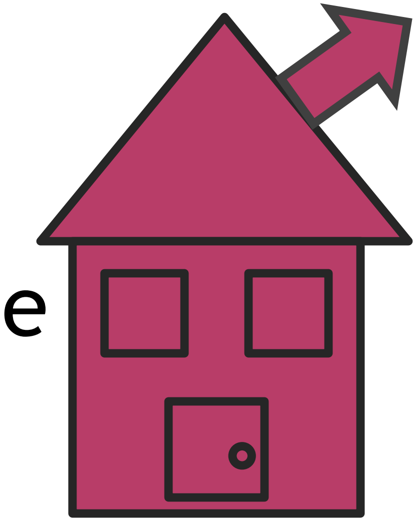
- Problem: Write a program that draws this picture of a house.



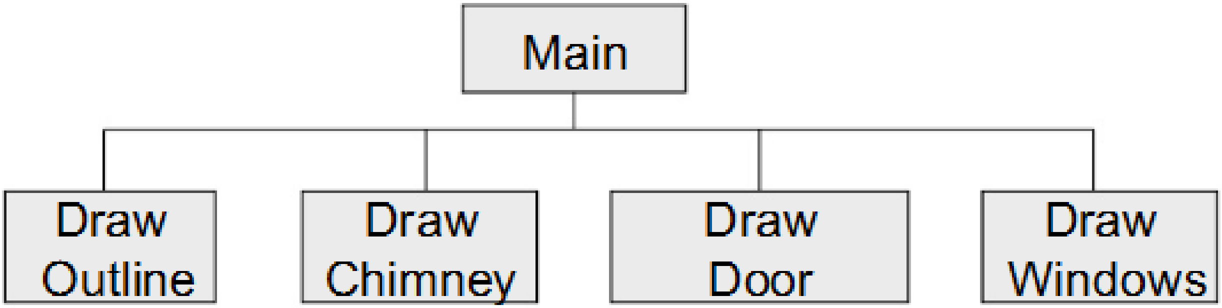
1. Top-down Design – (cont.)

Example

- **Possible solution:**



1. Draw the outline of the house
2. Draw the chimney
3. Draw the door
4. Draw the windows



1. Top-down Design – (cont.)

Example

- **Possible solution:**

1 Draw the outline of the house

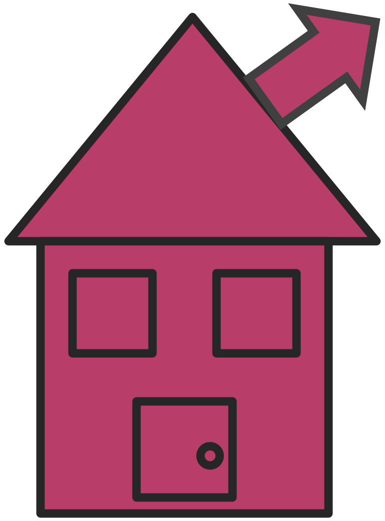
2 Draw the chimney

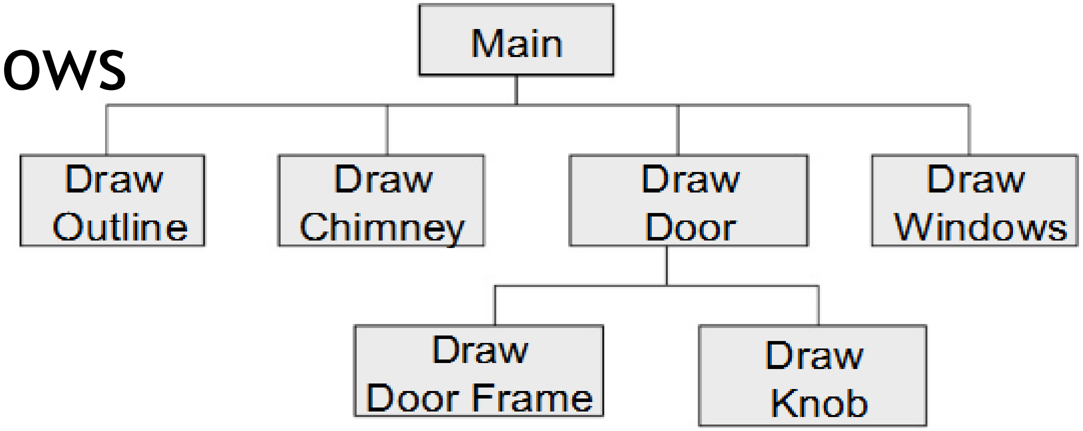
3 **Draw the door**

Call Draw Door Frame

Call Draw Knob

4 Draw the windows





1. Top-down Design – (cont.)

Example

- **Possible solution:**

1 Draw the outline of the house

2 Draw the chimney

3 <u>Draw the door</u>

Call Draw Door Frame

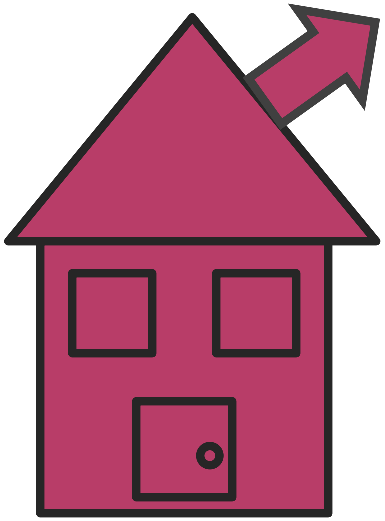
Call Draw Knob

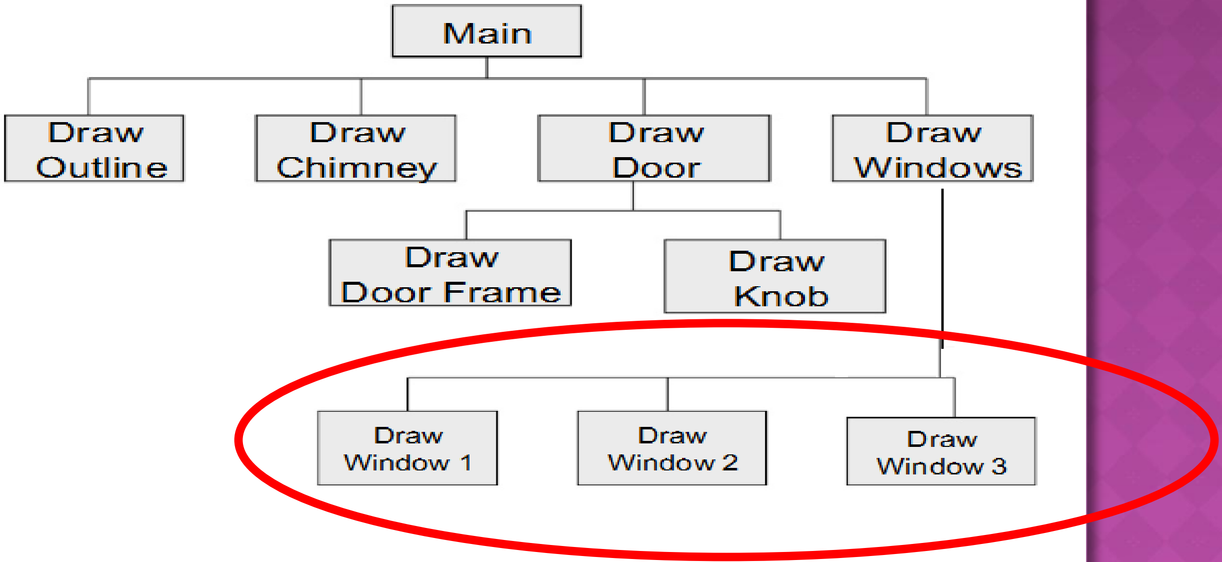
4 <u>Draw the windows</u>

Draw Window 1

Draw Window 2

Draw Window 3





Main

Draw Outline

Draw Chimney

Draw Door

Draw Windows

Draw Door Frame

Draw Knob

Draw Window 1

Draw Window 2

Draw Window 3

1. Top-down Design – (cont.)

Example

- **Possible solution:**

1 Draw the outline of the house

2 Draw the chimney

3 Draw the door

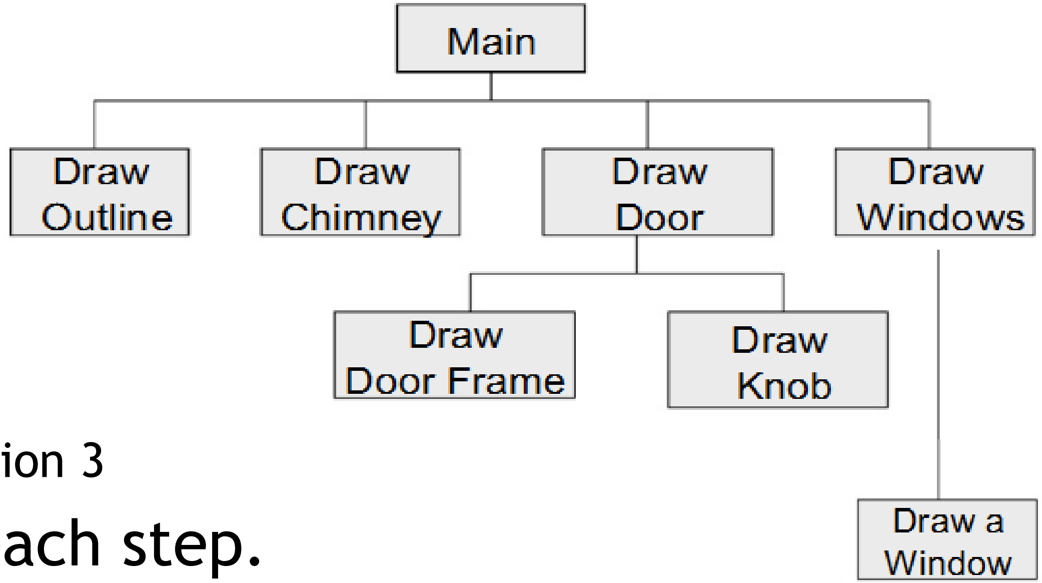
4 **<u>Draw the windows</u>**

Draw a Window in Locat

Draw a Window in Locat

Draw a Window in Location 3

5 Code/get code for each step.



Code reusability

1. Top-down Design – (cont.)

- We will use top-down design for all remaining programming projects.
- This is the standard way of writing programs.
- Programs produced using this method and using the three kinds of control structures, sequential, selection and repetition, are called structured programs.
- Structured programs are easier to test, modify, and are also easier for other programmers to understand.

1. Top-down Design – (cont.)

- Building blocks of programs: terminology in other languages:
  - Procedures, subprograms, methods
  - In C++: functions
- I-P-O
  - Input - Processing - Output
  - Sometimes a functions has no input.
  - Sometimes a functions returns no output.

2. Programmer-defined Functions

- Three parts in using any function:
  - Function Declaration (prototype)
    - Information for compiler
    - To properly interpret calls
  - Function Definition (Body of the function)
    - Actual implementation (code) of what the function does; the function body.
  - Function Call
    - Using the function for its designed purpose.
    - Transfers control to function body
    - Arguments are plugged in for formal parameters

2. Programmer-defined Functions – (cont.)

(1) Function Declaration/prototype

- An "informational" declaration for compiler.
- Tells compiler how to interpret calls.

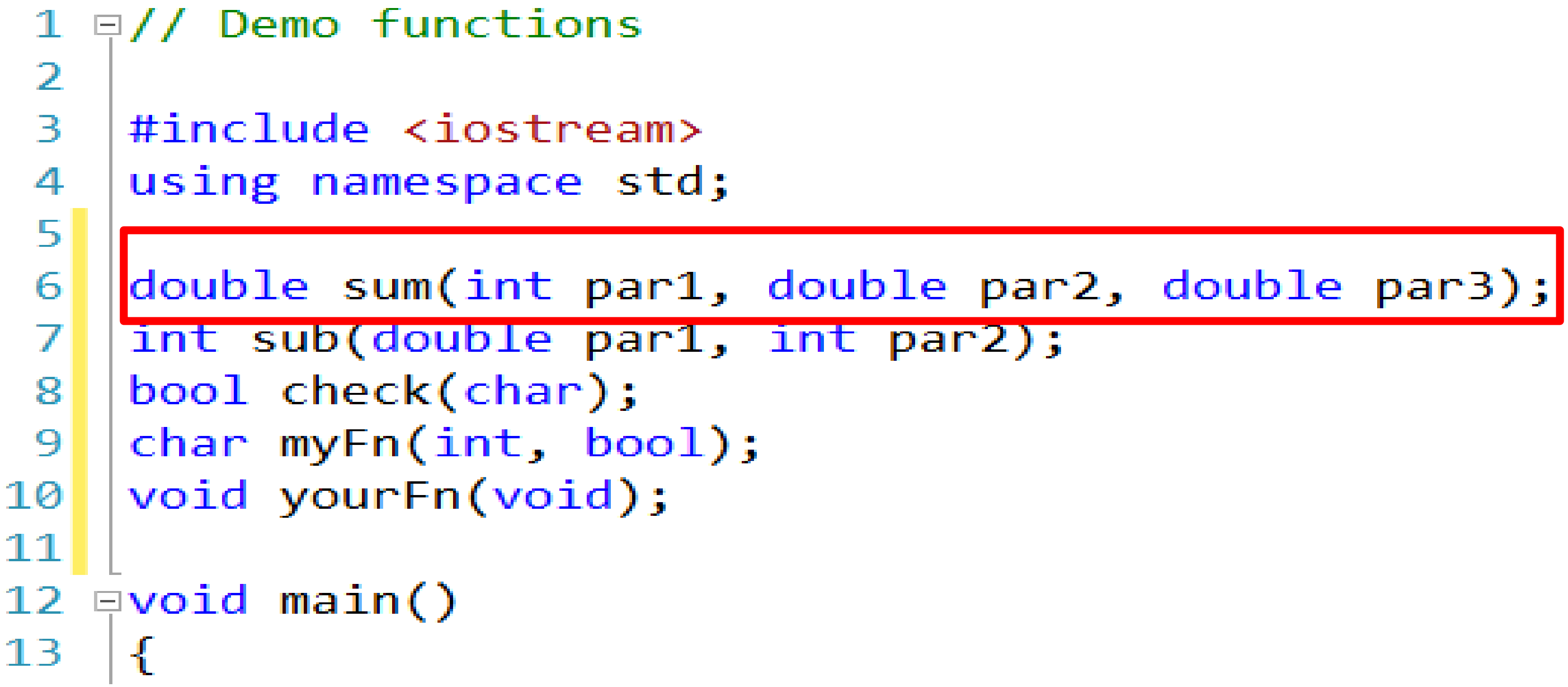
```
return_type Fn_name(parameter_list type);
```

- Formal parameters/attributes can be of any type and number.
- A function can return ONLY ONE value of any type or VOID.

2. Programmer-defined Functions – (cont.)

(1) Function Declaration/prototype (Passing by Value)

No need to mention Parameters names



```
// Demo functions

#include <iostream>
using namespace std;

double sum(int par1, double par2, double par3);
int sub(double par1, int par2);
bool check(char);
char myFn(int, bool);
void yourFn(void);

void main()
{
```

2. Programmer-defined Functions – (cont.)

(2) Function Definition (Body)

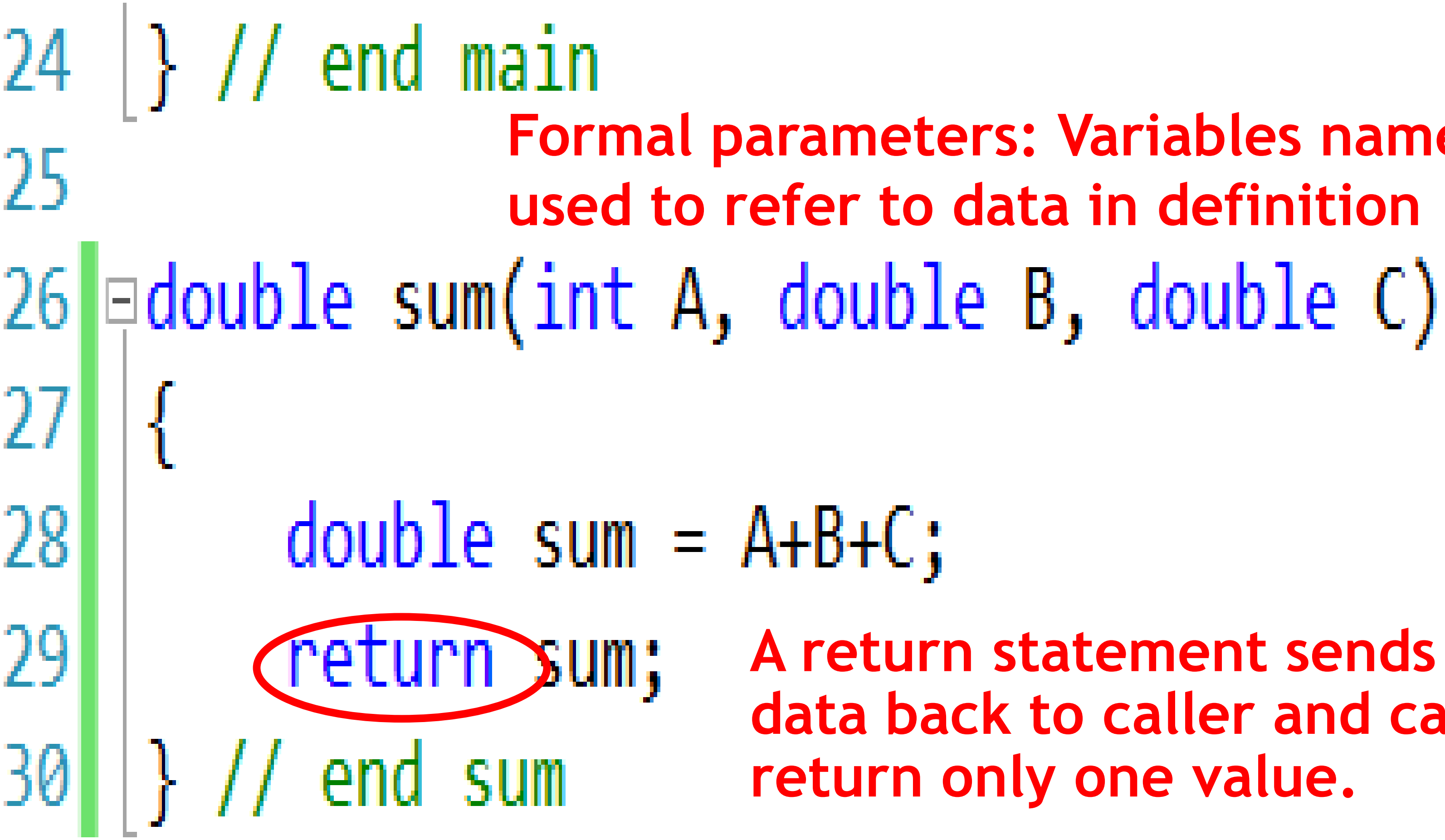
- Implementation of the function.
- Placed after main() or *instead of declaration*.
- Syntax:

```
return_type Fn_name(parameter_list)
{
    // your code goes here
}
```

- Returned value type, and formal parameters type and number should agree with the function declaration.

2. Programmer-defined Functions – (cont.)

(2) Function Definition



```
} // end main

double sum(int A, double B, double C)
{
    double sum = A+B+C;
    return sum;
} // end sum
```

Formal parameters: Variables names used to refer to data in definition

A return statement sends data back to caller and can return only one value.

2. Programmer-defined Functions – (cont.)

(3) Function Call

- To invoke the function and execute its body.
- Syntax:

```
Fn_name(arguments_list);
```

if the function returns nothing (void)

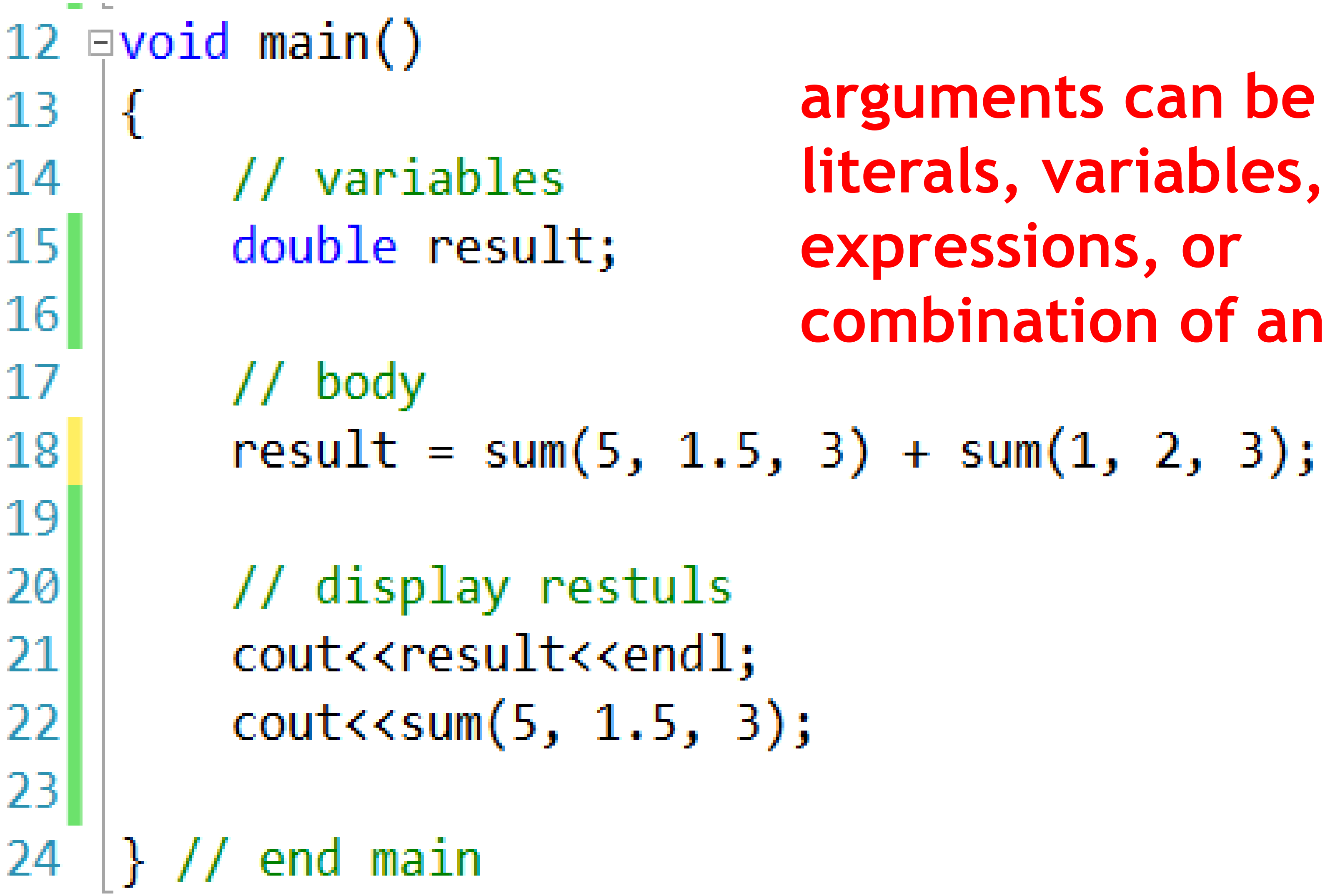
```
result = Fn_name(arguments_list);
```

if the function *returns A VALUE OF THE SAME TYPE AS RESULT*

- Arguments type and number should agree with the function's declaration.
- A function to execute must be called in a way or another in the main function.
- May a function call another???

2. Programmer-defined Functions – (cont.)

(3) Function Call



```
void main()
{
    // variables
    double result;

    // body
    result = sum(5, 1.5, 3) + sum(1, 2, 3);

    // display restuls
    cout<<result<<endl;
    cout<<sum(5, 1.5, 3);

} // end main
```

arguments can be literals, variables, expressions, or combination of any

2. Programmer-defined Functions – (cont.)

(4) Example 1 - void function

- A void function does not return a value.
- When being called, it cannot be used in expressions, or in assignment statements.
- Notice that a function's output is NOT always the same as return value.
- void welcome (void) / void welcome()
- void welcome (int)
- Lec7Ex1.cpp - comment on
  - void function with void parameters
  - void function with parameters
  - Parameter variables in declaration

2. Programmer-defined Functions – (cont.)

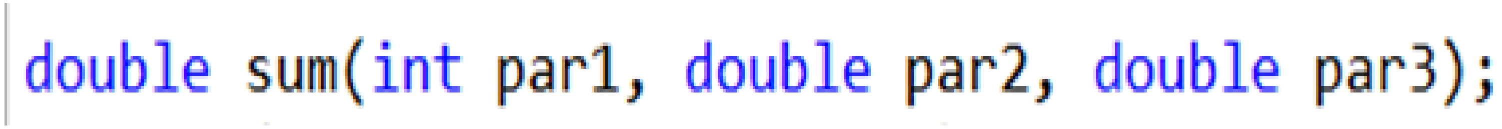
(4) Example 2 - double function

- Write a program that simulates the math power function.
- Lec7Ex2.cpp - comment on
  - Function definition placement
  - Naming convention
  - Choosing same variables names

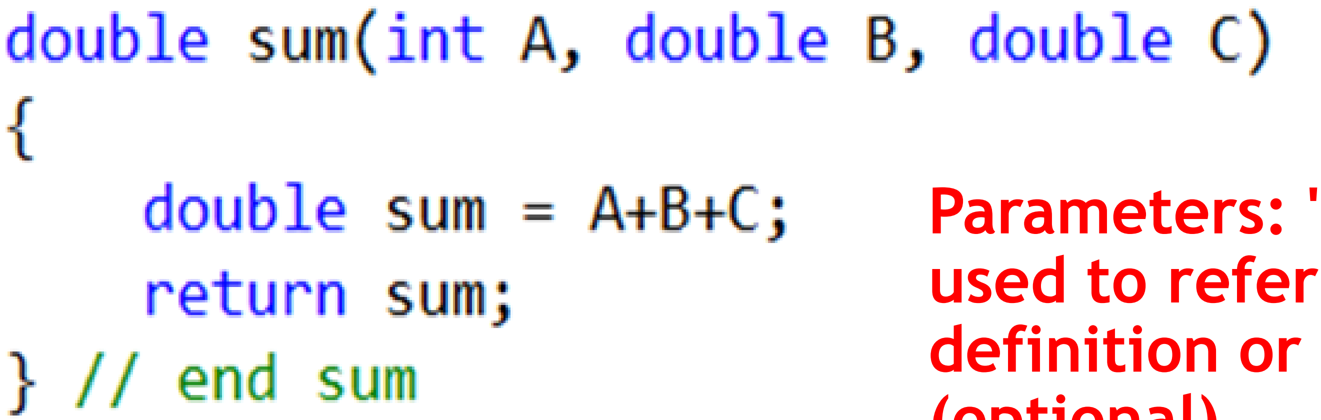
2. Programmer-defined Functions – (cont.)

(5) Parameters vs Arguments

- Formal parameters/arguments
  - In function declaration
  - In function definition’s header



```
double sum(int par1, double par2, double par3);
```



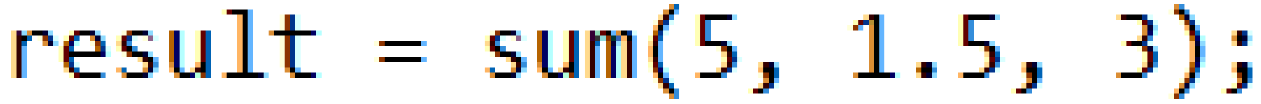
```
double sum(int A, double B, double C)
{
    double sum = A+B+C;
    return sum;
} // end sum
```

Parameters: "Variables names" used to refer to data in definition or declaration (optional)

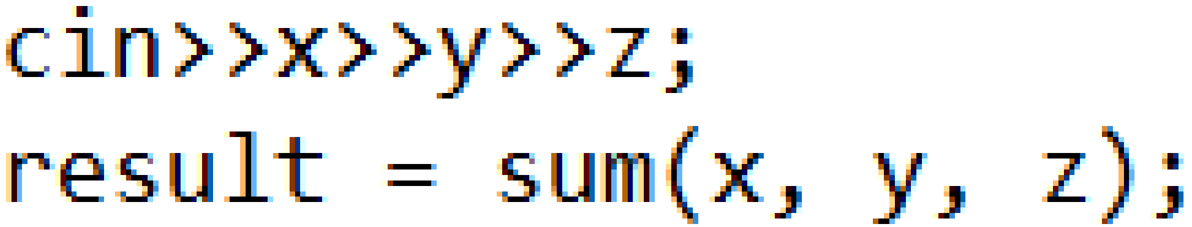
2. Programmer-defined Functions – (cont.)

(5) Parameters vs Arguments

- Actual parameters/arguments
  - In function call



```
result = sum(5, 1.5, 3);
```

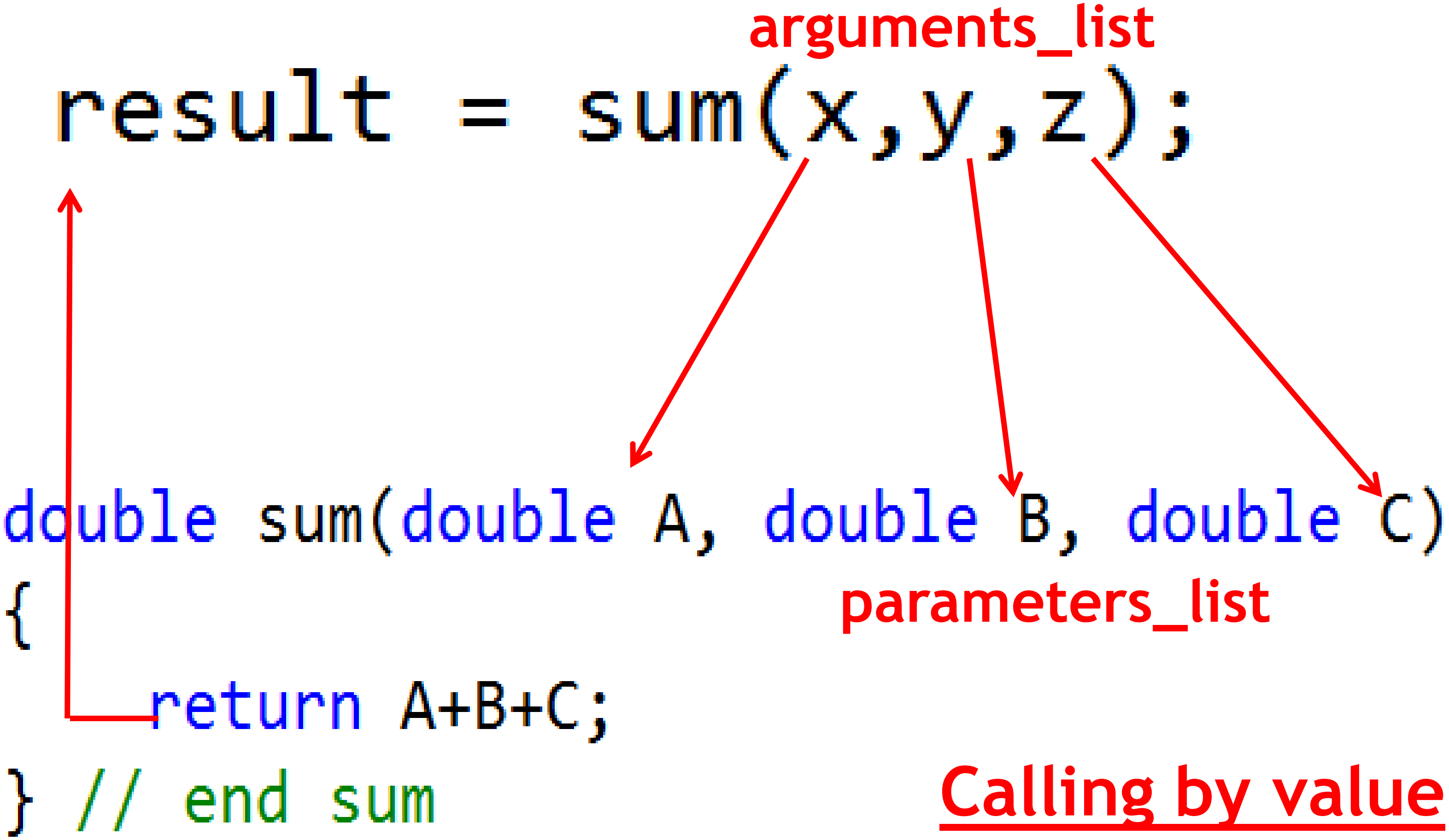


```
cin>>x>>y>>z;
result = sum(x, y, z);
```

Calling by value

2. Programmer-defined Functions – (cont.)

(5) Parameters vs Arguments



```
result = sum(x,y,z);
```

```
double sum(double A, double B, double C)
{
    return A+B+C;
} // end sum
```

Calling by value

2. Programmer-defined Functions – (cont.)

Keep in mind:

Calling/Passing by value

1. The value of the argument (not the variable) that is plugged in for the formal parameter.

2. Arguments are plugged in for formal parameters in the order they appear in the function call.

3. When an argument is plugged in for a formal parameter, it is plugged in for ALL instances of this parameter that occur in the function body.

12
1
2
3
11
10
9
8
TIME
for a
BREAK!

2. Programmer-defined Functions – (cont.)

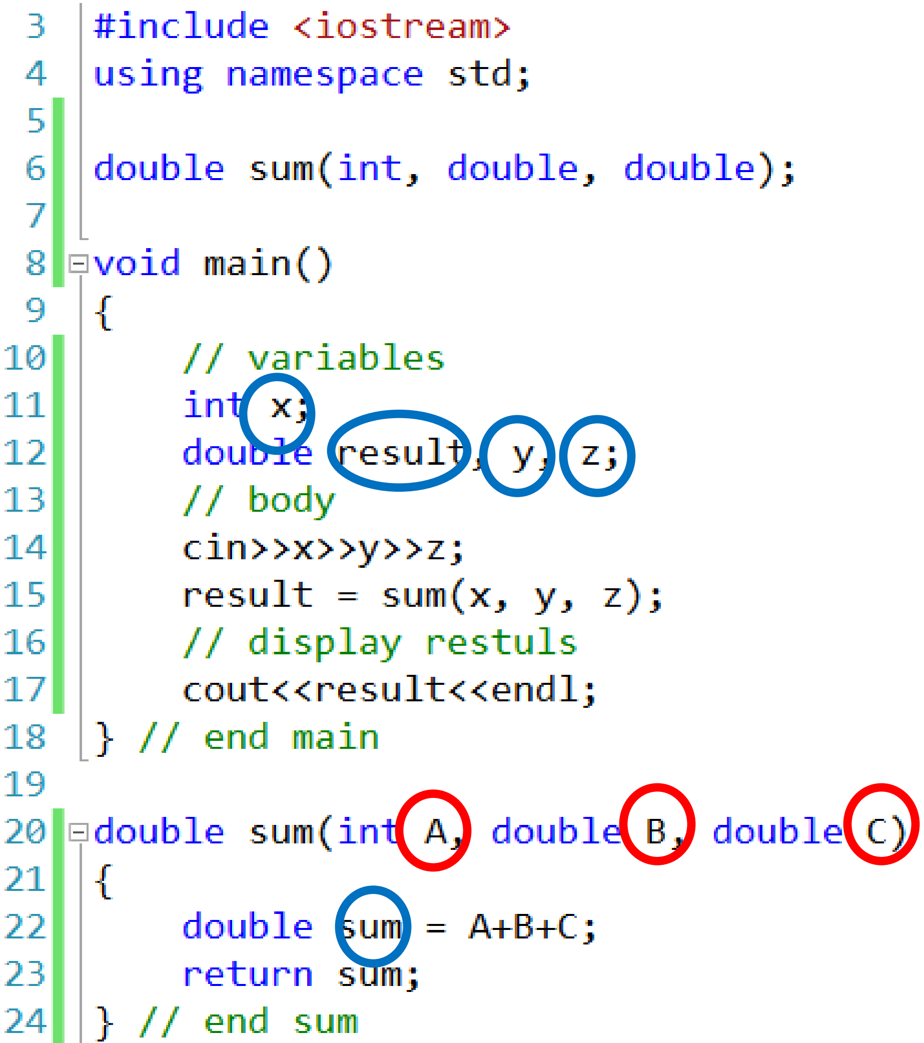
(6) Local Variables

- Declared inside body of given function
- Available only within that function

- Can have variables with same names declared in different functions
  - Scope is local: "that function is it's scope"
- Local variables preferred
  - Maintain individual control over data
  - Functions should declare whatever local data needed to "do their job“.

2. Programmer-defined Functions – (cont.)

(6) Local Variables

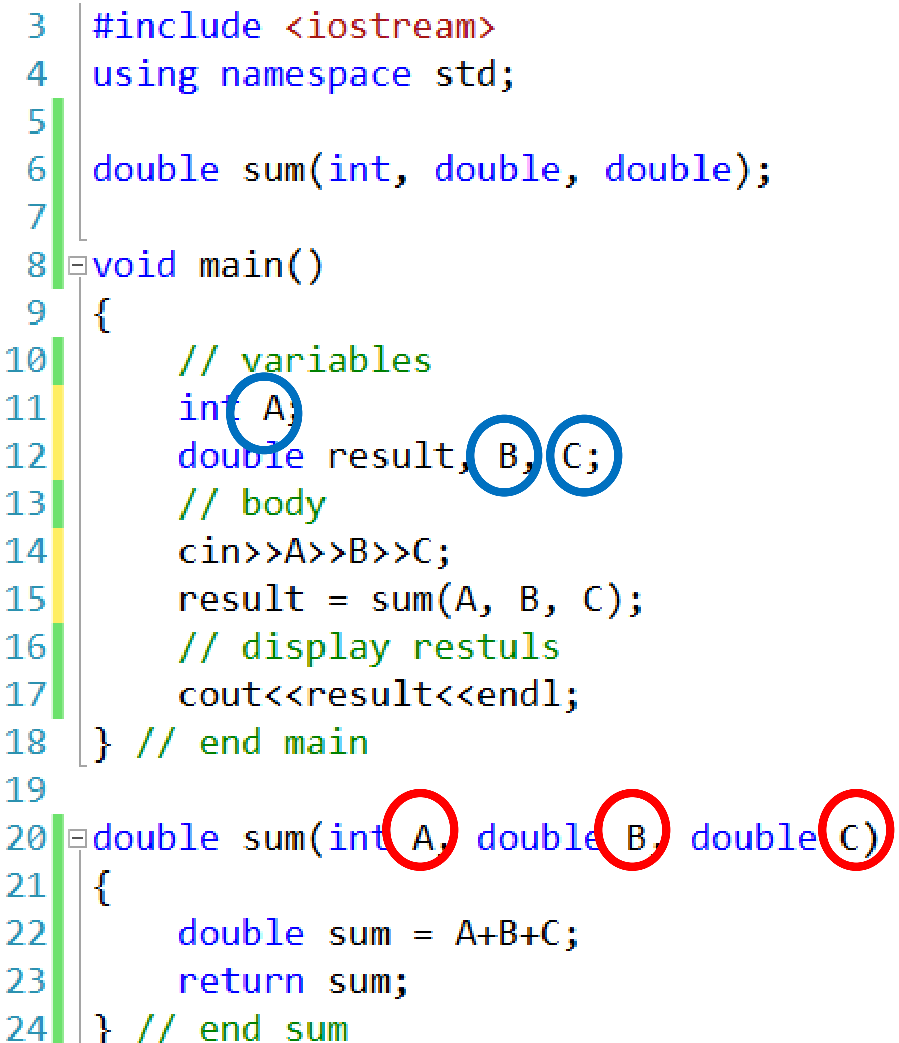


```
#include <iostream>
using namespace std;

double sum(int, double, double);

void main()
{
    // variables
    int x;
    double result, y, z;
    // body
    cin>>x>>y>>z;
    result = sum(x, y, z);
    // display restuls
    cout<<result<<endl;
} // end main

double sum(int A, double B, double C)
{
    double sum = A+B+C;
    return sum;
} // end sum
```

- Local variables are declared within a scope.
- Formal parameters are local variables to their functions.
- A function can declare its own local variables as needed.

2. Programmer-defined Functions – (cont.)

(6) Local Variables



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

double sum(int, double, double);

void main()
{
    // variables
    int A;
    double result, B, C;
    // body
    cin>>A>>B>>C;
    result = sum(A, B, C);
    // display restuls
    cout<<result<<endl;
} // end main

double sum(int A, double B, double C)
{
    double sum = A+B+C;
    return sum;
} // end sum
```

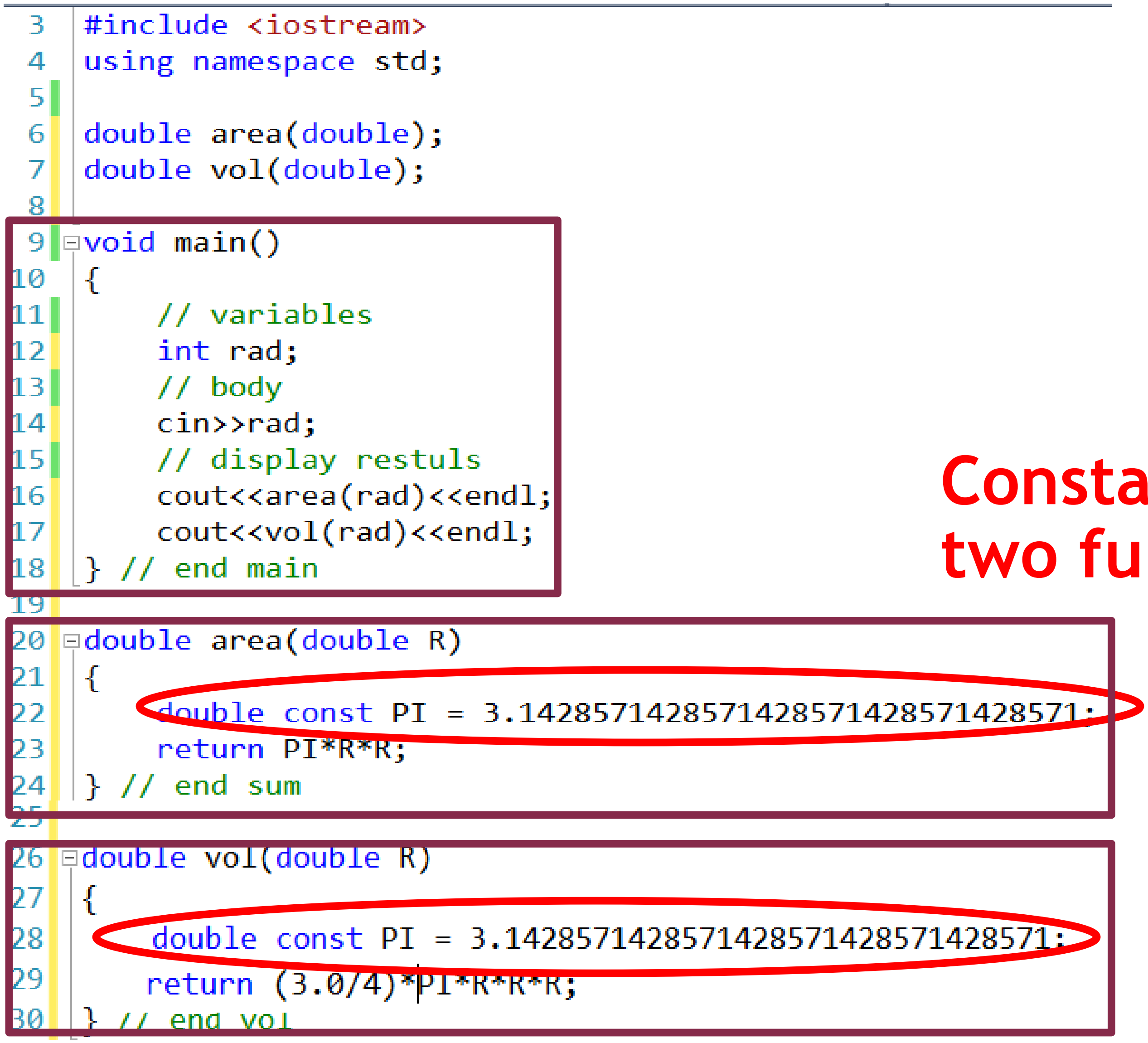
- **Can have variables with same names declared in different functions**
- **Same name BUT they are totally different locations in memory.**
- **Example3 Lec7Ex3.cpp**
  - **Same variables names**
  - **Display values to check if they differ.**

2. Programmer-defined Functions – (cont.)

(7) Global Constants/Variables

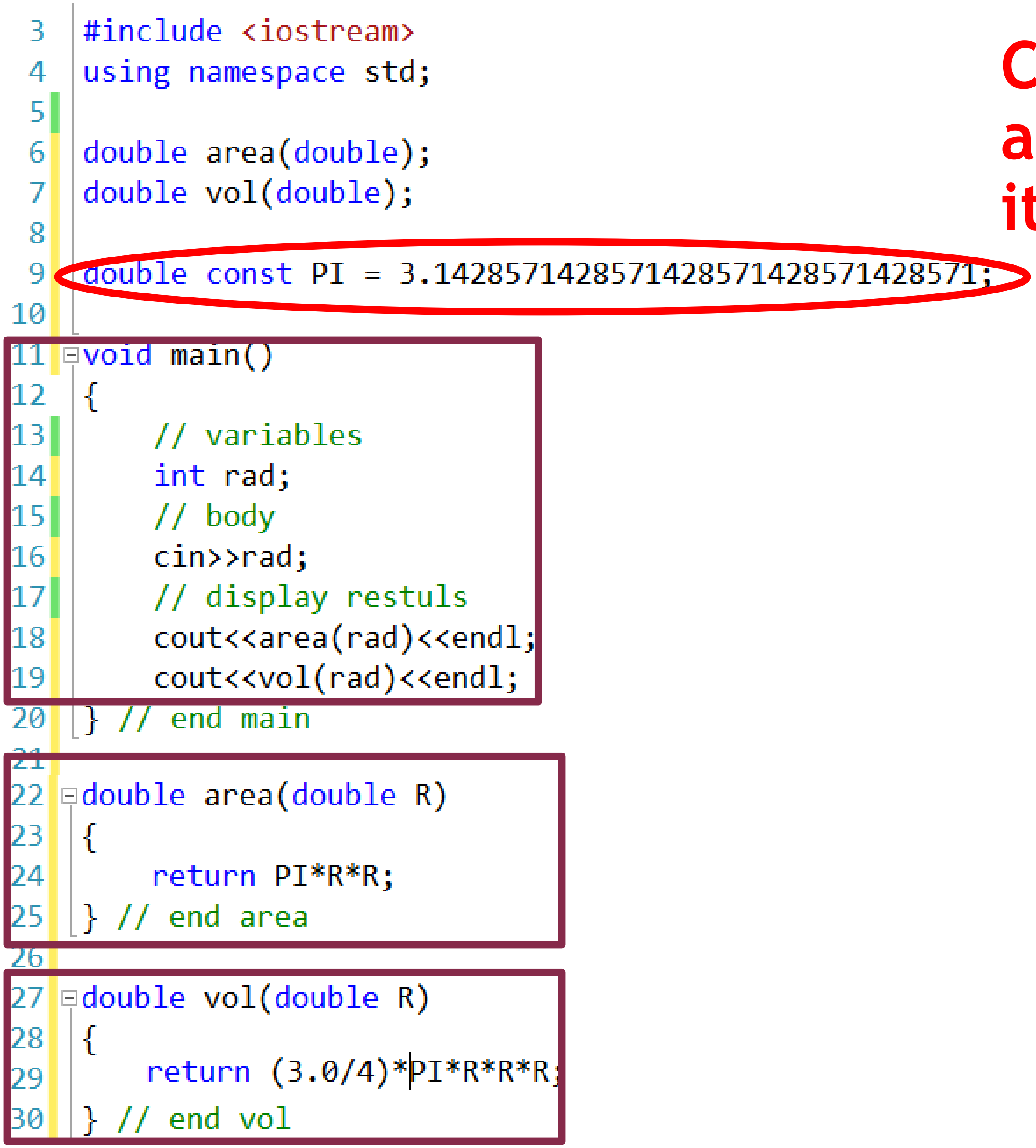
- Declared outside all modules, hence defined for all modules below it.

2. Programmer-defined Functions – (cont.)



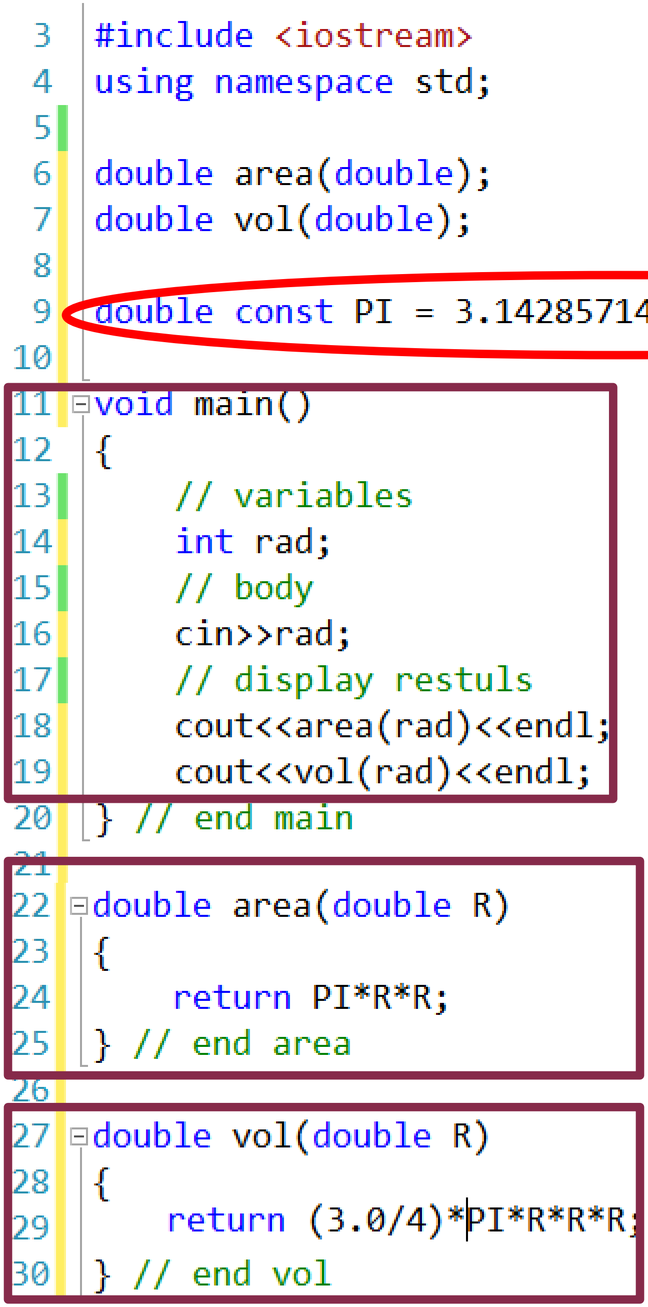
```
#include <iostream>
using namespace std;

double area(double);
double vol(double);

void main()
{
    // variables
    int rad;
    // body
    cin>>rad;
    // display restuls
    cout<<area(rad)<<endl;
    cout<<vol(rad)<<endl;
} // end main

double area(double R)
{
    double const PI = 3.1428571428571428571428571428571;
    return PI*R*R;
} // end sum

double vol(double R)
{
    double const PI = 3.1428571428571428571428571428571;
    return (3.0/4)*PI*R*R*R;
} // end vol
```

Constant needed in the two functions



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

double area(double);
double vol(double);

double const PI = 3.142857142857142857142857142857l;

void main()
{
    // variables
    int rad;
    // body
    cin>>rad;
    // display restuls
    cout<<area(rad)<<endl;
    cout<<vol(rad)<<endl;
} // end main

double area(double R)
{
    return PI*R*R;
} // end area

double vol(double R)
{
    return (3.0/4)*PI*R*R*R;
} // end vol
```

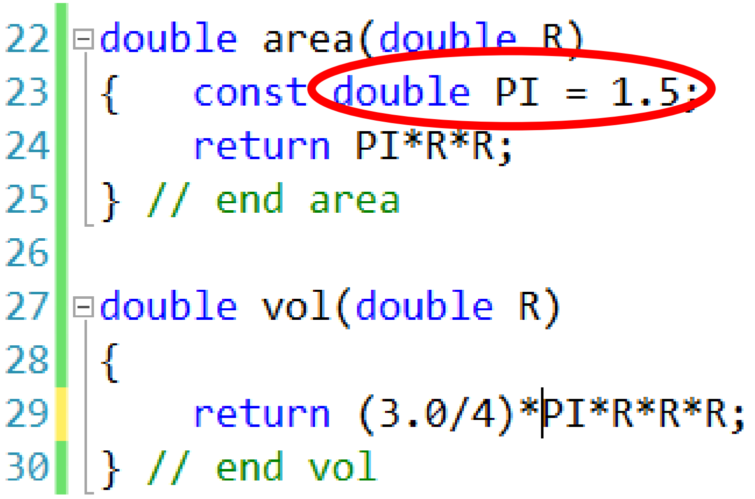
Constant global to all functions below its declaration



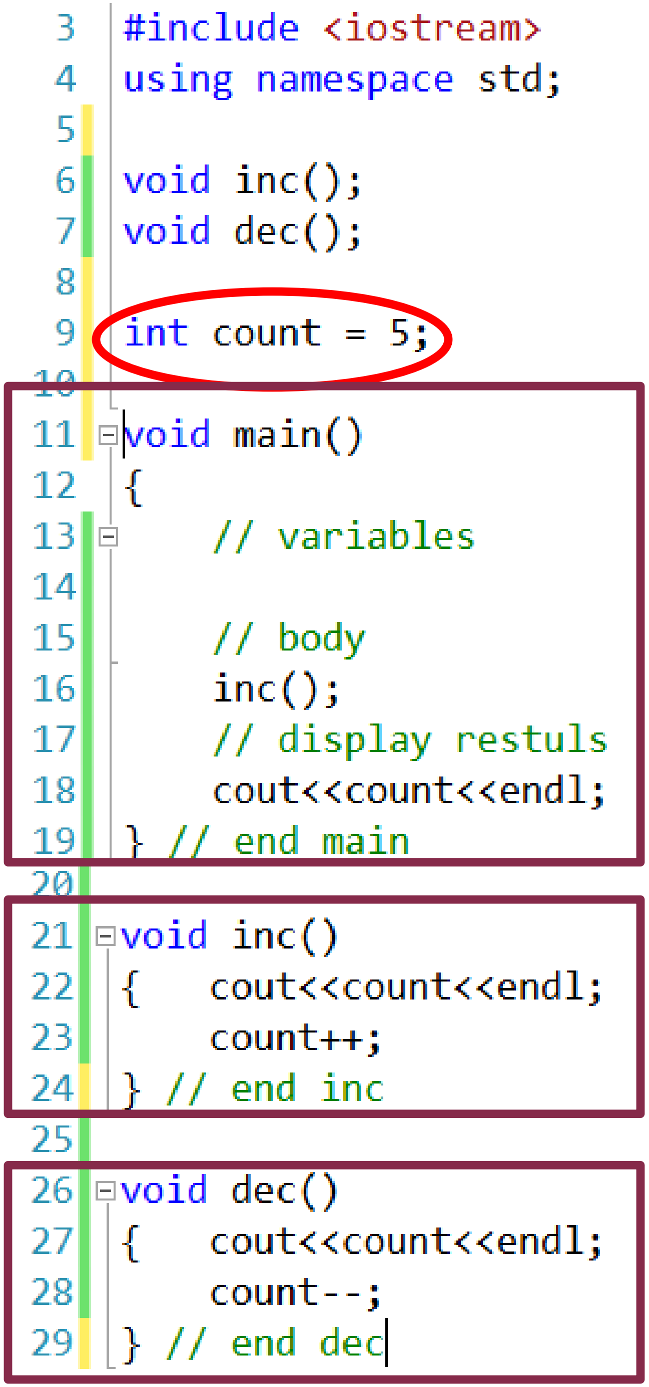
```
#include <iostream>
using namespace std;

double area(double);
double vol(double);

double const PI = 3.1428571428571428571428571428571;

void main()
{
    // variables
    int rad;
    // body
    cin>>rad;
    // display restuls
    cout<<area(rad)<<endl;
    cout<<vol(rad)<<endl;
} // end main

double area(double R)
{
    return PI*R*R;
} // end area

double vol(double R)
{
    return (3.0/4)*PI*R*R*R;
} // end vol
```

Constant global to all functions below its declaration

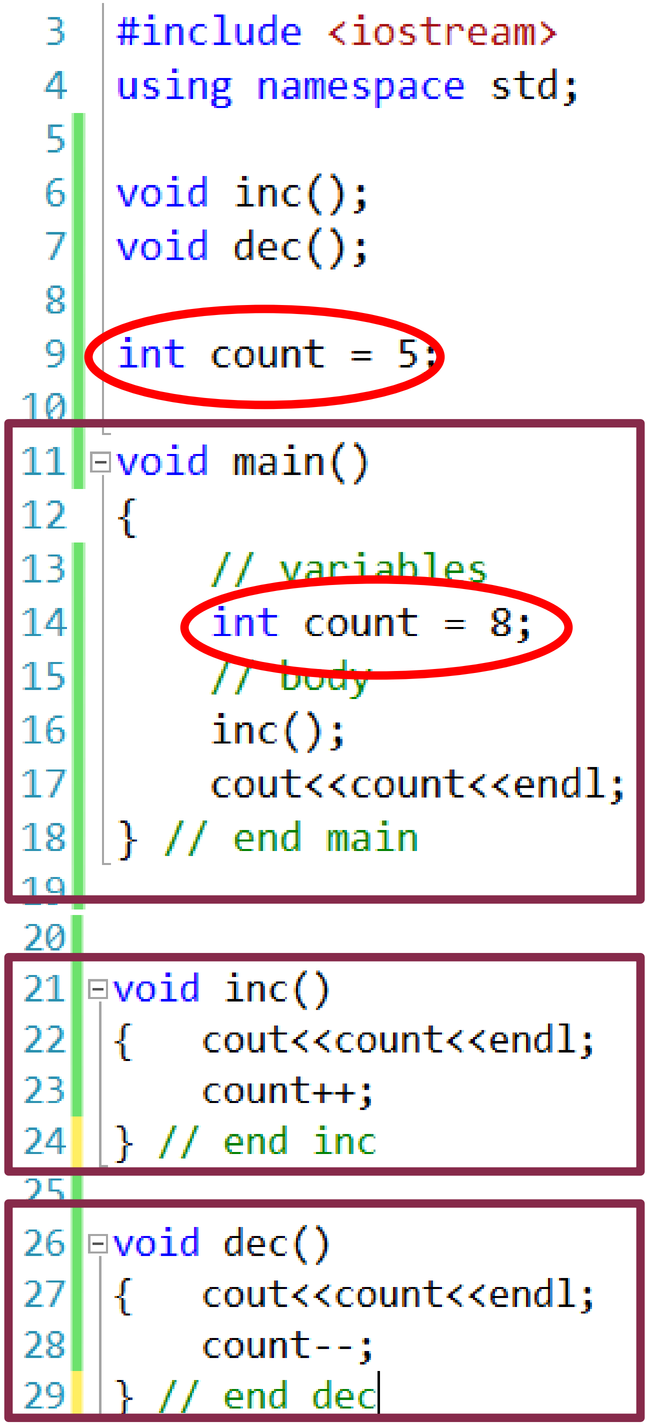
What if we declared a local variable with the same name as a global one?



```
double area(double R)
{    const double PI = 1.5;
     return PI*R*R;
} // end area

double vol(double R)
{
     return (3.0/4)*PI*R*R*R;
} // end vol
```



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

void inc();
void dec();

int count = 5;

void main()
{
    // variables

    // body
    inc();
    // display restuls
    cout<<count<<endl;
} // end main

void inc()
{   cout<<count<<endl;
    count++;
} // end inc

void dec()
{   cout<<count<<endl;
    count--;
} // end dec
```



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

void inc();
void dec();

int count = 5;

void main()
{
    // variables
    int count = 8;
    // body
    inc();
    cout<<count<<endl;
} // end main

void inc()
{   cout<<count<<endl;
    count++;
} // end inc

void dec()
{   cout<<count<<endl;
    count--;
} // end dec
```

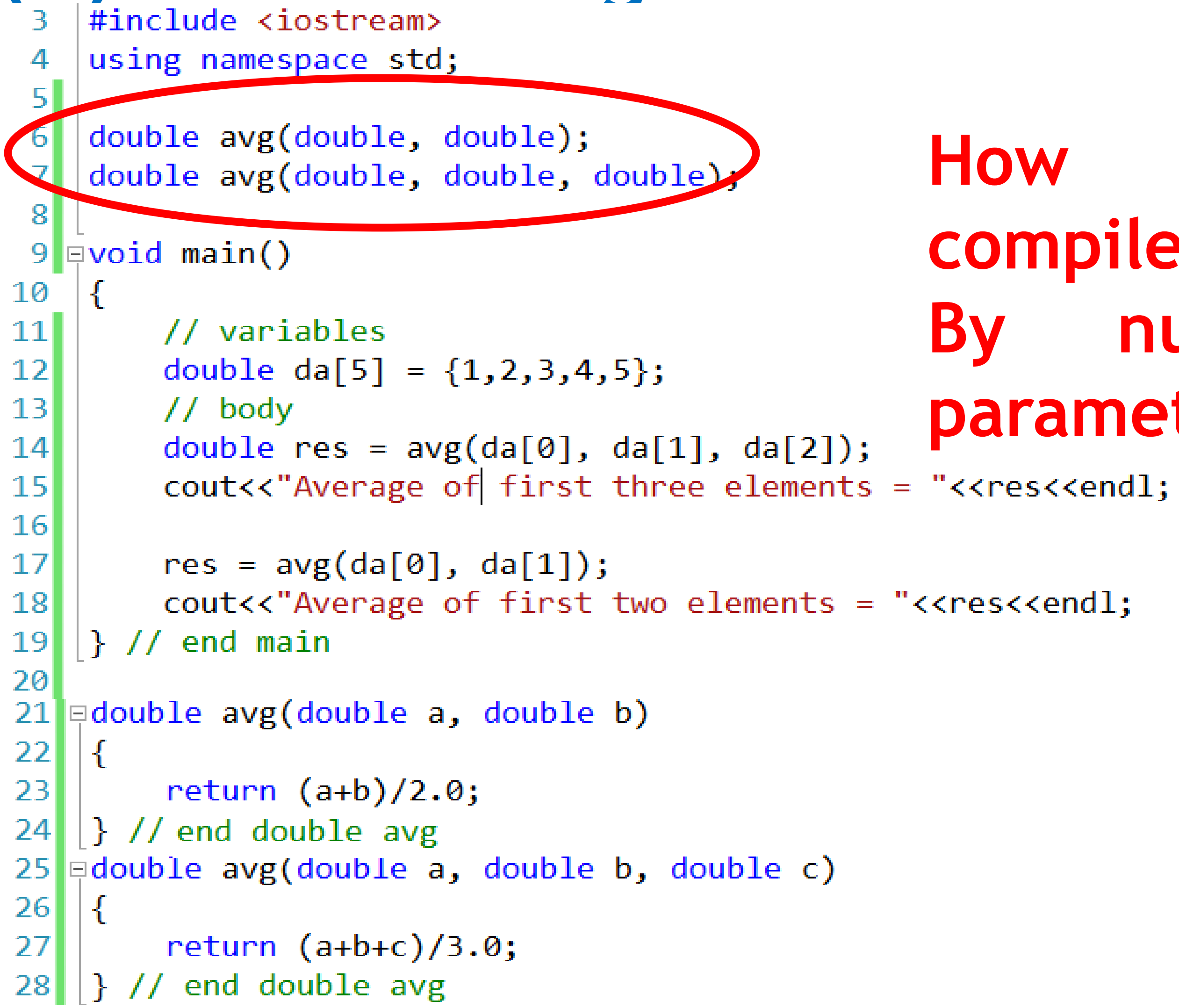
2. Programmer-defined Functions – (cont.)

(8) Overloading a Function Name

- Declaring/Defining more than one function with the same name.
- How does the compiler know?
  - By number of parameters
  - By type of parameter
  - NOT by return value.
- Should be used with care!

2. Programmer-defined Functions – (cont.)

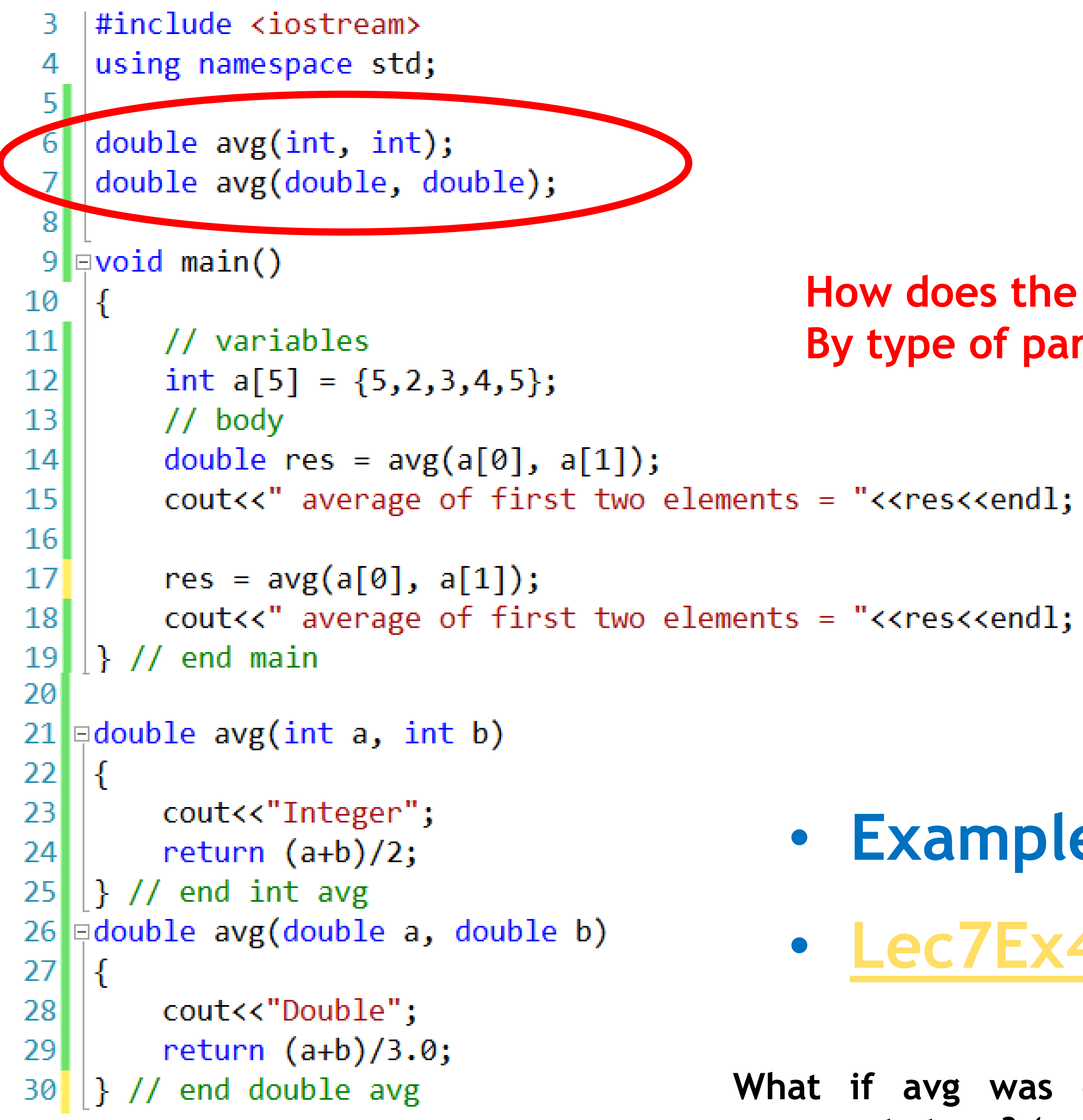
(8) Overloading a Function Name



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

double avg(double, double);
double avg(double, double, double);

void main()
{
    // variables
    double da[5] = {1,2,3,4,5};
    // body
    double res = avg(da[0], da[1], da[2]);
    cout<<"Average of first three elements = "<<res<<endl;

    res = avg(da[0], da[1]);
    cout<<"Average of first two elements = "<<res<<endl;
} // end main

double avg(double a, double b)
{
    return (a+b)/2.0;
} // end double avg
double avg(double a, double b, double c)
{
    return (a+b+c)/3.0;
} // end double avg
```

How does the compiler know?

By number of parameters



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

double avg(int, int);
double avg(double, double);

void main()
{
    // variables
    int a[5] = {5,2,3,4,5};
    // body
    double res = avg(a[0], a[1]);
    cout<<" average of first two elements = "<<res<<endl;

    res = avg(a[0], a[1]);
    cout<<" average of first two elements = "<<res<<endl;
} // end main

double avg(int a, int b)
{
    cout<<"Integer";
    return (a+b)/2;
} // end int avg
double avg(double a, double b)
{
    cout<<"Double";
    return (a+b)/3.0;
} // end double avg
```

How does the compiler know?
By type of parameters

- Example4
- Lec7Ex4.cpp

What if avg was called with two different arguments type? (syntax error)

2. Programmer-defined Functions – (cont.)

(9) Passing Different Types

- Formal parameters, and hence arguments, can be of simple types (int, float, char, …), or of aggregate types.

(10) Returning Different Types

- A function's return value, can be of simple type (int, float, char, …), or of aggregate type.

3. Examples

Example 5 - Lec7Ex5.cpp

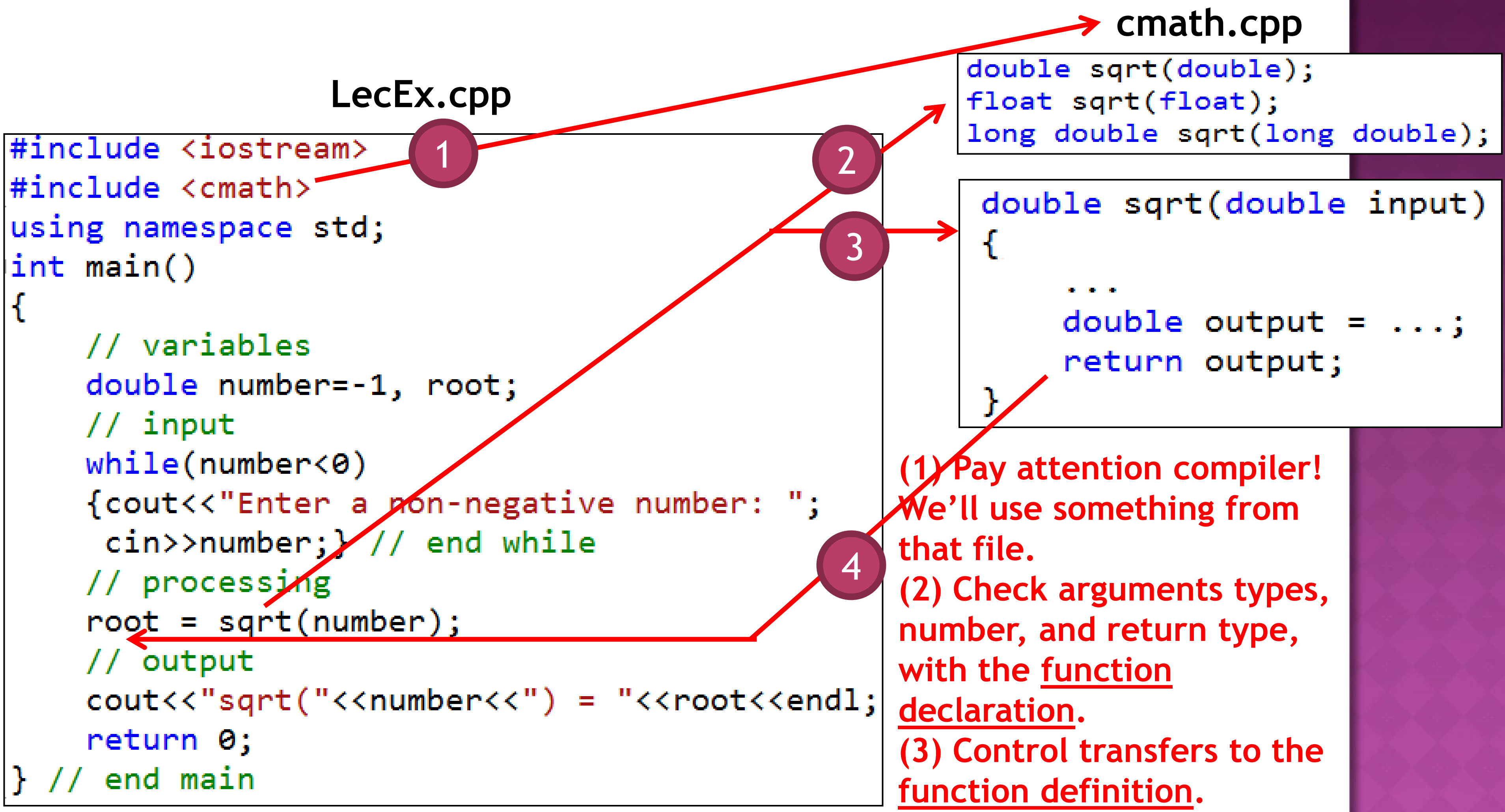
- Write a function definition for a function called `in_order` that takes three arguments and returns true of the three arguments are ordered, otherwise it returns false.

Summary of Function usage

Input	None Starting value(s)
Processing	Job done Job done on the input
Output	Result of processing Result of processing the input(s)
Declaration	Function name and type of input and output
Definition	Job in detail
Call	Function name Function Name with input argument(s) Output = Function Name Output = Function name with the input argument(s)

What if we need more than one thing from the function?

Summary of Function usage



LecEx.cpp

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    // variables
    double number=-1, root;
    // input
    while(number<0)
    {cout<<"Enter a non-negative number: ";
     cin>>number;} // end while
    // processing
    root = sqrt(number);
    // output
    cout<<"sqrt("<<number<<") = "<<root<<endl;
    return 0;
} // end main
```

cmath.cpp

```
double sqrt(double);
float sqrt(float);
long double sqrt(long double);
```

```
double sqrt(double input)
{
    ...
    double output = ...;
    return output;
}
```

(1) Pay attention compiler! We'll use something from that file.

(2) Check arguments types, number, and return type, with the function declaration.

(3) Control transfers to the function definition.

(4) Control returns to the caller bringing back a value.

Class Accumulative Project: Employees Salary for Companies

Class Accumulative Project: Employees Salary for Companies

Tasks 1, 2, 3,4 (DONE☺)

TASKs 5 and 6 (2 NEW* BONUS):

- **Add variable name** to the employee and company structs , that can accept a name composed of more than 1 word input by the user.
- **Create** a login function that takes as input the correct password and let the user enter his password (displayed as stars/astriscks) and returns a boolean variable verifying whether he logged in successfully or not. It should look like this:

  bool Login(string correctpw)

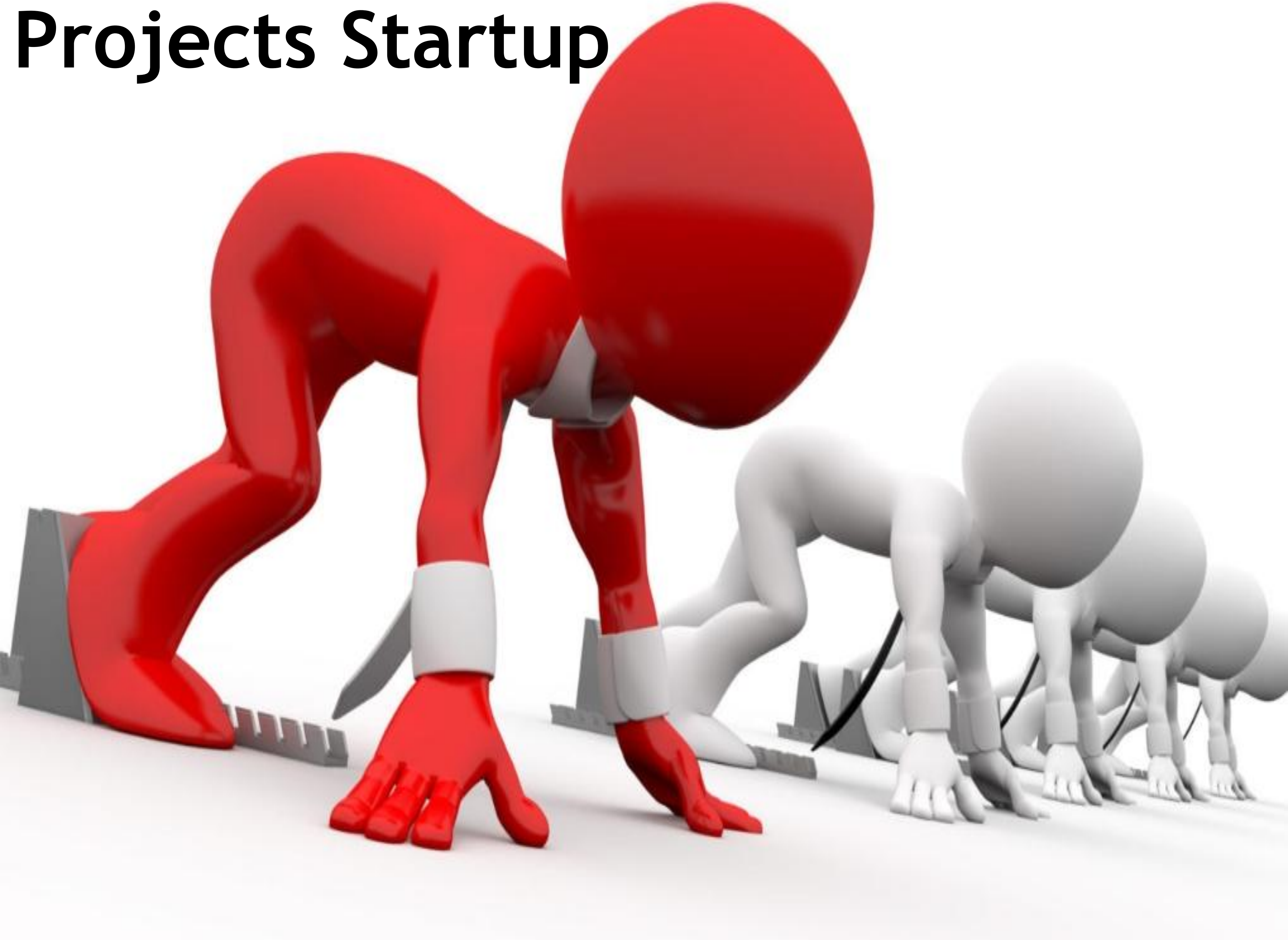
- **Create** a function that takes as input the gross pay and total taxes and calculates the net salary for each employee. It should look like this:

  float ComputeNetSalary (float grossPay, float taxes)

- (Extra Bonus)
- **Save** the 10 employees information in a text file and name the text file with the company name.
- **Make the appropriate changes.**
- Submit your code as text in this form, **from Thursday 5/4/2018** till **due Date Friday 13/4/2018 at 11:59 pm (Extended due to late lectures)**

https://goo.gl/forms/xdjOdxjF7VE2gY6o1

Projects Startup



Projects

- Mentors Names/Emails/Office Hours.
- Delivery by the end of April.
- C++
- GUI is optional (Good to learn)

4. Summary

- Procedural Abstraction through a top-down approach (also known as step-wise design) is essentially the breaking down of a system to gain insight into its compositional sub-systems. In a top-down approach an overview of the system is formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. A top-down model is often specified with the assistance of "black boxes", these make it easier to manipulate. However, black boxes may fail to clarify elementary mechanisms or be detailed enough to realistically validate the model.

4. Summary – (cont.)

- Two kinds of functions:
  - "Return-a-value" and void functions
- Functions should be "black boxes"
  - Hide "how" details
  - Declare own local data
- Function declarations should self-document
  - Provide pre- & post-conditions in comments
  - Provide all "caller" needs for use

4. Summary – (cont.)

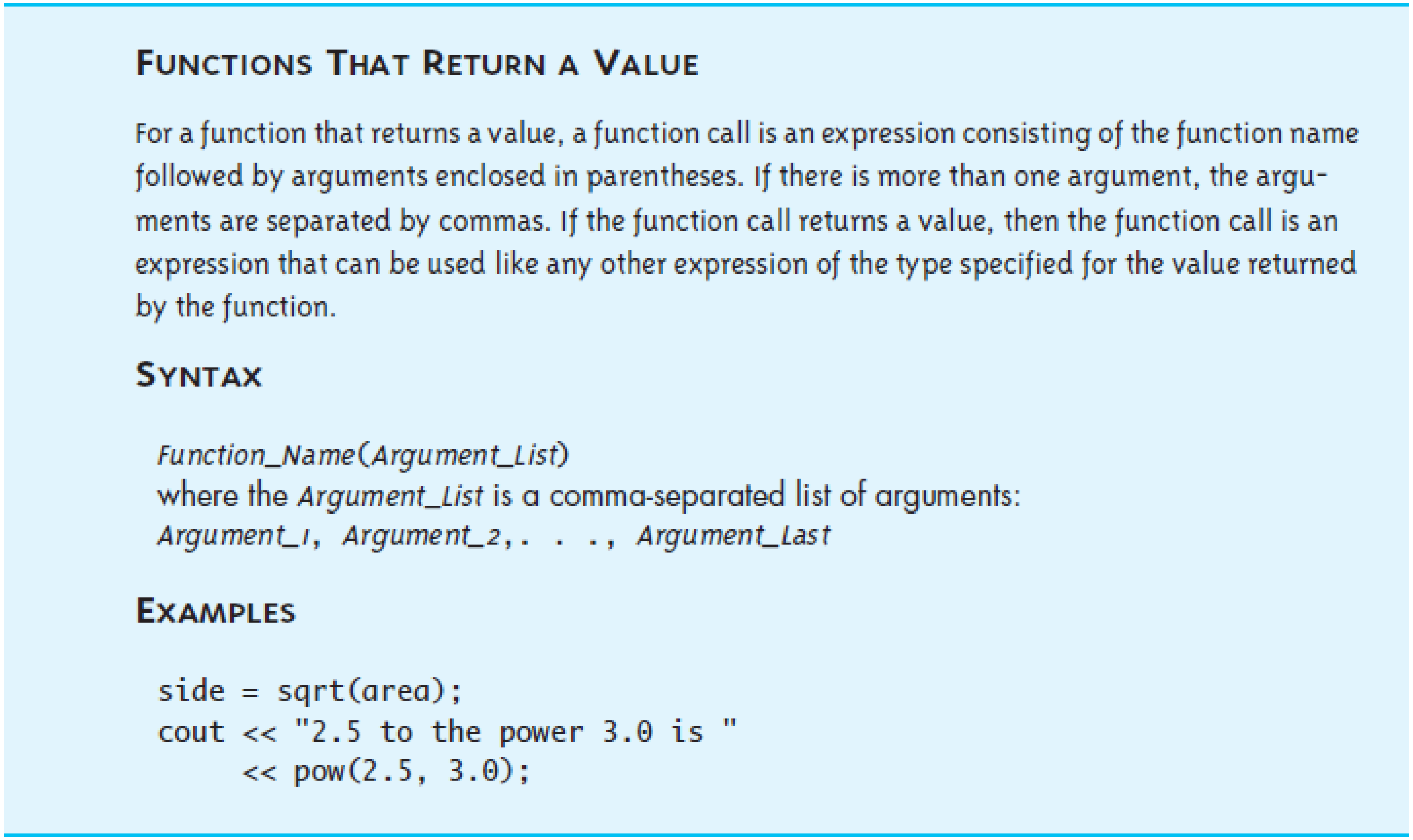
- Local data
  - Declared in function definition
- Global data
  - Declared above function definitions
  - OK for constants, not for variables
- Parameters/Arguments
  - Formal: In function declaration and definition
    - Placeholder for incoming data
  - Actual: In function call
    - Actual data passed to function

4. Summary – (cont.)

Keep in mind:

- Procedural abstraction.
- Need to know "what" function does, not "how" it does it!
- Think "black box"
  - Device you know how to use, but not it’s method of operation
- Implement functions like black box
  - User of function only needs: declaration
  - Does NOT need function definition (Called Information Hiding).

4. Summary – (cont.)



Functions That Return a Value

For a function that returns a value, a function call is an expression consisting of the function name followed by arguments enclosed in parentheses. If there is more than one argument, the arguments are separated by commas. If the function call returns a value, then the function call is an expression that can be used like any other expression of the type specified for the value returned by the function.

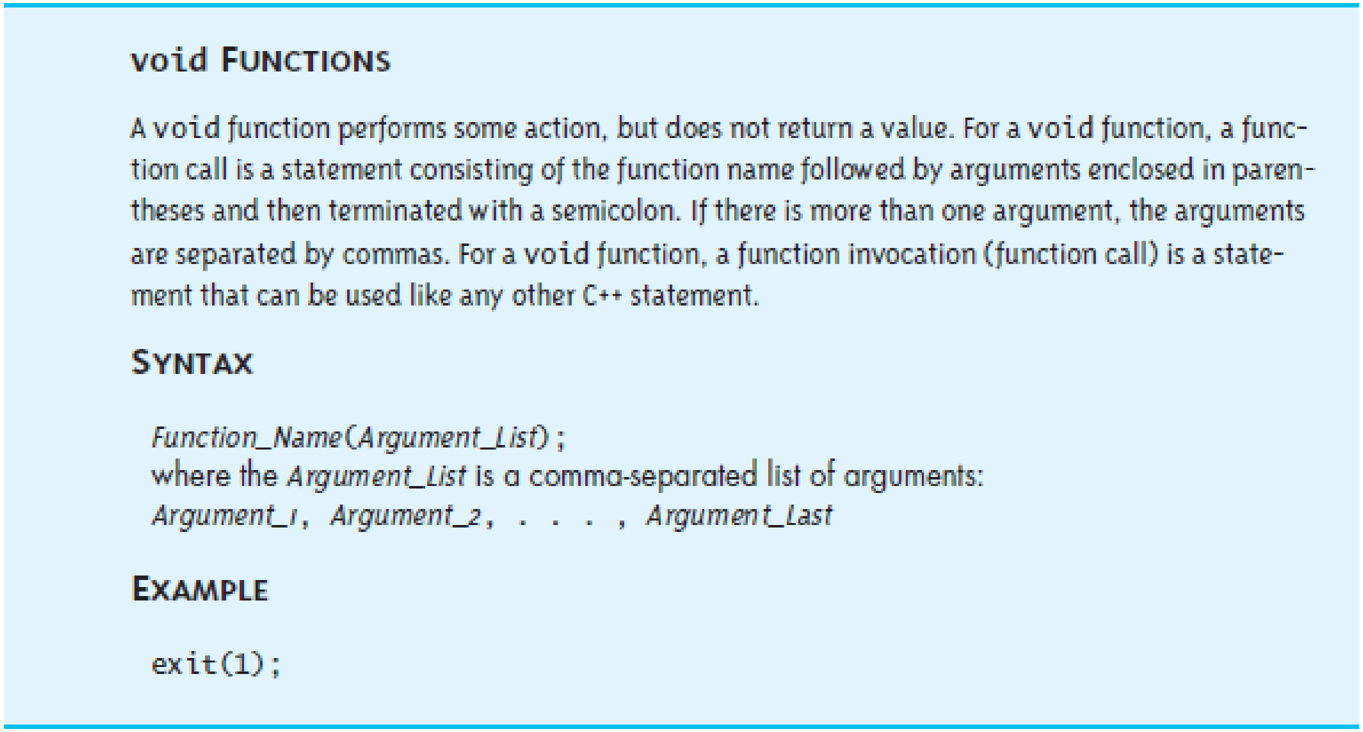
Syntax

Function_Name(Argument_List)
where the *Argument_List* is a comma-separated list of arguments:
Argument_1, Argument_2,. . ., Argument_Last

Examples

```
side = sqrt(area);
cout << "2.5 to the power 3.0 is "
     << pow(2.5, 3.0);
```

4. Summary – (cont.)



void FUNCTIONS

A void function performs some action, but does not return a value. For a void function, a function call is a statement consisting of the function name followed by arguments enclosed in parentheses and then terminated with a semicolon. If there is more than one argument, the arguments are separated by commas. For a void function, a function invocation (function call) is a statement that can be used like any other C++ statement.

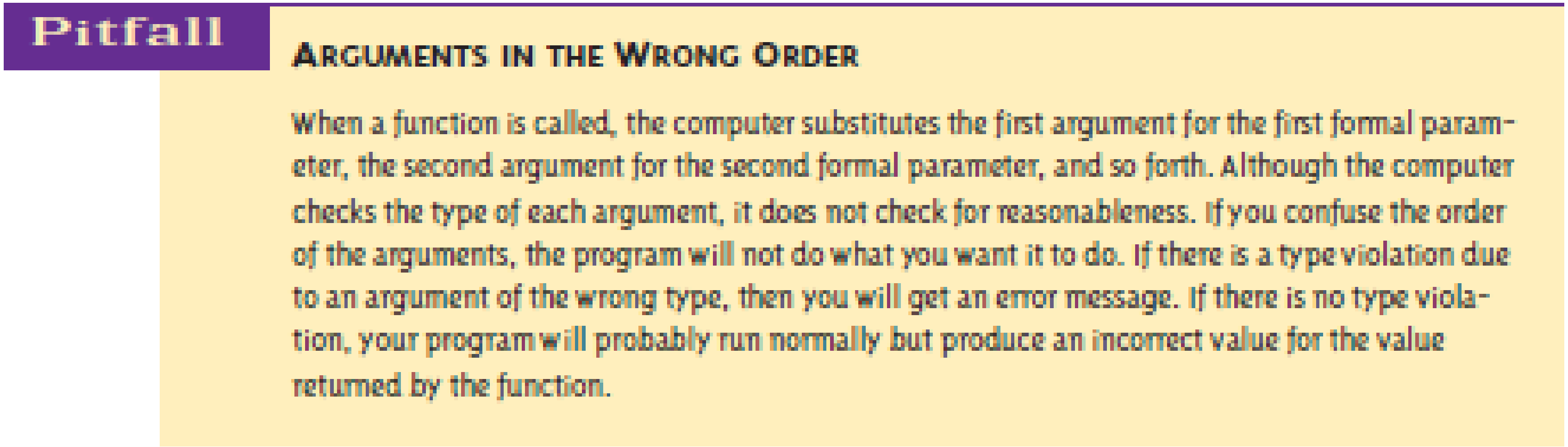
SYNTAX

Function_Name(*Argument_List*);
where the *Argument_List* is a comma-separated list of arguments:
Argument_1, *Argument_2*, . . . , *Argument_Last*

EXAMPLE

```
exit(1);
```

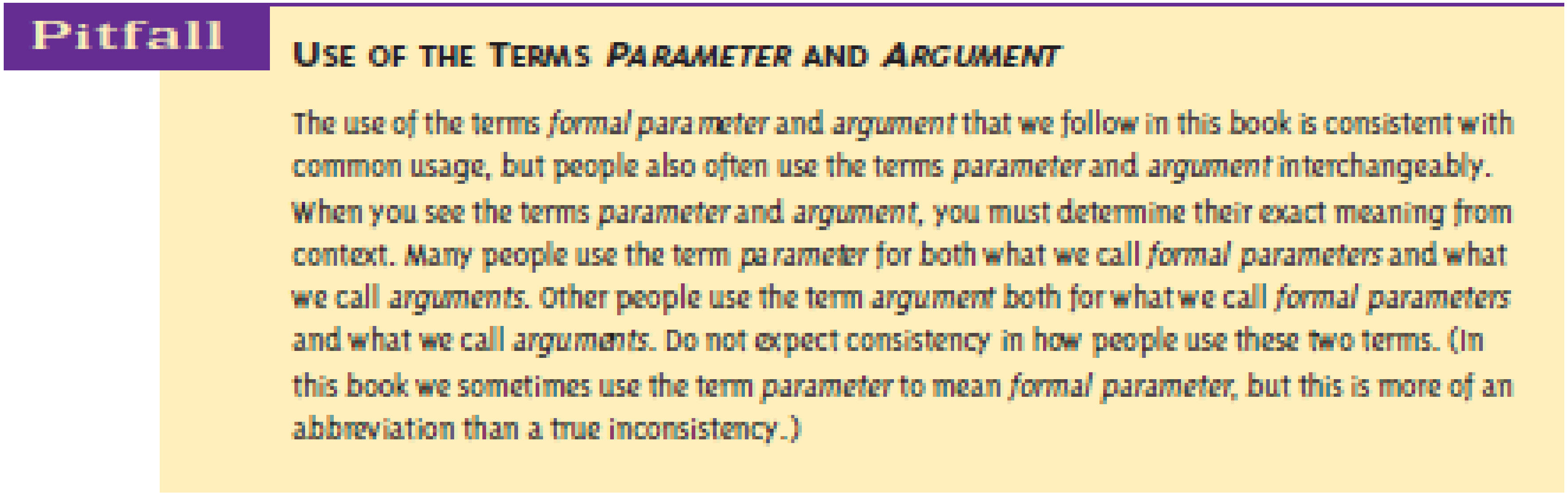
4. Summary – (cont.)



Pitfall

Arguments in the Wrong Order

When a function is called, the computer substitutes the first argument for the first formal parameter, the second argument for the second formal parameter, and so forth. Although the computer checks the type of each argument, it does not check for reasonableness. If you confuse the order of the arguments, the program will not do what you want it to do. If there is a type violation due to an argument of the wrong type, then you will get an error message. If there is no type violation, your program will probably run normally but produce an incorrect value for the value returned by the function.

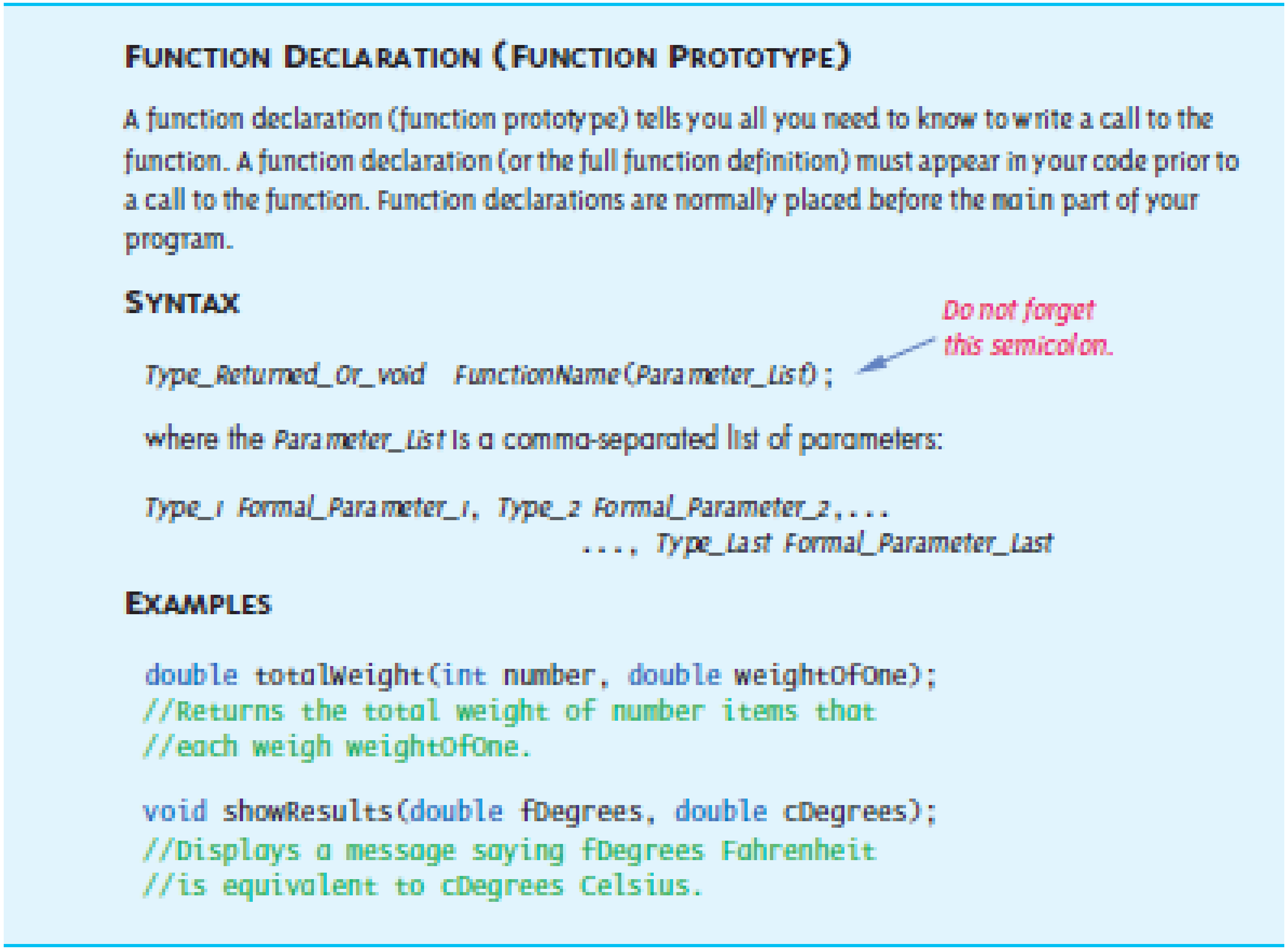


Pitfall

Use of the Terms *Parameter* and *Argument*

The use of the terms *formal parameter* and *argument* that we follow in this book is consistent with common usage, but people also often use the terms *parameter* and *argument* interchangeably. When you see the terms *parameter* and *argument*, you must determine their exact meaning from context. Many people use the term *parameter* for both what we call *formal parameters* and what we call *arguments*. Other people use the term *argument* both for what we call *formal parameters* and what we call *arguments*. Do not expect consistency in how people use these two terms. (In this book we sometimes use the term *parameter* to mean *formal parameter*, but this is more of an abbreviation than a true inconsistency.)

4. Summary – (cont.)



Function Declaration (Function Prototype)

A function declaration (function prototype) tells you all you need to know to write a call to the function. A function declaration (or the full function definition) must appear in your code prior to a call to the function. Function declarations are normally placed before the main part of your program.

Syntax

```
Type_Returned_Or_void   FunctionName(Parameter_List);
```

Do not forget this semicolon.

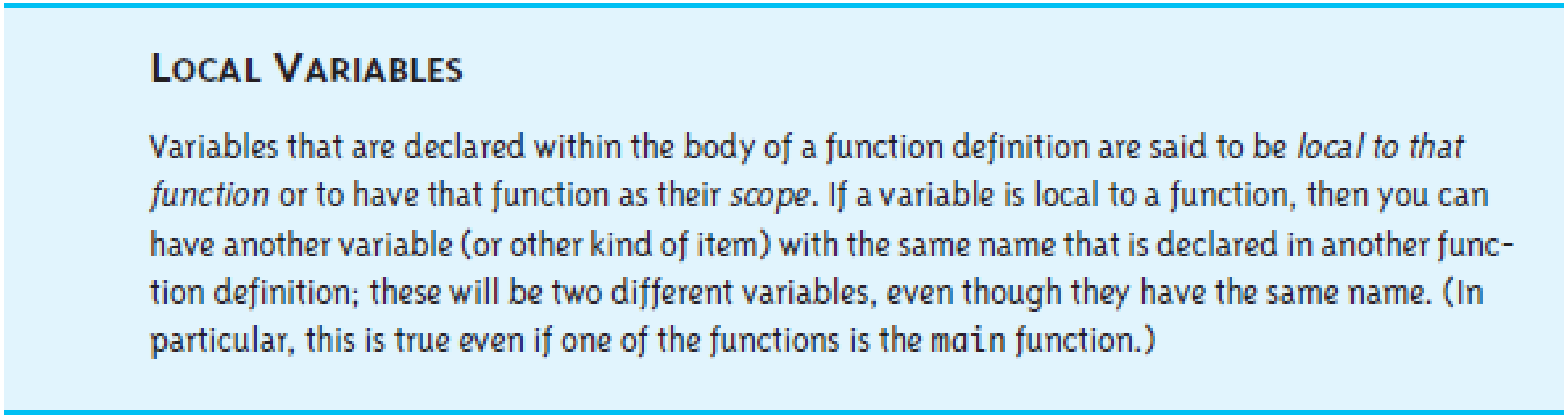
where the *Parameter_List* is a comma-separated list of parameters:

```
Type_1 Formal_Parameter_1, Type_2 Formal_Parameter_2,...
                                  ..., Type_Last Formal_Parameter_Last
```

Examples

```
double totalWeight(int number, double weightOfOne);
//Returns the total weight of number items that
//each weigh weightOfOne.

void showResults(double fDegrees, double cDegrees);
//Displays a message saying fDegrees Fahrenheit
//is equivalent to cDegrees Celsius.
```

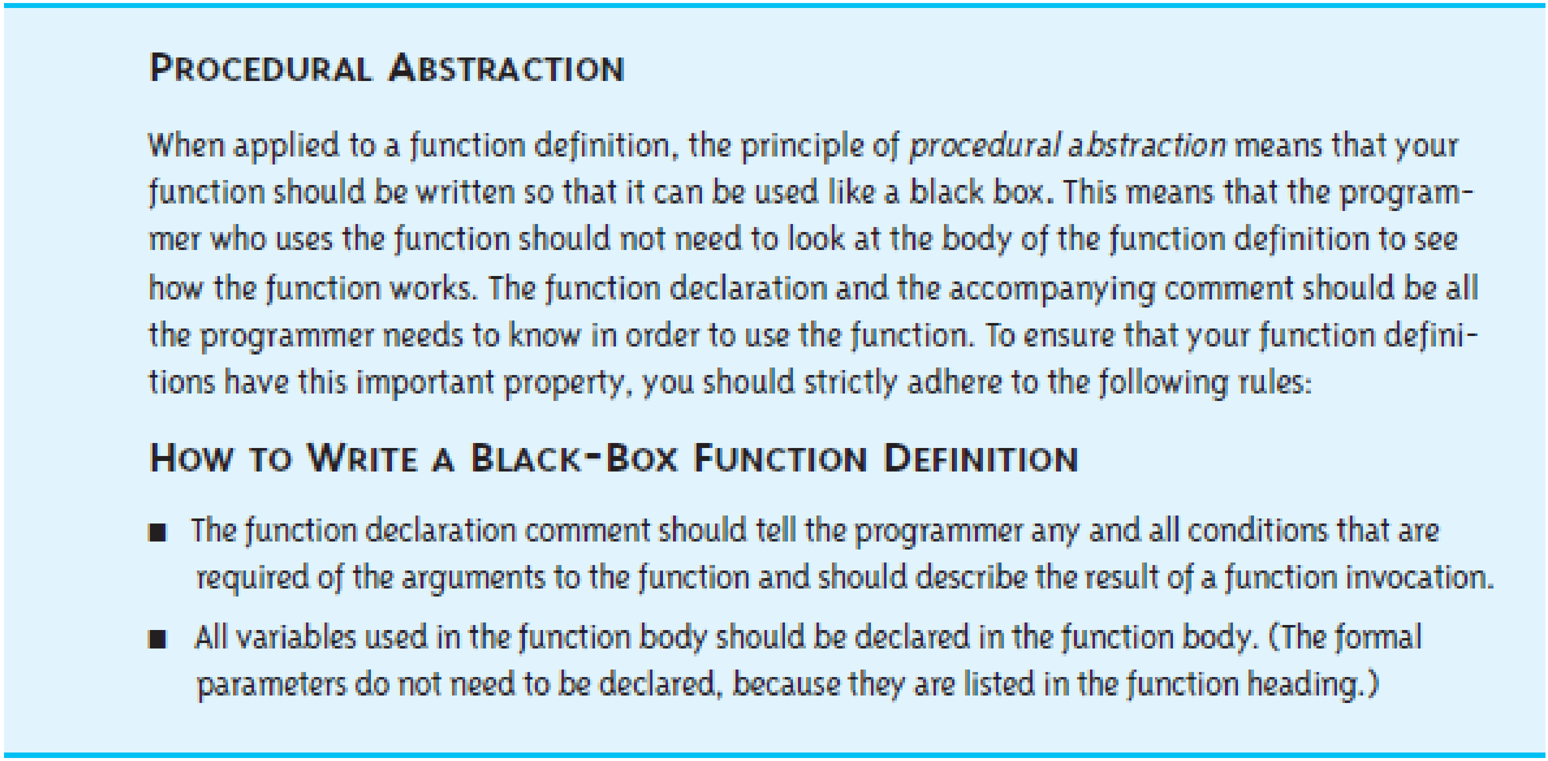
4. Summary – (cont.)



Local Variables

Variables that are declared within the body of a function definition are said to be *local to that function* or to have that function as their *scope*. If a variable is local to a function, then you can have another variable (or other kind of item) with the same name that is declared in another function definition; these will be two different variables, even though they have the same name. (In particular, this is true even if one of the functions is the `main` function.)

4. Summary – (cont.)



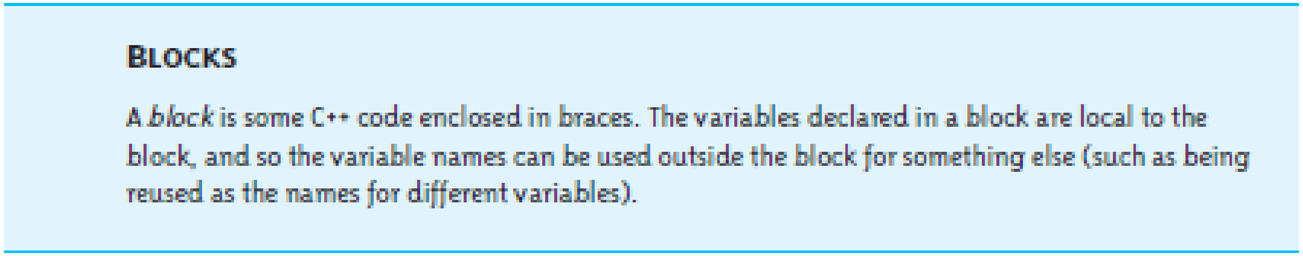
Procedural Abstraction

When applied to a function definition, the principle of *procedural abstraction* means that your function should be written so that it can be used like a black box. This means that the programmer who uses the function should not need to look at the body of the function definition to see how the function works. The function declaration and the accompanying comment should be all the programmer needs to know in order to use the function. To ensure that your function definitions have this important property, you should strictly adhere to the following rules:

How to Write a Black-Box Function Definition

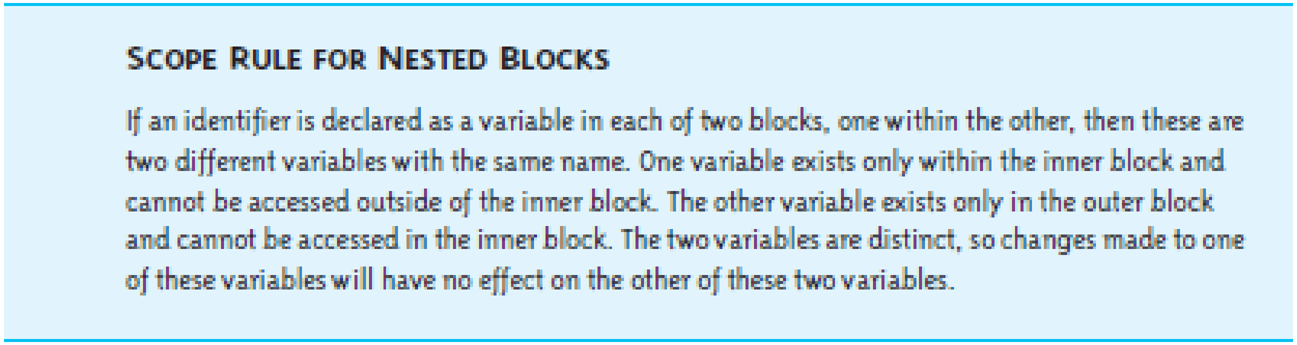
- The function declaration comment should tell the programmer any and all conditions that are required of the arguments to the function and should describe the result of a function invocation.
- All variables used in the function body should be declared in the function body. (The formal parameters do not need to be declared, because they are listed in the function heading.)

4. Summary – (cont.)



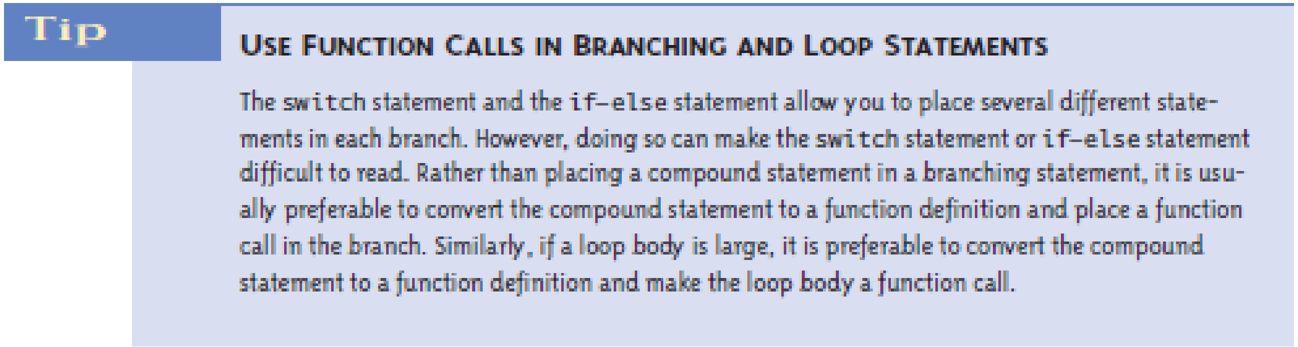
Blocks

A *block* is some C++ code enclosed in braces. The variables declared in a block are local to the block, and so the variable names can be used outside the block for something else (such as being reused as the names for different variables).



Scope Rule for Nested Blocks

If an identifier is declared as a variable in each of two blocks, one within the other, then these are two different variables with the same name. One variable exists only within the inner block and cannot be accessed outside of the inner block. The other variable exists only in the outer block and cannot be accessed in the inner block. The two variables are distinct, so changes made to one of these variables will have no effect on the other of these two variables.



Tip

Use Function Calls in Branching and Loop Statements

The `switch` statement and the `if-else` statement allow you to place several different statements in each branch. However, doing so can make the `switch` statement or `if-else` statement difficult to read. Rather than placing a compound statement in a branching statement, it is usually preferable to convert the compound statement to a function definition and place a function call in the branch. Similarly, if a loop body is large, it is preferable to convert the compound statement to a function definition and make the loop body a function call.

Thank You
ThanksImages.com