Programming Fundamentals I

Chapter 6:

User-Defined Functions

Dr. Adriana Badulescu

Objectives

- Learn about standard (predefined) functions
- Learn about user-defined functions
- Examine value-returning functions, including actual and formal parameters
- Learn how to construct and use void functions in a program
- Discover the difference between value and reference parameters
- Explore reference parameters and value-returning functions
- Learn about the scope of an identifier Discover static variables
- Learn function overloading
- Explore functions with default parameters

Introduction

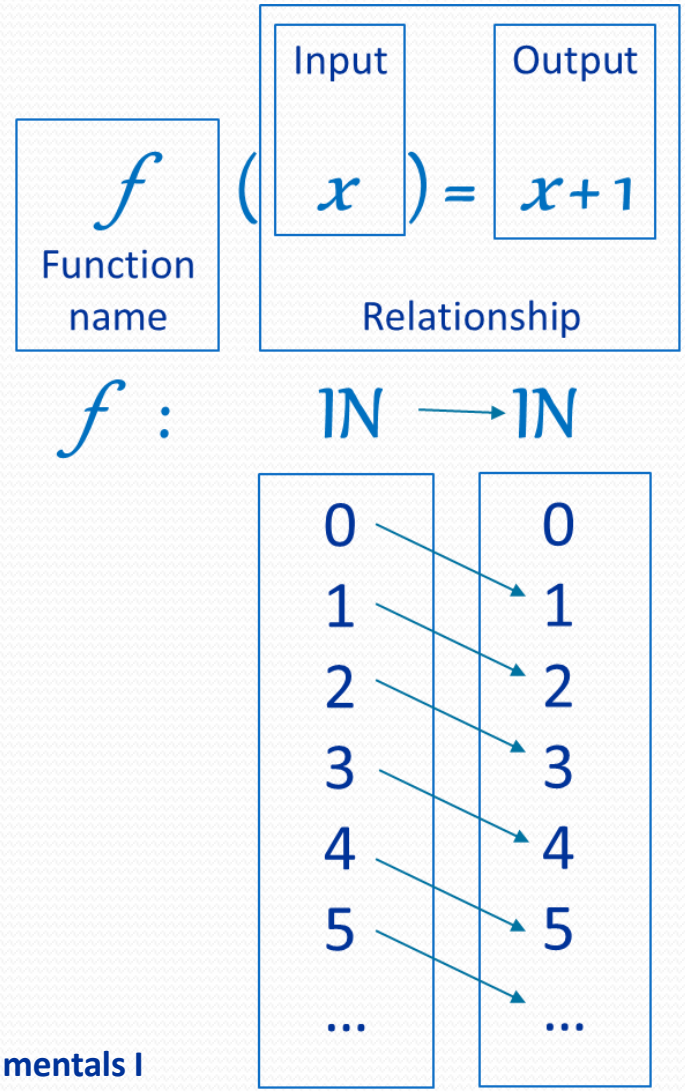
- **Functions** are like building blocks
- They allow complicated programs to be divided into manageable pieces
- Functions
  - Called modules
  - Like miniature programs
  - Can be put together to form a larger program
  - Can be called more than one time

Advantages

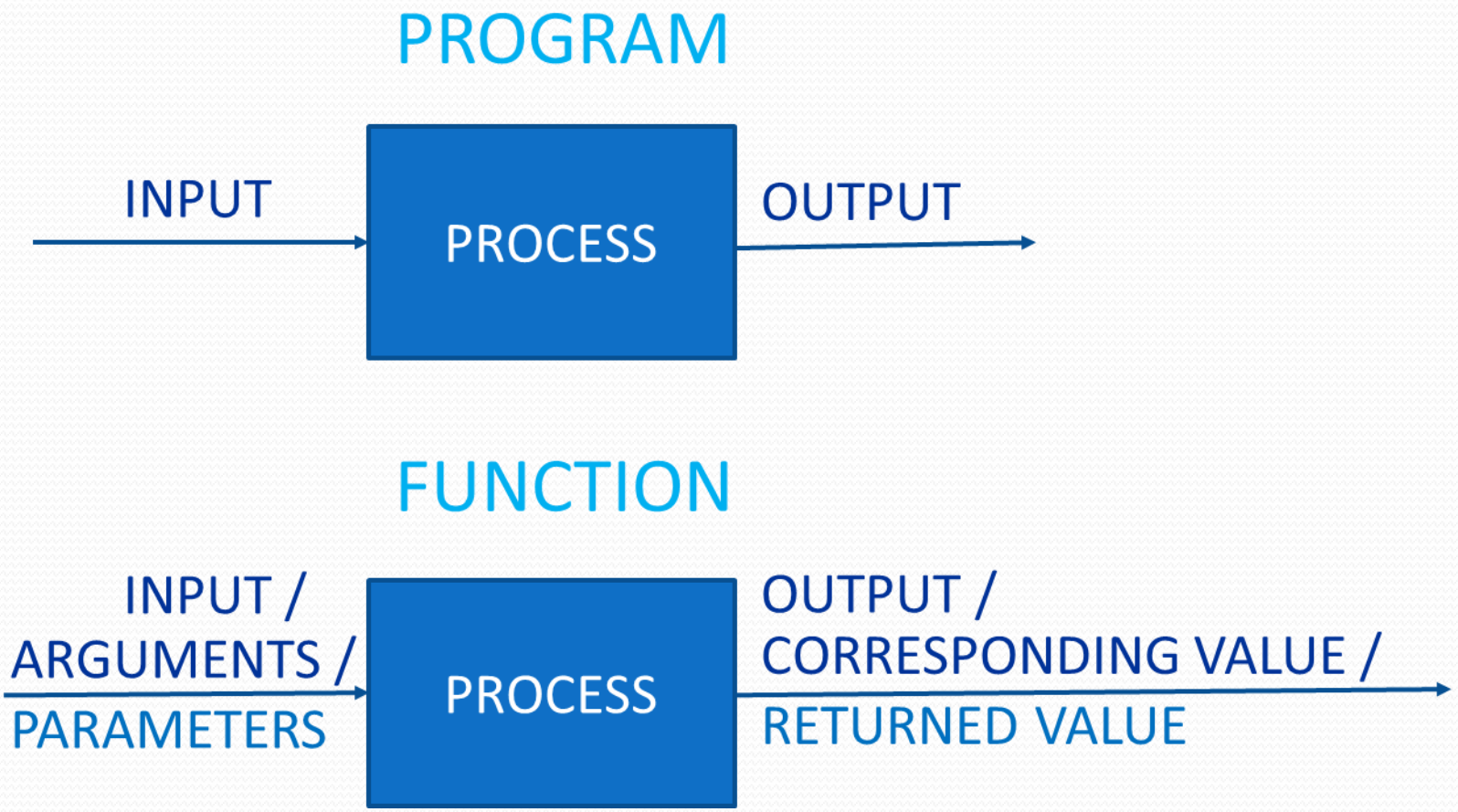
- A programmer can focus on just that part of the program and construct it, debug it, and perfect it
- Different people can work on different functions simultaneously
- Can be re-used (even in different programs)
- Enhance program readability

Functions in Math

- A function relates an input to an output.
- A function relates *each element* of a set with *exactly one* element of another set (possibly the same set).



Functions


PROGRAM
INPUT
PROCESS
OUTPUT
FUNCTION
INPUT /
ARGUMENTS /
PARAMETERS
PROCESS
OUTPUT /
CORRESPONDING VALUE /
RETURNED VALUE

Predefined Functions

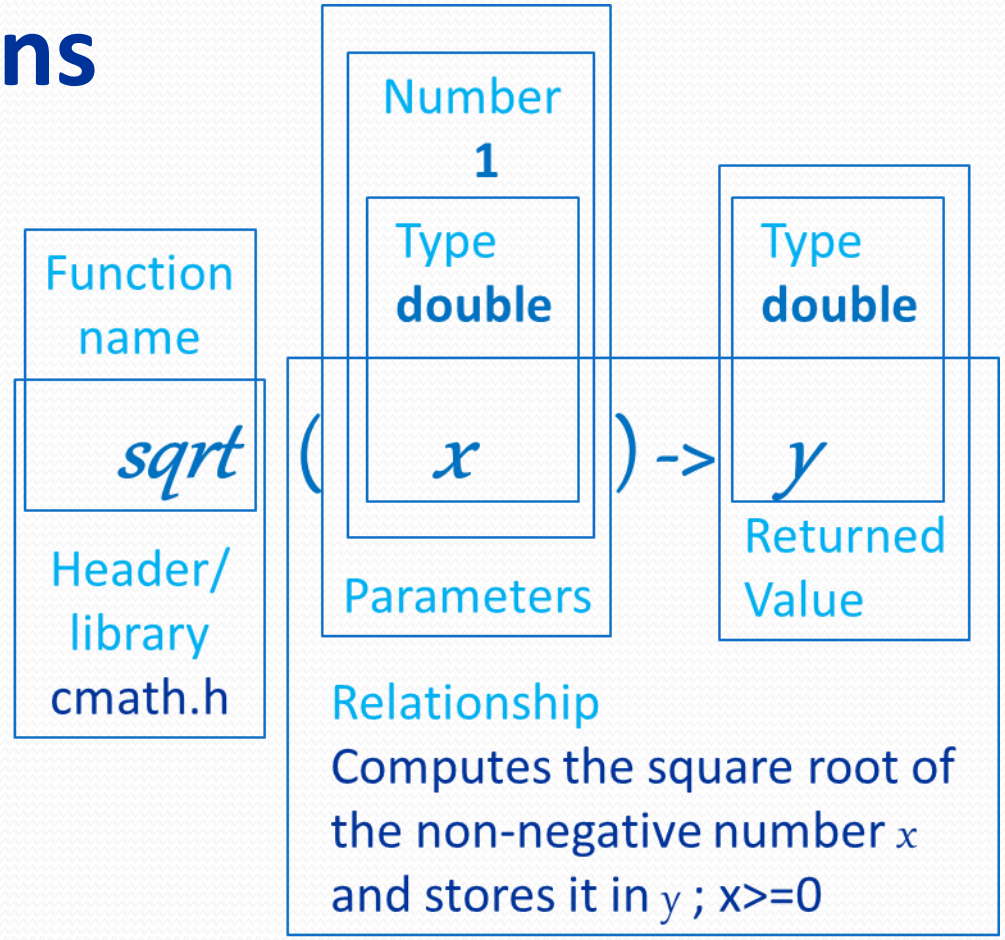
- Some of the predefined mathematical functions are:

```
sqrt (x)
pow (x,y)
floor (x)
```

- Predefined functions are organized into separate libraries
- I/O functions are in `iostream` header
- Math functions are in `cmath` header

Predefined Functions

- To use a predefined function, you need :
  - Function name
  - What the function does / relationship
  - Name of the header file
  - Number of parameters
  - Order of parameters
  - Type of each parameter
  - Returned value



$sqrt(4.0)$ -> **2.0**

$sqrt(16)$ -> **4.0**

Predefined Functions

Function	Header	Relationship/Purpose	Parameters Type	Returned Type	Example
pow(x,y)	cmath	calculates x^y	double, double	double	pow(2, 3) is 8.0
sqrt(x)	cmath	calculates the nonnegative square root of x; x >= 0.0	double	double	sqrt(2.25) is 1.5
floor(x)	cmath	calculates largest whole number not greater than x	double	double	floor(48.79) is 48.0
ceil(x)	cmath	returns the smallest integral value that is not less than x	double	double	ceil(48.79) is 49.0
abs(x)	cmath	Returns the absolute value of its argument	int	int	abs(-5) is 5
fabs(x)	cmath	Returns the absolute value of its argument	double	double	fabs(-4.5) is 4.5

Predefined Functions

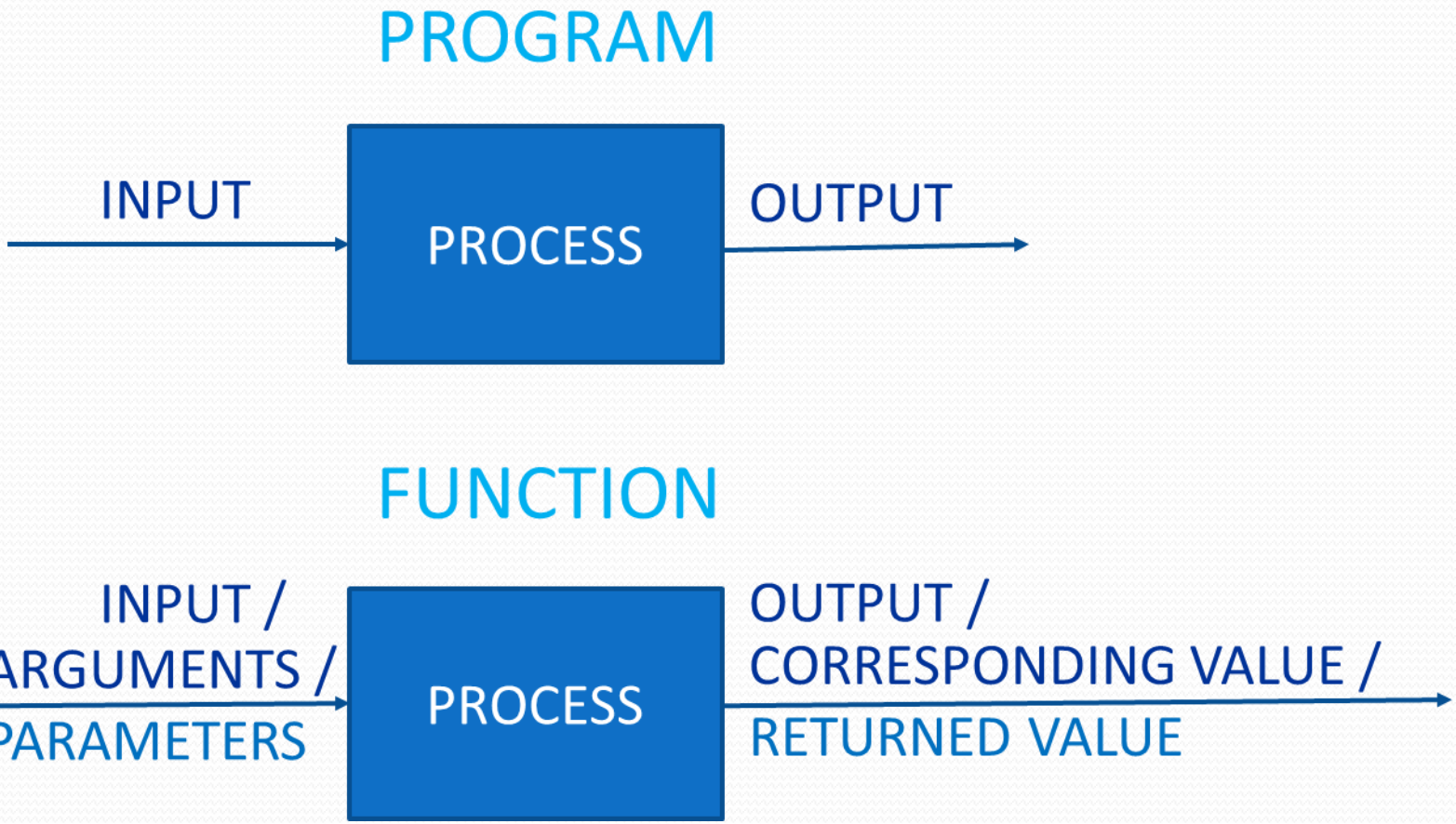
Function	Header	Relationship/Purpose	Param Type	Returned Type	Example
islower(x)	cctype	Returns true if x is a lowercase letter; otherwise, it returns false	int	int	islower('a') is true islower('A') is false islower('#') is false
isupper(x)	cctype	Returns true if x is a uppercase letter; otherwise, it returns false	int	int	isupper('A') is true isupper('a') is false isupper('1') is false
tolower(x)	cctype	Return the lowercase value of x if x is uppercase; otherwise, it returns x	int	int	tolower('A') is 97->'a' tolower('1') is 49->'1'
toupper(x)	cctype	Return the uppercase value of x if x is lowercase; otherwise, it returns x	int	int	toupper('a') is 65->'A' toupper('A') is 65->'A' tolower('1') is 49->'1'

Predefined Functions

```
#include <iostream>
#include <cmath>
#include <cctype>
#include <conio.h>

using namespace std;

int main()
{
     int    x;
     double u, v;

     cout << "Uppercase a is " << static_cast<char>(toupper('a')) << endl;

     u = 4.2;
     v = 3.0;
     cout << u << " to the power of " << v << " is " << pow(u, v) << endl;

     cout << "5.0 to the power of 4 is " << pow(5.0, 4) << endl;

     u = u + pow(3.0, 3);
     cout << "4.2 + pow(3.0, 3) is " << u << endl;

     x = -15;
     cout << "Absolute value of " << x << " is " << abs(x) << endl;

     _getch();

      return 0;
}
```

```
c:\Users\Adriana\Documents\Visual Studio 2008\P
Uppercase a is A
4.2 to the power of 3 is 74.088
5.0 to the power of 4 is 625
4.2 + pow(3.0, 3) is 31.2
Absolute value of -15 is 15
```

Functions and Programs

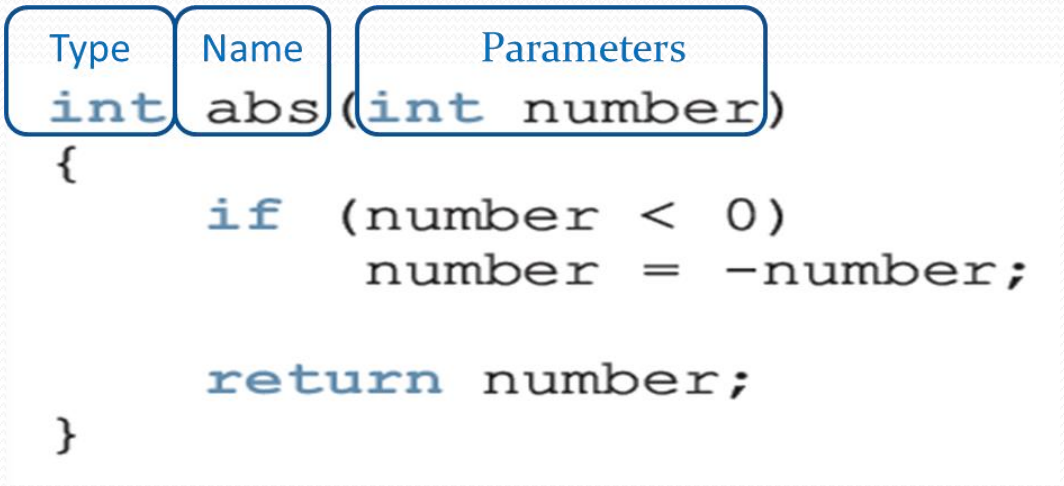


User-Defined Functions

- **Value-returning functions**: have a return type
  - Return a value of a specific data type using the `return` statement
- **Void functions**: do not have a return type
  - *Do not* use a `return` statement to return a value

Value-Returning Functions

- **Value-returning functions**: have a return type
  - Return a value of a specific data type using the `return` statement (called the type of the function)



Value-Returning Functions

Function Heading (header)

Formal Parameter: variable declared in the heading

```
int abs(int number)
{
        if (number < 0)
                number = -number;
        return number;
}
int main()
{
        int u = abs(-5);
        u = 5 + abs(u)
        cout << abs(x + 9);
        return 0;
}
```

Function Call

Actual Parameter: variable or expression listed in a call to a function

Value-Returning Function

- **Function definition syntax:**

```
FunctionType FunctionName(FormalParameterList)
{
      Statements
}
```

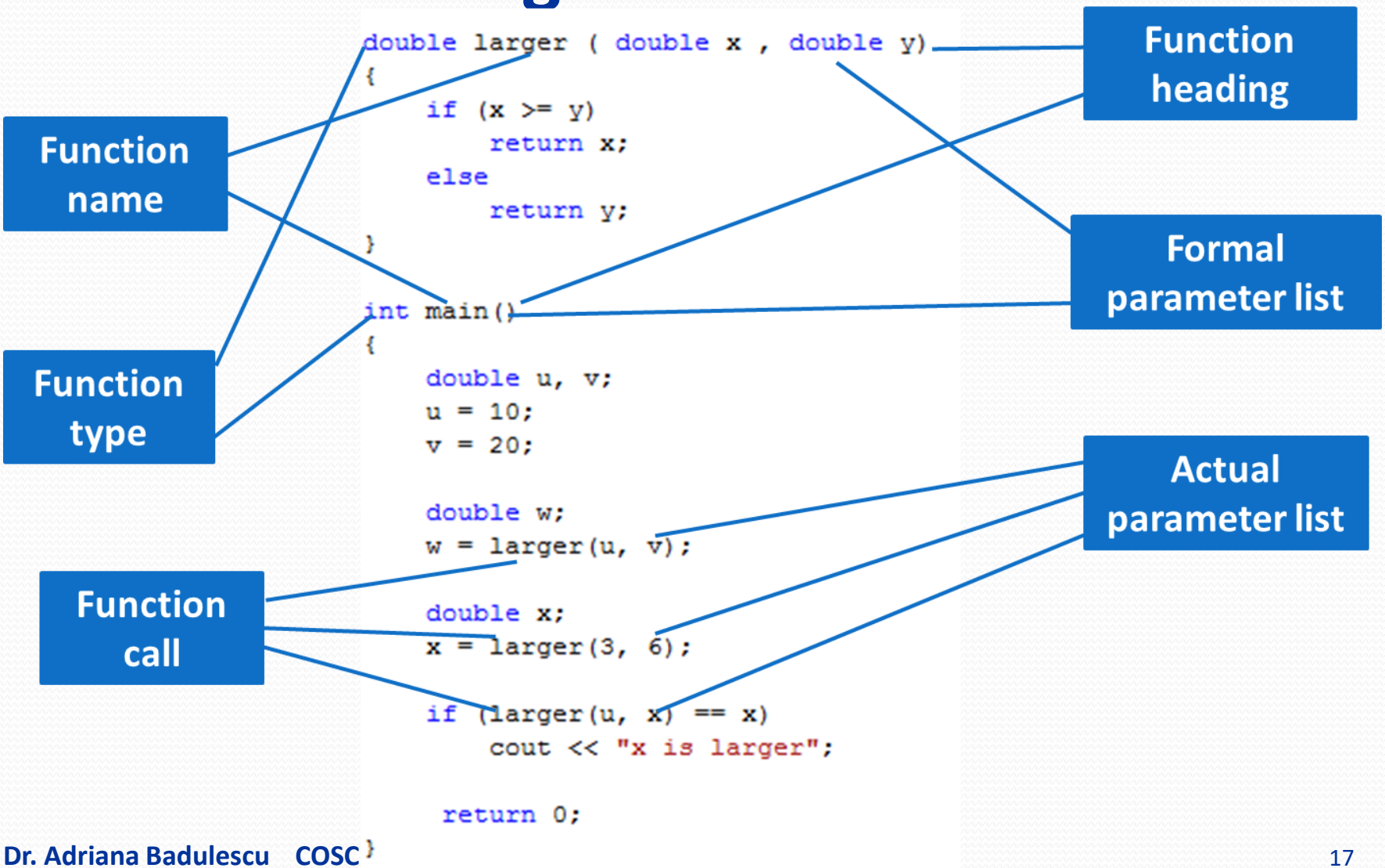
  - FunctionType - also called the data type or return type
  - FormalParameterList: DataType Parameter1, DataType Parameter2, …

- **Function call syntax:**

```
FunctionName(ActualParameterList)
```

  - ActualParameterList: ValueParameter1, ValueParameter2,…
  - ValueParameter: expresion or variable or constant

Value-Returning Function



Value-Returning Functions

- Because the value returned by a value-returning function is unique, must:
  - Save the value for further calculation
  - Use the value in some calculation
  - Print the value
- A value-returning function is used in an assignment or in an output statement
- One more thing is associated with functions:
  - The code required to accomplish the task

`return` Statement

- Once a value-returning function computes the value, the function returns this value via the `return` statement
- The `return` statement has the following syntax:

```
return expr;
```

- In C++, `return` is a reserved word
- When a return statement executes
  - Function immediately terminates
  - Control goes back to the caller
- When a `return` statement executes in the function `main`, the program terminates

return Statement

```
double larger(double x, double y)
{
    double max;

    if (x >= y)
        max = x;
    else
        max = y;

    return max;
}
```

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

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

    return y;
}
```

`return` Statement: Peculiarity

- A value returning function must return a value

```
int secret(int x)
{
     if (x > 5)
          return 2 * x;
}
```

- The correct definitions of the function are:

```
int secret(int x)
{
     if (x > 5)
          return 2 * x;

     return x;
}
```

```
int secret(int x)
{
     if (x > 5)
          return 2 * x;
     else
          return x;
}
```

`return` Statement: Peculiarity

- It is legal for a function to return more than one variable. Only the last one will be considered.

```
     return x, y;   //only the value of y will be returned

int funcRet1()
{
     int x = 45;

     return 23, x;  //only the value of x is returned
}

int funcRet2(int z)
{
     int a = 2;
     int b = 3;

     return 2 * a + b, z + b; //only the value of z + b is returned
}
```

Exercise: Enter an Integral Number

- Definition:

```
int EnterNumber()
{
        //declare N
        int N;
        //prompt user for number
        cout << "Enter a number for N: ";
        //read N
        cin >> N;
        //return the number
        return N;
}
```

- Call:

```
int N1=EnterNumber();
```

Exercise: Echo a Number

- Definition:

```
//Function to echo number
int EchoNumber(int N)
{
      //echo number
      cout << "You entered " << N;
      //return the number
      return N;
}
```

- Call:

```
int N = EchoNumber(N1);
```

Exercise: Next Number

- Definition:

```
int NextNumber(int N)
{
      return N + 1;
}
```

- Call:

```
int N2=NextNumber(N1);
```

Function Prototype

- **Function prototype**: function heading without the body of the function
- Syntax:

```
FunctionType FunctionName(FormalParameterList);
```

- It is not necessary to specify the variable name in the parameter list
- The data type of each parameter must be specified

Function Prototype

Without a Prototype

```
int EnterNumber()
{
        //declare N
        int N;
        //prompt user for number
        cout << "Enter a number: ";
        //read N
        cin >> N;
        //return the number
        return N;
}
int main()
{
        //declare N1
        int N1=EnterNumber();
       return 1;
}
```

With Prototype

```
int EnterNumber();
int main()
{
        //declare N1
        int N1=EnterNumber();
       return 1;
}
int EnterNumber()
{
        //declare N
        int N;
        //prompt user for number
        cout << "Enter a number:
";
        //read N
        cin >> N;
        //return the number
        return N;
```

Flow of Execution

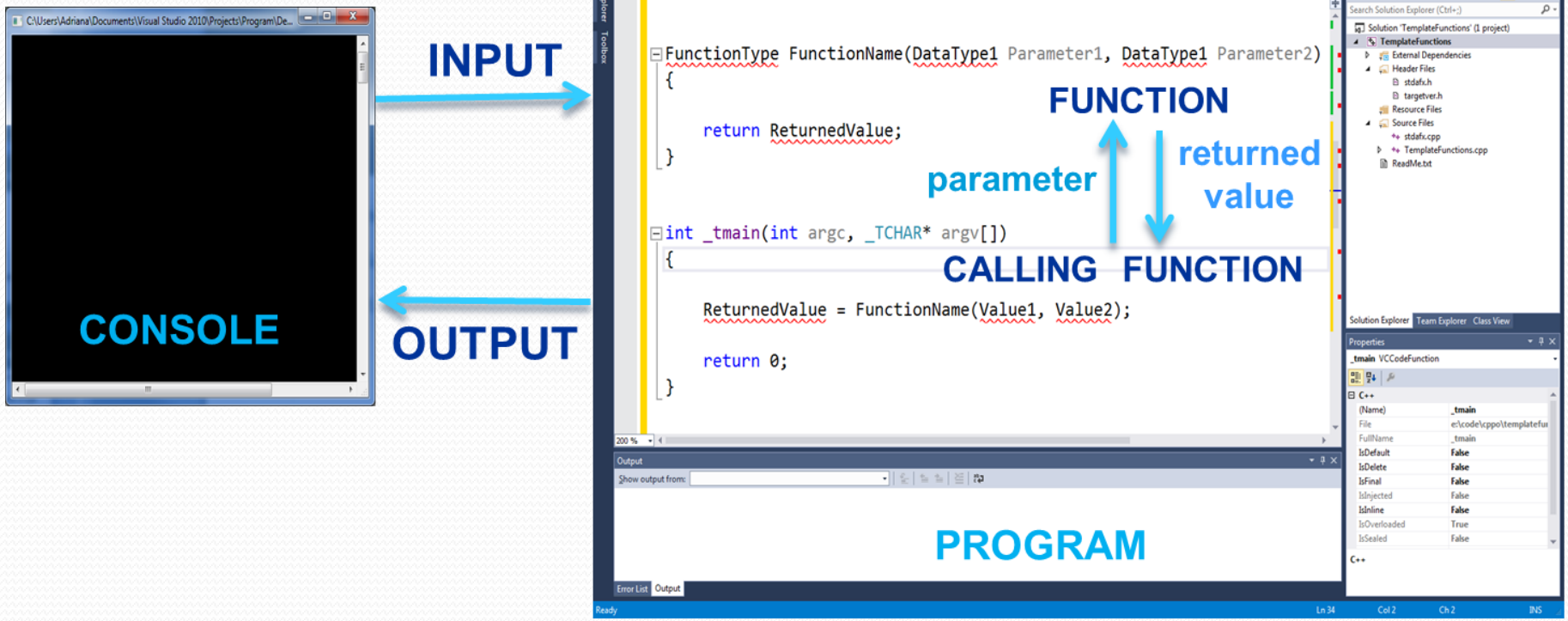
- Functions are **compiled** in the order they are defined in the source code
  - Function prototypes appear before any function definition
    - The compiler translates these first
    - The compiler can then correctly translate a function call
- **Execution** always begins at the first statement in the function `main`
- Other functions are executed only when they are called

Flow of Execution

- A function call results in transfer of control to the first statement in the body of the called function
- After the last statement of a function is executed, control is passed back to the point immediately following the function call
- A value-returning function returns a value
  - After executing the function the returned value replaces the function call statement

Value-Returning Functions

```
FunctionType FunctionName (Type Param1, Type Param2)
{ //code
return ReturnedValue;
}
```




```
ReturnedValue=FunctionName(Value1, Value2);
```

Void Function

- **Function definition syntax:**

```
void FunctionName(FormalParameterList)
{
      Statements
}
```

  - FunctionType - **also called the data type or return type**
  - **FormalParameterList:** DataType Parameter1, DataType Parameter2, …

- **Function call syntax:**

```
FunctionName(ActualParameterList)
```

  - ActualParameterList: ValueParameter1, ValueParameter2, …
  - ValueParameter: expresion or variable or constant

Value-Returning Functions vs. Void Functions

Value-Returning Function

```
FunctionType FunctionName(
      DataType Parameter1,
      DataType Parameter2)
    {
          statements
          return Value;
    }
```

```
ReturnedValue=FunctionName(
      Value1,
      Value2);
```

Void Function

```
void FunctionName(
      DataType Parameter1,
      DataType Parameter2)
    {
          statements
          return;
    }
```

```
FunctionName(
      Value1,
      Value2);
```

Void Functions

- Void functions and value-returning functions have similar features and structures
  - Both have a heading part and a body part
  - Both can be placed either before or after the function `main`
    - If they are placed after the function `main`, the function prototype must be placed before the function `main`
  - Formal parameters are optional
- Void functions and value-returning functions have different features
  - A void function does not have a return type
    - `return` statement without any value is typically used to exit function early
  - A call to a void function is a stand-alone statement

Void Function

Print the numbers between 1 and 10

```
int PRINT ()
{
    int k;
    for (k=1;k<=10;k++)
        cout << k << " ";
    return 0;
}
int main()
{
    int x=PRINT();
}
```

Does the value matter? Are we going to use the returned value?

```
void PRINT ()
{
    int k;
    for (k=1;k<=10;k++)
        cout << k << " ";
    return;
} …
int main()
{
    PRINT();
}
```

If the functions does not need to return a value, you should use a void function

Exercise: Print Integral Number

- Definition

```
//Function to print an integral number N
void PrintNumber(int N)
{
      //output the number N
      cout << "\nThe number is " << N;
}
```

- Call

```
PrintNumber(N1);

PrintNumber(9);

PrintNumber(N1 + 10);
```

Exercise: Sum Numbers

- Definition

```
//Function to sum 2 numbers M and N
int SumNumbers(int M, int N)
{
      return M + N;
}
```

- Call

```
int sum=SumNumbers(N1, N2);
```

Exercise: Print All Numbers in Range

- Definition

```
//Function to print the numbers between 1 and 10
void PrintNumbers()
{
        cout << "\nThe numbers between 1 and 10 are: ";
        int k;
        for (k = 1; k <= 10; k++)
                cout << k<<" ";
}
```

- Call

```
PrintNumbers();
```

Exercise: Print Even Numbers in Range

- Definition

```
//Function to print the even numbers between 1 and 10
void PrintEvenNumbers()
{
	cout << "\nThe numbers even between 1 and 10 are:
";
	int k;
	for (k = 1; k <= 10; k++)
		if (k%2==0)
			cout << k << " ";
}
```

- Call

```
PrintEvenNumbers();
```

Exercise: Print Odd Numbers in Range

- Definition

```
//Function to print all the odd numbers between 1 and 10
void PrintOddNumbers()
{
        cout << "\nThe odd numbers between 1 and 10 are: ";
        int k;
        for (k = 1; k <= 10; k = k + 2)
        cout << k << " ";
}
```

- Call

```
PrintOddNumbers();
```

Exercise: Print All Numbers in Range

- Definition

```
//Function to print the numbers between 1 and N
void PrintNumbers(int N)
{
     cout<< "\nThe numbers between 1 and " << N << " are: ";
     int k;
     if (N >= 1)
      for (k = 1; k <= N; k++)
            cout << k << " ";
     else
      for (k = 1; k >= N; k--)
            cout << k << " ";
}
```

- Call

```
PrintNumbers(N1);
```

Exercise: Print All Numbers in Range

- Definition

```
//Function to print the numbers between A and B
void PrintNumbers(int A, int B)
{
    cout << "\nThe numbers between " <<A<<" and " << B << " are: ";
    int k;
    if (A<=B)
        for (k = A; k <= B; k++)
            cout << k << " ";
    else
        for (k = A; k >= B; k--)
            cout << k << " ";
}
```

- Call

```
PrintNumbers(N1,N2);
PrintNumbers(1, 10);
PrintNumbers(10, 1);
PrintNumbers(N1, -10);
PrintNumbers(1, N2);
```

Exercise: Check if a Number is in Range

- Definition

```
//Function to check if a number N is between A and B
//and return true if it is and false if it does not
bool Between(int N, int A, int B)
{
        if (A <= B)
                if ((A <= N) && (N <= B))
                        return true;
                else
                        return false;
        else //B<A
                if ((B <= N) && (N <= A))
                        return true;
                else
                        return false;
}
```

- Call

```
cout << "\n" << Between(5, 1, 10);
cout << "\n" << Between(-5, 1, 10);
cout << "\n" << Between(5, 10, 1);
cout << "\n" << Between(-5, 10, 1);
```

Exercise: Enter a valid Number in Range

- Definition

```
//Function to enter a valid number between A and B
int EnterNumber(int A, int B)
{
     //declare N
     int N;
     do
     {
          //prompt user for number
          cout << "\nEnter a number between " << A << " and " << B <<": ";
          //read N
          cin >> N;
          //validate the number
          if (Between(N, A, B) == true)
               //return the number
               return N;
          else
               cout << "\nThe number "<<N<<" is in range! Try again!";
     }
     while (Between(N, A, B) == false);
     return 0;
}
```

- Call `int N4 = EnterNumber(1, 10);`

Value-Returning Functions

Minimum of 2 integers

```
int MIN (int x, int y)
{
      int z;
      if (x<=y)
            z=x;
      else
            z=y;
      return z;
}
int main()
{
      cout << MIN(3,9);
}
```

Maximum of 2 integers

```
int MAX (int x, int y)
{
      int z;
      if (x<=y)
            z=y;
      else
            z=x;
      return z;
}
int main()
{
      cout << MAX(3,9);
}
```

Value-Returning Functions

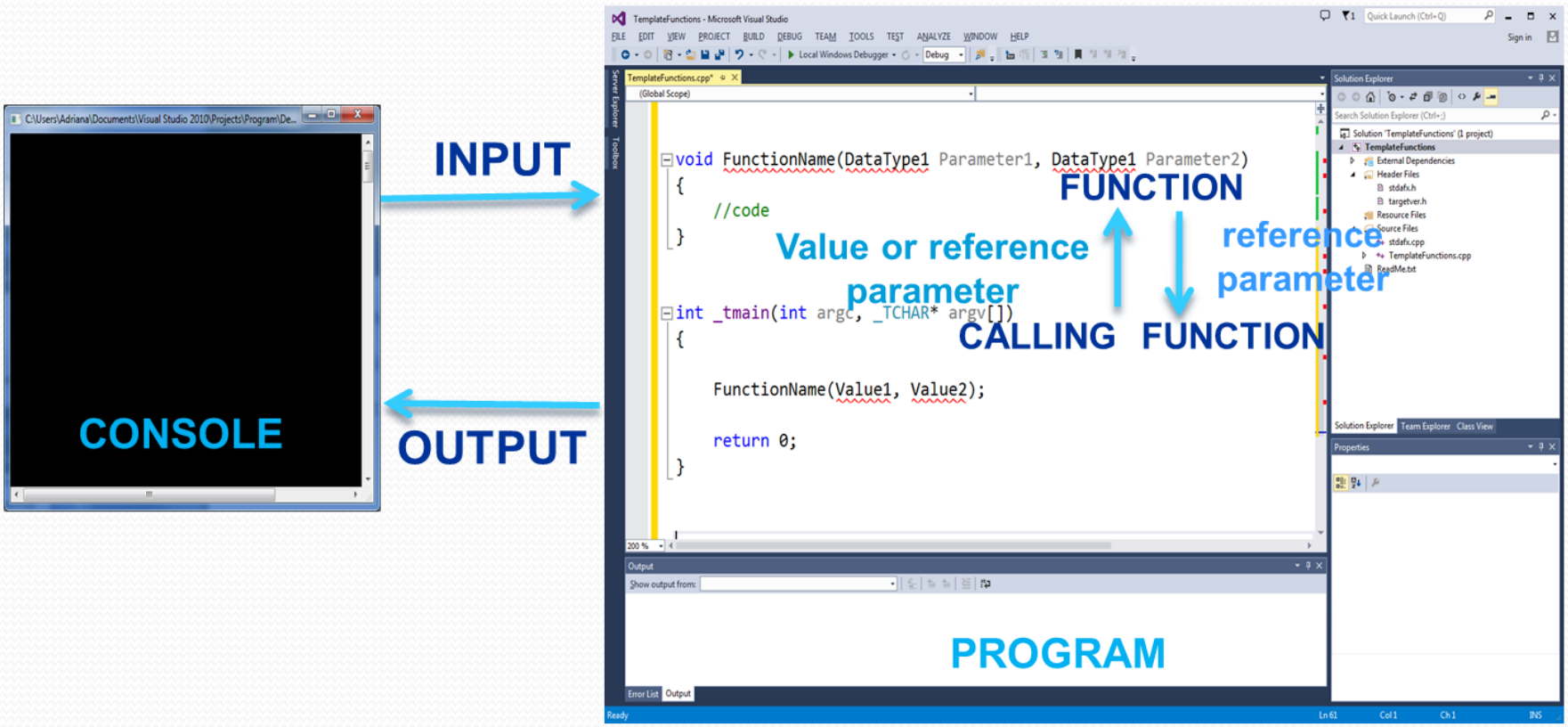
Minimum and maximum of 2 integers (one function)

```
int MINMAX (int x, int y)
{
    int min,max;
    if (x<=y)
    {
        min=x;
        max=y;
    }
    else
    {
        min=y;
        max=x;
    }
    return ? ;
}
```

Can return only one value! Not 2 or more!

Void Functions

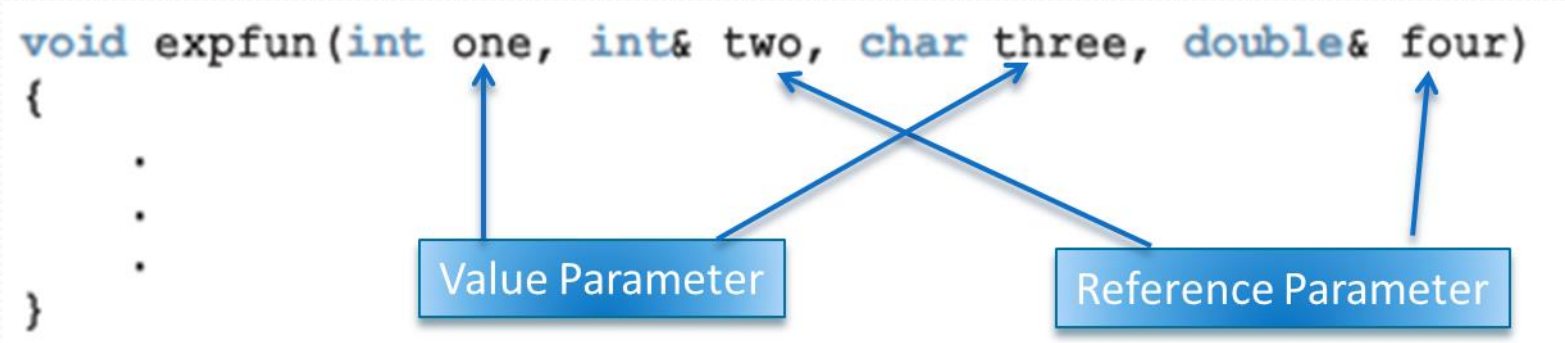
```
void FunctionName (Type Param1, Type &Param2)
{ //code
}
```




```
FunctionName(Value1, Value2);
```

Types of Parameters

- **Value parameter**: a formal parameter that receives a copy of the content of corresponding actual parameter
- **Reference parameter**: a formal parameter that receives the location (memory address) of the corresponding actual parameter

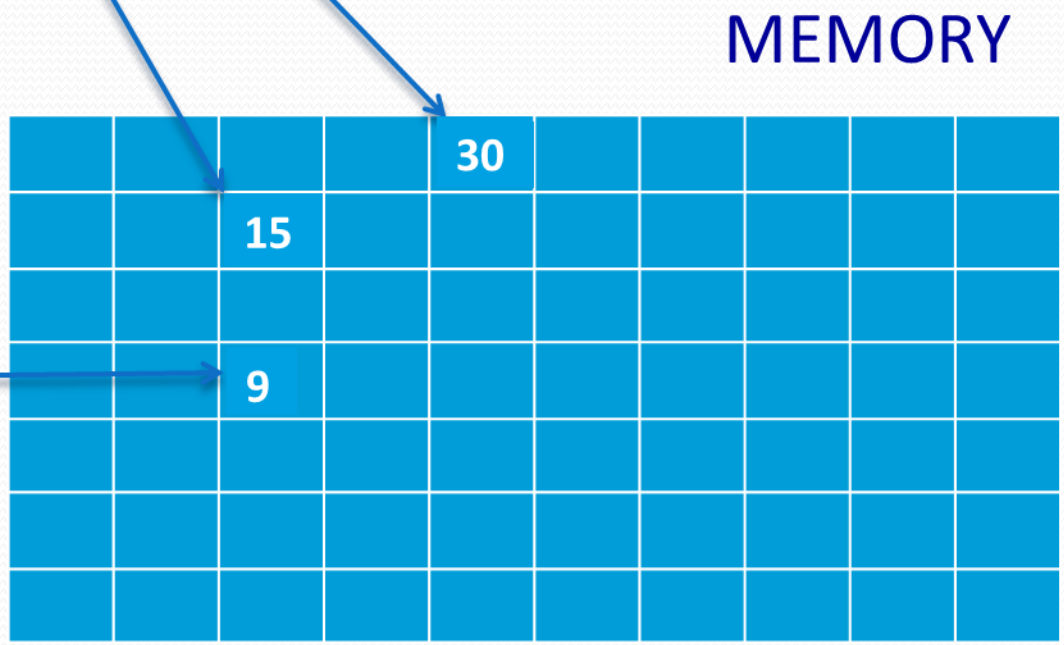


Value Parameters

- If a formal parameter is a value parameter, the value of the corresponding actual parameter is copied into it
- The value parameter has its own copy of the data
- During program execution, the value parameter manipulates the data stored in its own memory space
- If the function changes the value of a value parameter, it will change the copy of the actual parameter and will not change the actual parameter
- The actual parameter can be an expression, variable, or constant

Value Parameters

```
void FUNCTION(int x, int y)
{
     x=15;
     y=30;
}
int main()
{
     int z=9;
     FUNCTION(z,3);
     cout <<z;
}
```

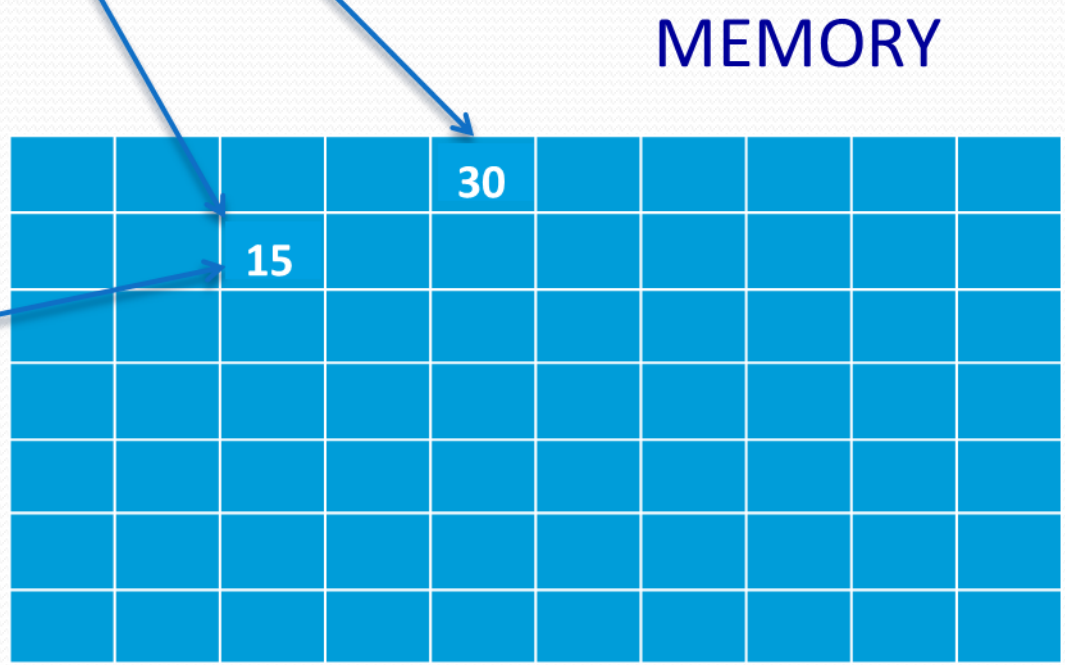


Reference Variables as Parameters

- If a formal parameter is a reference parameter, it receives the memory address of the corresponding actual parameter
- A reference parameter stores the address of the corresponding actual parameter
- The address stored in the reference parameter directs it to the memory space of the corresponding actual parameter
- If the function changes the value of a value parameter (the memory), it will change the value from that memory address and therefore it will change the actual parameter
- The actual parameter should be a variable

Reference Parameters

```
void FUNCTION(int &x, int y)
{
    x=15;
    y=30;
}
int main()
{
    int z=9;
    FUNCTION(z,3);
    cout <<z;
}
```



Reference Variables as Parameters

- Reference parameters are useful in three **situations**:
  - Returning more than one value
  - Changing the actual parameter
  - When passing the address would save memory space and time

Value-Returning vs. Void Functions

Value-Returning Function

```
int MIN (int x, int y)
{
      int z;
      if (x<=y)
            z=x;
      else
            z=y;
      return z;
}
 int main()
{
      int t;
      t=MIN(3,9);
}
```

Local variable

Returned value

Local variable (returned value)

Void Function

```
void MIN(int x, int y, int &z)
{

      if (x<=y)
            z=x;
      else
            z=y;
}
int main()
{
      int t;
      MIN(3,9,t);
}
```

Local variable (reference parameter)

Local variable (reference parameter)

Value-Returning Functions

Minimum and maximum of 2 integers (one function)

```
int MINMAX (int x, int y)
{
    int min,max;
    if (x<=y)
    {
        min=x;
        max=y;
    }
    else
    {
        min=y;
        max=x;
    }
    return ? ;
}
```

Can return only one value! Not 2 or more!

Void Functions

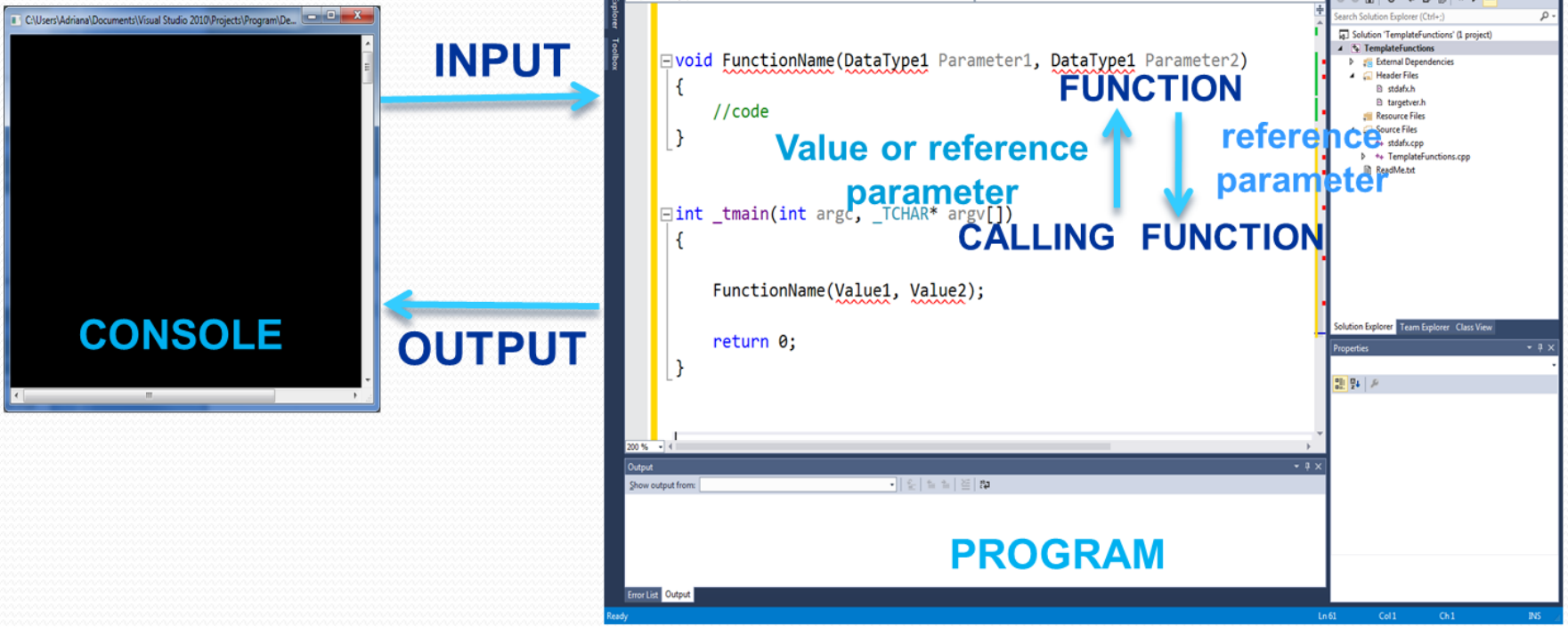
Minimum and maximum of 2 integers (one function)

```
void MINMAX (int x, int y, int &min, int &max)
{
    if (x<=y)
    {
        min=x;
        max=y;
    }
    else
    {
        min=y;
        max=x;
    }
}
```

Can change the actual values! Thus you can "return" more than one value from a function!

Void Functions

```
void FunctionName (Type Param1, Type &Param2)
{ //code
}
```




```
FunctionName(Value1, Value2);
```

Exercise: Enter Number using a Reference

- Declare

```
//Function to enter a number via a reference
void EnterNumber(int &N)
{
        //INPUT the number N
        //prompt user for number
        cout << "\n\nEnter a number: ";
        //read N
        cin >> N;
}
```

- Call

```
int N3;
EnterNumber(N3);
```

Value and Reference Parameters and Memory Allocation

- When a function is **called**
  - Memory for its formal parameters and variables declared in the body of the function (called **local variables**) is allocated in the function data area
- In the case of a **value parameter**
  - The value of the actual parameter is copied into the memory cell of its corresponding formal parameter

Value and Reference Parameters and Memory Allocation

- In the case of a **reference parameter**
  - The address of the actual parameter passes to the formal parameter
- Content of formal parameter is an address
- During execution, changes made by the formal parameter permanently change the value of the actual parameter
- Stream variables (e.g., `ifstream`) should be passed by reference to a function

Reference Parameters and Value-Returning Functions

- You can also use reference parameters in a value-returning function
  - Not recommended
- By definition, a value-returning function returns a single value
  - This value is returned via the return statement
- If a function needs to return more than one value, you should change it to a void function and use the appropriate reference parameters to return the values

Reference Parameters and Value-Returning Functions

With Reference

- Declare

```
//Function to enter a number via a reference
void EnterNumber(int &N)
{
        //INPUT the number N
        //prompt user for number
        cout << "Enter a number:
";
        //read N
        cin >> N;
}
```

- Call

```
int N3;
EnterNumber(N3);
```

With Value-Returning Function

- Definition:

```
int EnterNumber()
{
        //declare N
        int N;
        //prompt user for number
        cout << "Enter a number:
";
        //read N
        cin >> N;
        //return the number
        return N;
}
```

- Call:

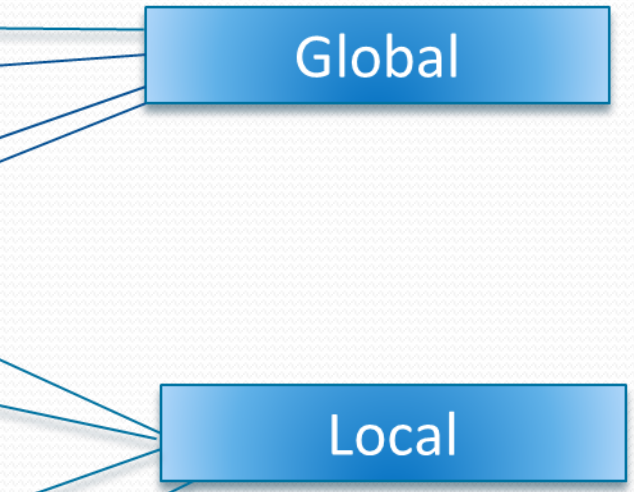
```
int N1=EnterNumber();
```

Scope of an Identifier

- The **scope of an identifier** refers to where in the program an identifier is accessible
- **Global identifier**: identifiers declared outside of every function definition; accessible from anywhere in the code
  - Functions are global identifiers
    - The definition of one function cannot be included in the body of another function
- **Local identifier**: identifiers declared within a function (or block); accessible only from that function (or block)
  - Parameters of a function
  - Variable declared within a function or block
  - Local identifier are accessible in the nested blocks if there is no variable with the same name in the nested block

Scope of an Identifier

```
const int n = 3;
double z;

void myfunction(int m)
{
    int j;
    j = n + m;
}

int i;

int main()
{
    int j, k;
    j = 2;
    k = i + j;
    {
        j++;
        double k;
        k++;
    }
    myfunction(k);
    return 0;
}
```



Scope of an Identifier

- Local variable can be redeclared in nested block
- The operator : : is called the **scope resolution operator**
  - A global variable declared before the definition of a function (block) can be accessed by the function (or block)
  - Even if the function (or block) has an identifier with the same name as the variable

```
int i = 1;

int main()
{
    cout << i << endl;
    int i = 5;
    cout << i << endl;
    {
        int i = 7;
        cout << i << endl;
    }
    cout << i << endl;
    cout << ::i << endl;

    return 0;
}
```

Global Variables, Named Constants, and Side Effects

- Using **global identifiers** causes side effects
  - A function that uses global variables is not independent
  - If more than one function uses the same global variable and something goes wrong
    - It is difficult to find what went wrong and where
    - Problems caused in one area of the program may appear to be from another area
- Avoid using global identifiers

Avoid Global Variables?

- Declare them in made and add them as parameters to functions (reference or value – depending if you need the value to change or not)

```
int x;
void Initialize()
{
     x = 0;
}
void Print()
{
     cout << x;
}
int main()
{
    Initialize();
    cin >> x;
    Print();
}
```

Global variable

```
void Initialize(int &x)
{
     x = 0;
}
void Print(int x)
{
     cout << x;
}
int main()
{
    int x;
    Initialize(x);
    cin >> x;
    Print(x);
}
```

Reference Parameter (can change value)

Value Parameter (cannot change value)

Local variable

Static and Automatic Variables

- **Automatic variable**: memory is allocated at block entry and deallocated at block exit
  - Variables declared within a block are automatic variables
- **Static variable**: memory remains allocated as long as the program executes
  - Variables declared outside of any block are static variables
  - Declare a static variable within a block by using the reserved word `static`

```
static DataType  identifier;
```

```
void func()
{
    int x = 0;
    static int y = 0;
    x += 2;
    y += 2;
    cout << x << " " << y << endl;
}

int main()
{
    int i;
    for (i = 0; i<5; i++)
        func();

    return 0;
}
```

```
2 2
2 4
2 6
2 8
2 10
```

Function Overloading

- In a C++ program, several functions can have the same name - this is called **function overloading** or **overloading a function name**
- Two functions can be **overloaded** (have the same name) if they have **different formal parameter lists (signatures)** meaning:
  - A different number of formal parameters, or
  - If the number of formal parameters is the same, then the data type of the formal parameters, in the order you list them, must differ in at least one position

```
int F()
int F(int x)
int F(double y)
int F(int x, int y)
int F(int x, double y)
```

Can overload

```
int  F(int y, double t)
void F(int z, double t)
```

Cannot overload

Functions with Default Parameters

- In a function call, the number of actual and formal parameters must be the same, unless the function has default parameters
- In the function with default parameters, you can omit trailing actual parameters in the function call
- If you do not specify the value of a default parameter, the default value is used
- All default parameters must be the rightmost parameters of the function

```
int F(int n, double d=3.5, char c='k', bool b=false);
```

```
int i;
F(i, 4.5, 'A', true);
F(3, 9);
F(5);
```

Functions with Default Parameters

- If you omit the actual value for a default parameter, you must omit all of the arguments to its right
- Default values can be constants, global variables, or function calls
- Reference parameters cannot have a default value

```
int F(int n, double d=3.5, char c='k', bool b=false);
```

```
    F(3, 9);
```

```
    F(3, 'A');
```

```
int F1(int n, double d=3.5, char c, bool b=false);

int F2(int n, double d=3.5, char &c='k', bool b= false);
```

Summary

- **Functions** are miniature programs
- Functions divide a program into manageable tasks
- C++ provides the **standard/predefined functions**
- Two types of user-defined functions: value-returning functions and void functions
- Variables defined in a function heading are called **formal parameters**
- Expressions, variables, or constant values in a function call are called **actual parameters**
- In a function call, the number of actual parameters and their types must match with the formal parameters in the order given

Summary

- Function heading and the body of the function are called the **definition** of the function
- A **value-returning function** returns its value via the return statement
- A **prototype** is the function heading without the body of the function; prototypes end with the semicolon
- Prototypes are placed before every function definition, including main
- User-defined functions execute only when they are called
- In a function call statement, specify only the actual parameters, not their data types

Summary

- **Void function**: does not have a data type
  - A return statement without any value can be used in a void function to exit it early
  - The heading starts with the word void
  - To call the function, you use the function name together with the actual parameters in a stand-alone statement
- Two types of **formal parameters**:
  - Value parameters
  - Reference parameters
- A **reference parameter** receives the memory address of its corresponding actual parameter
  - If a formal parameter needs to change the value of an actual parameter, you must declare this formal parameter as a reference parameter in the function heading
- A **value parameter** receives a copy of its corresponding actual parameter

Summary

- Variables declared within a function (or block) are called **local variables**
- Variables declared outside of every function definition (and block) are **global variables**
- **Automatic variable**: variable for which memory is allocated on function/block entry and deallocated on function/block exit
- **Static variable**: memory remains allocated throughout the execution of the program
- C++ functions can have **default parameters**
- Functions can be **overloaded** if they have different signatures