

# C++ FUNCTIONS

# AGENDA

- ◉ What is a function?
- ◉ Types of C++ functions:
  - Standard functions
  - User-defined functions
- ◉ C++ function structure
  - Function signature
  - Function body
- ◉ Declaring and Implementing C++ functions
- ◉ Sharing data among functions through function parameters
  - Value parameters
  - Reference parameters
  - Const reference parameters
- ◉ Scope of variables
  - Local Variables
  - Global variable

# FUNCTIONS AND SUBPROGRAMS

- The Top-down design approach is based on dividing the main problem into smaller tasks which may be divided into simpler tasks, then implementing each simple task by a subprogram or a function
- A C++ function or a subprogram is simply a chunk of C++ code that has
  - A descriptive function name, e.g.
    - *computeTaxes* to compute the taxes for an employee
    - *isPrime* to check whether or not a number is a prime number
  - A returning value
    - The *computeTaxes* function may return with a double number representing the amount of taxes
    - The *isPrime* function may return with a Boolean value (true or false)

# C++ STANDARD FUNCTIONS

- ⦿ C++ language is shipped with a lot of functions which are known as standard functions
- ⦿ These standard functions are groups in different libraries which can be included in the C++ program, e.g.
  - Math functions are declared in `<math.h>` library
  - Character-manipulation functions are declared in `<ctype.h>` library
  - C++ is shipped with more than 100 standard libraries, some of them are very popular such as `<iostream.h>` and `<stdlib.h>`, others are very specific to certain hardware platform, e.g. `<limits.h>` and `<largeInt.h>`

# EXAMPLE OF USING STANDARD C++ MATH FUNCTIONS

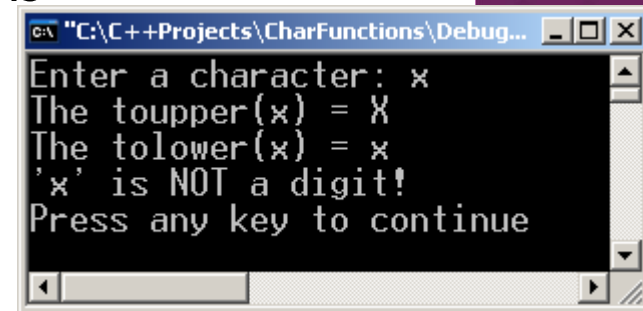
```
#include <iostream.h>
#include <math.h>
void main()
{
    // Getting a double value
    double x;
    cout << "Please enter a real number: ";
    cin >> x;
    // Compute the ceiling and the floor of the real
    number
    cout << "The ceil(" << x << ") = " << ceil(x) <<
    endl;
    cout << "The floor(" << x << ") = " << floor(x) <<
    endl;
}
```

# EXAMPLE OF USING STANDARD C++ CHARACTER FUNCTIONS

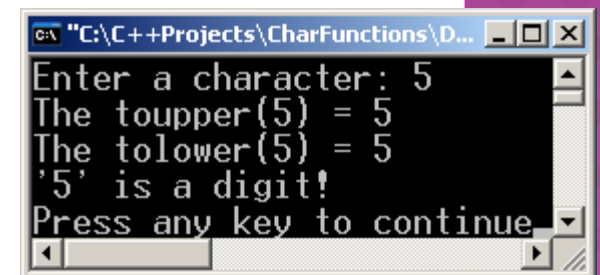
```
#include <iostream.h> // input/output handling
#include <ctype.h> // character type functions
void main()
```

```
{
    char ch;
    cout << "Enter a character: ";
    cin >> ch;
    cout << "The toupper(" << ch << ") = " << (char) toupper(ch) << endl;
    cout << "The tolower(" << ch << ") = " << (char) tolower(ch) << endl;
    if (isdigit(ch))
        cout << "" << ch << "" is a digit!\n";
    else
        cout << "" << ch << "" is NOT a digit!\n";
}
```

Explicit casting



```
G:\C++Projects\CharFunctions\Debug...
Enter a character: x
The toupper(x) = X
The tolower(x) = x
'x' is NOT a digit!
Press any key to continue
```



```
G:\C++Projects\CharFunctions\D...
Enter a character: 5
The toupper(5) = 5
The tolower(5) = 5
'5' is a digit!
Press any key to continue
```

# USER-DEFINED C++ FUNCTIONS

- ⦿ Although C++ is shipped with a lot of standard functions, these functions are not enough for all users, therefore, C++ provides its users with a way to define their own functions (or user-defined function)
- ⦿ For example, the `<math.h>` library does not include a standard function that allows users to round a real number to the  $i^{\text{th}}$  digits, therefore, we must declare and implement this function ourselves

# HOW TO DEFINE A C++ FUNCTION?

- Generally speaking, we define a C++ function in two steps (preferably but not mandatory)
  - Step #1 - declare the *function signature* in either a header file (.h file) or before the main function of the program
  - Step #2 - Implement the function in either an implementation file (.cpp) or after the main function



# WHAT IS THE SYNTACTIC STRUCTURE OF A C++ FUNCTION?

- ⦿ A C++ function consists of two parts
  - The function header, and
  - The function body
- ⦿ The function header has the following syntax  
*<return value> <name> (<parameter list>)*
- ⦿ The function body is simply a C++ code enclosed between { }

# EXAMPLE OF USER-DEFINED C++ FUNCTION

```
double computeTax(double income)
{
    if (income < 5000.0) return 0.0;
    double taxes = 0.07 * (income-5000.0);
    return taxes;
}
```

# EXAMPLE OF USER-DEFINED C++ FUNCTION

```
double computeTax(double income)
```

Function  
header

```
{  
    if (income < 5000.0) return 0.0;  
    double taxes = 0.07 * (income-5000.0);  
    return taxes;  
}
```

# EXAMPLE OF USER-DEFINED C++ FUNCTION

Function  
header

Function  
body

```
double computeTax(double income)
```

```
{  
    if (income < 5000.0) return 0.0;  
    double taxes = 0.07 * (income-5000.0);  
    return taxes;  
}
```

# FUNCTION SIGNATURE

- ◉ The function signature is actually similar to the function header except in two aspects:
  - The parameters' names may not be specified in the function signature
  - The function signature must be ended by a semicolon

- ◉ Example

```
double computeTaxes(double) ;
```

Unnamed  
Parameter

Semicolon  
;

# WHY DO WE NEED FUNCTION SIGNATURE?

## ⦿ For Information Hiding

- If you want to create your own library and share it with your customers without letting them know the implementation details, you should declare all the function signatures in a header (.h) file and distribute the binary code of the implementation file

## ⦿ For Function Abstraction

- By only sharing the function signatures, we have the liberty to change the implementation details from time to time to
  - Improve function performance
  - make the customers focus on the purpose of the function, not its implementation

# EXAMPLE

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

// Function Signature
double getIncome(string);
double computeTaxes(double);
void printTaxes(double);

void main()
{
    // Get the income;
    double income = getIncome("Please enter
the employee income: ");

    // Compute Taxes
    double taxes = computeTaxes(income);

    // Print employee taxes
    printTaxes(taxes);
}
```

```
double computeTaxes(double income)
{
    if (income<5000) return 0.0;
    return 0.07*(income-5000.0);
}

double getIncome(string prompt)
{
    cout << prompt;
    double income;
    cin >> income;
    return income;
}

void printTaxes(double taxes)
{
    cout << "The taxes is $" << taxes << endl;
}
```

# BUILDING YOUR LIBRARIES

- ◉ It is a good practice to build libraries to be used by you and your customers
- ◉ In order to build C++ libraries, you should be familiar with
  - How to create header files to store function signatures
  - How to create implementation files to store function implementations
  - How to include the header file to your program to use your user-defined functions



# C++ HEADER FILES

- ⦿ The C++ header files must have .h extension and should have the following structure
  - #ifndef compiler directive
  - #define compiler directive
  - May include some other header files
  - All functions signatures with some comments about their purposes, their inputs, and outputs
  - #endif compiler directive

# TAXESRULES HEADER FILE

```
#ifndef _TAXES_RULES_
#define _TAXES_RULES_

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

double getIncome(string);
// purpose -- to get the
// employee income
// input -- a string prompt to
// be displayed to the user
// output -- a double value
// representing the income

double computeTaxes(double);
// purpose -- to compute the
// taxes for a given income
// input -- a double value
// representing the income
// output -- a double value
// representing the taxes

void printTaxes(double);
// purpose -- to display taxes to
// the user
// input -- a double value
// representing the taxes
// output -- None

#endif
```

# TAXESRULES IMPLEMENTATION FILE

```
#include "TaxesRules.h"
```

```
double computeTaxes(double  
    income)  
{  
    if (income<5000) return 0.0;  
    return 0.07*(income-5000.0);  
}
```

```
void printTaxes(double taxes)  
{  
    cout << "The taxes is $" <<  
    taxes << endl;  
}
```

```
double getIncome(string  
    prompt)  
{  
    cout << prompt;  
    double income;  
    cin >> income;  
    return income;  
}
```

# MAIN PROGRAM FILE

```
#include "TaxesRules.h"
void main()
{
    // Get the income;
    double income =
        getIncome("Please enter the employee
income: ");

    // Compute Taxes
    double taxes = computeTaxes(income);

    // Print employee taxes
    printTaxes(taxes);
}
```

# INLINE FUNCTIONS

◉ Sometimes, we use the keyword *inline* to define user-defined functions

- Inline functions are very small functions, generally, one or two lines of code
- Inline functions are very fast functions compared to the functions declared without the inline keyword

◉ Example

```
inline double degrees( double radian)
```

```
{  
    return radian * 180.0 / 3.1415;  
}
```

## EXAMPLE #1

- Write a function to test if a number is an odd number

```
inline bool odd (int x)  
{  
    return (x % 2 == 1);  
}
```

## EXAMPLE #2

- Write a function to compute the distance between two points  $(x_1, y_1)$  and  $(x_2, y_2)$

```
Inline double distance (double x1, double y1,  
                        double x2, double y2)
```

```
{  
    return sqrt(pow(x1-x2,2)+pow(y1-y2,2));  
}
```

## EXAMPLE #3

- Write a function to compute  $n!$

```
int factorial( int n)  
{  
    int product=1;  
    for (int i=1; i<=n; i++) product *= i;  
    return product;  
}
```



# EXAMPLE #4

## FUNCTION OVERLOADING

- Write functions to return with the maximum number of two numbers

```
inline int max( int x, int y)  
{  
    if (x>y) return x; else return y;  
}
```

```
inline double max( double x, double y)  
{  
    if (x>y) return x; else return y;  
}
```

An overloaded function is a function that is defined more than once with different data types or different number of parameters

# SHARING DATA AMONG USER-DEFINED FUNCTIONS

- There are two ways to share data among different functions
  - Using global variables (very bad practice!)
  - Passing data through function parameters
    - Value parameters
    - Reference parameters
    - Constant reference parameters

# C++ VARIABLES

- ◉ A variable is a place in memory that has
  - A name or identifier (e.g. income, taxes, etc.)
  - A data type (e.g. int, double, char, etc.)
  - A size (number of bytes)
  - A scope (the part of the program code that can use it)
    - Global variables - all functions can see it and using it
    - Local variables - only the function that declare local variables see and use these variables
  - A life time (the duration of its existence)
    - Global variables can live as long as the program is executed
    - Local variables are lived only when the functions that define these variables are executed

# I. USING GLOBAL VARIABLES

```
#include <iostream.h>
int x = 0;
void f1() { x++; }
void f2() { x+=4; f1(); }
void main()
{
    f2();
    cout << x << endl;
}
```

# I. USING GLOBAL VARIABLES

```
#include <iostream.h>
int x = 0;
void f1() { x++; }
void f2() { x+=4; f1(); }
void main()
{
    f2();
    cout << x << endl;
}
```

x

0


# I. USING GLOBAL VARIABLES

```
#include
<iostream.h>

int x = 0;
void f1() { x++; }
void f2() { x+=4;
    f1(); }
void main()
{
    f2();
    cout << x << endl;
}
```

x

0



```
void main()
{
    f2();
    cout << x << endl ;
}
```

# I. USING GLOBAL VARIABLES

```
#include
<iostream.h>
int x = 0;
void f1() { x++; }
void f2() { x+=4;
    f1(); }
void main()
{
    f2();
    cout << x << endl;
}
```

x

4

2

```
void f2()
{
    x += 4;
    f1();
}
```

1

```
void main()
{
    f2();
    cout << x << endl ;
}
```

# I. USING GLOBAL VARIABLES

```
#include
<iostream.h>
int x = 0;
void f1() { x++; }
void f2() { x+=4;
           f1(); }
void main()
{
    f2();
    cout << x << endl;
}
```

x 5

4

```
void f1()
{
    x++;
}
```

3

```
void f2()
{
    x += 4;
    f1();
}
```

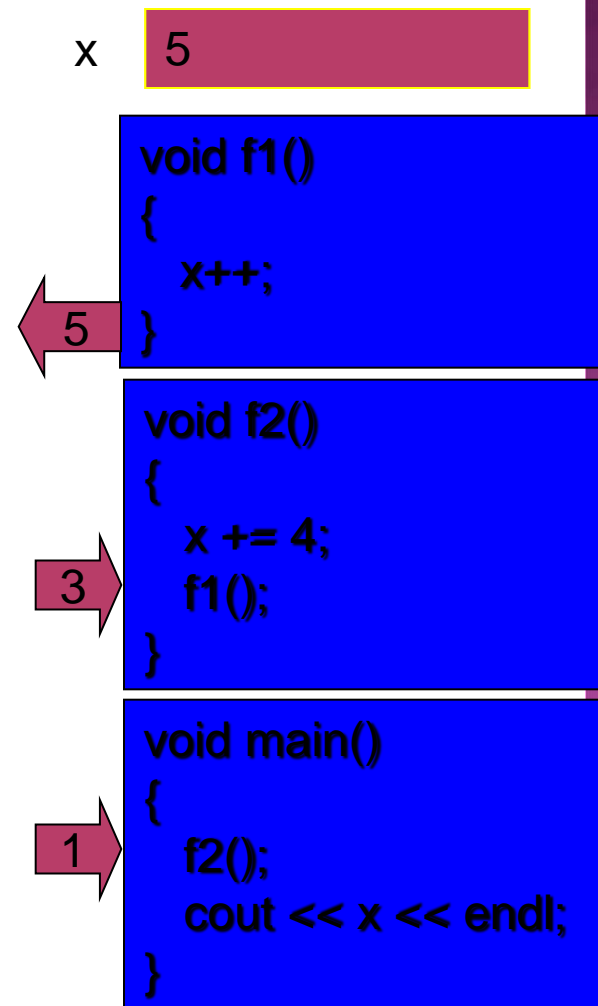
1

```
void main()
{
    f2();
    cout << x << endl ;
}
```



# I. USING GLOBAL VARIABLES

```
#include
<iostream.h>
int x = 0;
void f1() { x++; }
void f2() { x+=4;
           f1(); }
void main()
{
    f2();
    cout << x << endl;
}
```



# I. USING GLOBAL VARIABLES

```
#include
<iostream.h>
int x = 0;
void f1() { x++; }
void f2() { x+=4;
    f1(); }
void main()
{
    f2();
    cout << x << endl;
}
```

x

5

```
void f2()
{
    x += 4;
    f1();
}
```

6

```
void main()
{
    f2();
    cout << x << endl;
}
```

1

# I. USING GLOBAL VARIABLES

```
#include <iostream.h>
int x = 0;
void f1() { x++; }
void f2() { x+=4; f1(); }
void main()
{
    f2();
    cout << x << endl;
}
```

x 5



```
void main()
{
    f2();
    cout << x << endl;
}
```



# I. USING GLOBAL VARIABLES

```
#include <iostream.h>
int x = 0;
void f1() { x++; }
void f2() { x+=4; f1(); }
void main()
{
    f2();
    cout << x << endl;
}
```

x 5



```
void main()
{
    f2();
    cout << x << endl;
}
```

8

# I. USING GLOBAL VARIABLES

```
#include <iostream.h>
int x = 0;
void f1() { x++; }
void f2() { x+=4; f1(); }
void main()
{
    f2();
    cout << x << endl;
}
```



# WHAT HAPPENS WHEN WE USE INLINE KEYWORD?

```
#include <iostream.h>
int x = 0;
Inline void f1() { x++; }
Inline void f2() { x+=4; f1();}
void main()
{
    f2();
    cout << x << endl;
}
```

# WHAT HAPPENS WHEN WE USE INLINE KEYWORD?

```
#include <iostream.h>
```

```
int x = 0;
```

```
Inline void f1() { x++; }
```

```
Inline void f2() { x+=4; f1();}
```

```
void main()
```

```
{
```

```
    f2();
```

```
    cout << x << endl;
```

```
}
```

x 0

The inline keyword instructs the compiler to replace the function call with the function body!

1

```
void main()
{
    x+=4;
    x++;
    cout << x << endl;
}
```

# WHAT HAPPENS WHEN WE USE INLINE KEYWORD?

```
#include <iostream.h>
```

```
int x = 0;
```

```
Inline void f1() { x++; }
```

```
Inline void f2() { x+=4; f1();}
```

```
void main()
```

```
{
```


```
    f2();
```

```
    cout << x << endl;
```

```
}
```

x

4



```
void main()  
{  
    x+=4;  
    x++;  
    cout << x << endl;  
}
```



# WHAT HAPPENS WHEN WE USE INLINE KEYWORD?

```
#include <iostream.h>
```

```
int x = 0;
```

```
Inline void f1() { x++; }
```

```
Inline void f2() { x+=4; f1();}
```

```
void main()
```

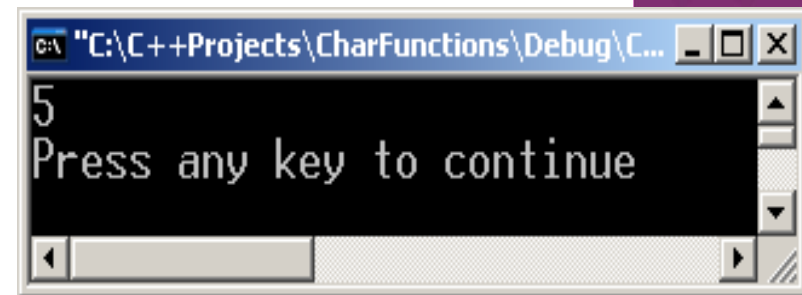
```
{
```

```
    f2();
```

```
    cout << x << endl;
```

```
}
```

x 5



3

```
void main()
{
    x+=4;
    x++;
    cout << x << endl;
}
```

# WHAT HAPPENS WHEN WE USE INLINE KEYWORD?

```
#include <iostream.h>
```

```
int x = 0;
```

```
Inline void f1() { x++; }
```

```
Inline void f2() { x+=4; f1();}
```

```
void main()
```

```
{
```

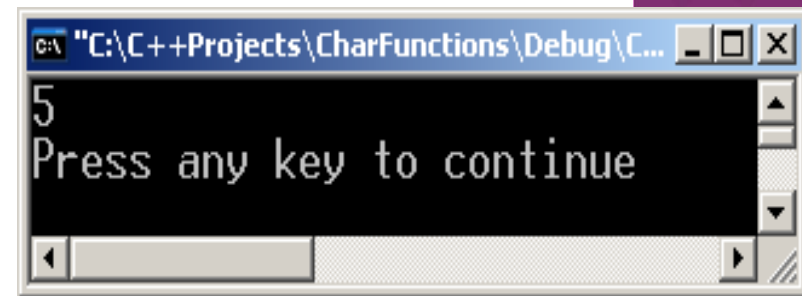
```
    f2();
```

```
    cout << x << endl;
```

```
}
```

x

5

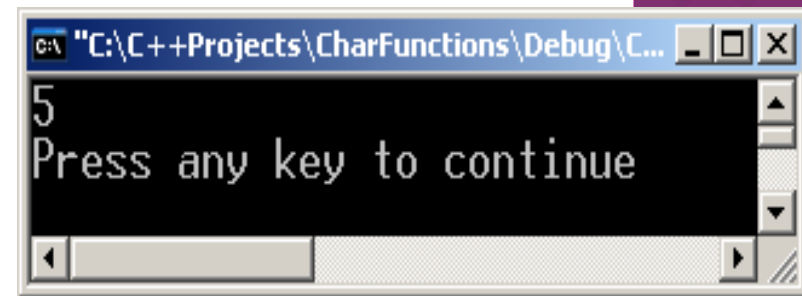


```
void main()  
{  
    x+=4;  
    x++;  
    cout << x << endl;  
}
```

4

# WHAT HAPPENS WHEN WE USE INLINE KEYWORD?

```
#include <iostream.h>
int x = 0;
Inline void f1() { x++; }
Inline void f2() { x+=4; f1();}
void main()
{
    f2();
    cout << x << endl;
}
```



A screenshot of a Windows command prompt window. The title bar shows the path "C:\C++Projects\CharFunctions\Debug\C...". The window has a black background with white text. It displays the number "5" on the first line and "Press any key to continue" on the second line. The window includes standard Windows controls (minimize, maximize, close) and a scrollbar on the right side.

# WHAT IS BAD ABOUT USING GLOBAL VAIRABLES?

## ⦿ Not safe!

- If two or more programmers are working together in a program, one of them may change the value stored in the global variable without telling the others who may depend in their calculation on the old stored value!

## ⦿ Against The Principle of Information Hiding!

- Exposing the global variables to all functions is against the principle of information hiding since this gives all functions the freedom to change the values stored in the global variables at any time (unsafe!)

# LOCAL VARIABLES

- ◉ Local variables are declared inside the function body and exist as long as the function is running and destroyed when the function exit
- ◉ You have to initialize the local variable before using it
- ◉ If a function defines a local variable and there was a global variable with the same name, the function uses its local variable instead of using the global variable

# EXAMPLE OF DEFINING AND USING GLOBAL AND LOCAL VARIABLES

```
#include <iostream.h>
int x; // Global variable
Void fun(); // function signature
```

```
void main()
{
    x = 4;
    fun();
    cout << x << endl;
}
```

```
void fun()
{
    int x = 10; // Local variable
    cout << x << endl;
}
```

# EXAMPLE OF DEFINING AND USING GLOBAL AND LOCAL VARIABLES

```
#include <iostream.h>
int x; // Global variable
Void fun(); // function signature
```

```
void main()
{
    x = 4;
    fun();
    cout << x << endl;
}
```

```
void fun()
{
    int x = 10; // Local variable
    cout << x << endl;
}
```

x

0

**Global variables are automatically initialized to 0**

# EXAMPLE OF DEFINING AND USING GLOBAL AND LOCAL VARIABLES

```
#include <iostream.h>
int x; // Global variable
Void fun(); // function signature
```

```
void main()
{
    x = 4;
    fun();
    cout << x << endl;
}
```

```
void fun()
{
    int x = 10; // Local variable
    cout << x << endl;
}
```

x

0

1

```
void main()
{
    x = 4;
    fun();
    cout << x << endl;
}
```



# EXAMPLE OF DEFINING AND USING GLOBAL AND LOCAL VARIABLES

```
#include <iostream.h>
int x; // Global variable
Void fun(); // function signature
```

```
void main()
{
    x = 4;
    fun();
    cout << x << endl;
}
```

```
void fun()
{
    int x = 10; // Local variable
    cout << x << endl;
}
```

x

4

void fun()

x

????

3

```
{
    int x = 10;
    cout << x << endl;
}
```

2

void main()

```
{
    x = 4;
    fun();
    cout << x << endl;
}
```

# EXAMPLE OF DEFINING AND USING GLOBAL AND LOCAL VARIABLES

```
#include <iostream.h>
int x; // Global variable
Void fun(); // function signature
```

```
void main()
{
    x = 4;
    fun();
    cout << x << endl;
}
```

```
void fun()
{
    int x = 10; // Local variable
    cout << x << endl;
}
```

x

4

void fun()

x 10

3

```
{
    int x = 10;
    cout << x << endl;
}
```

void main()

2

```
{
    x = 4;
    fun();
    cout << x << endl;
}
```

# EXAMPLE OF DEFINING AND USING GLOBAL AND LOCAL VARIABLES

```
#include <iostream.h>
int x; // Global variable
Void fun(); // function signature
```

```
void main()
{
    x = 4;
    fun();
    cout << x << endl;
}
```

```
void fun()
{
    int x = 10; // Local variable
    cout << x << endl;
}
```

x

4



4

**void fun()**

x

10

```
{
    int x = 10;
    cout << x << endl;
}
```

**void main()**

```
{
    x = 4;
    fun();
    cout << x << endl;
}
```

2

# EXAMPLE OF DEFINING AND USING GLOBAL AND LOCAL VARIABLES

```
#include <iostream.h>
int x; // Global variable
Void fun(); // function signature
```

x

4

```
void main()
{
    x = 4;
    fun();
    cout << x << endl;
}
```

```
void fun()
{
    int x = 10; // Local variable
    cout << x << endl;
}
```



**void fun()**

x

10

```
{
    int x = 10;
    cout << x << endl;
}
```

5

**void main()**

```
{
    x = 4;
    fun();
    cout << x << endl;
}
```

2

# EXAMPLE OF DEFINING AND USING GLOBAL AND LOCAL VARIABLES

```
#include <iostream.h>
int x; // Global variable
Void fun(); // function signature
```

x

4

```
void main()
{
    x = 4;
    fun();
    cout << x << endl;
}
```



```
void fun()
{
    int x = 10; // Local variable
    cout << x << endl;
}
```

6

```
void main()
{
    x = 4;
    fun();
    cout << x << endl;
}
```

# EXAMPLE OF DEFINING AND USING GLOBAL AND LOCAL VARIABLES

```
#include <iostream.h>
int x; // Global variable
Void fun(); // function signature
```

x

4

```
void main()
{
    x = 4;
    fun();
    cout << x << endl;
}
```



```
void fun()
{
    int x = 10; // Local variable
    cout << x << endl;
}
```

```
void main()
{
    x = 4;
    fun();
    cout << x << endl;
}
```

7

## II. USING PARAMETERS

- ⦿ Function Parameters come in three flavors:
  - *Value parameters* - which copy the values of the function arguments
  - *Reference parameters* - which refer to the function arguments by other local names and have the ability to change the values of the referenced arguments
  - *Constant reference parameters* - similar to the reference parameters but cannot change the values of the referenced arguments

# VALUE PARAMETERS

- This is what we use to declare in the function signature or function header, e.g.

```
int max (int x, int y);
```

- Here, parameters x and y are value parameters
- When you call the max function as *max(4, 7)*, the values 4 and 7 are copied to x and y respectively
- When you call the max function as *max (a, b)*, where a=40 and b=10, the values 40 and 10 are copied to x and y respectively
- When you call the max function as *max( a+b, b/2)*, the values 50 and 5 are copied to x and y respectively
- Once the value parameters accepted copies of the corresponding arguments data, they act as local variables!



# EXAMPLE OF USING VALUE PARAMETERS AND GLOBAL VARIABLES

```
#include <iostream.h>
int x; // Global variable
void fun(int x)
{
    cout << x << endl;
    x=x+5;
}
void main()
{
    x = 4;
    fun(x/2+1);
    cout << x << endl;
}
```

x

0

1 →

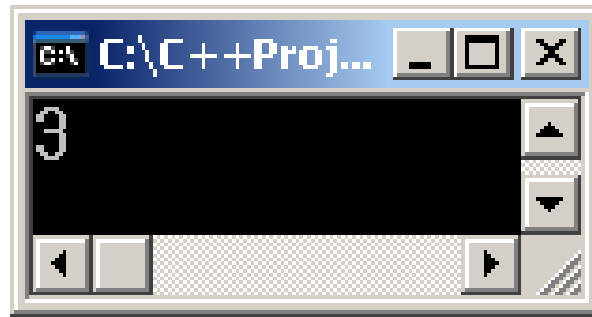
```
void main()
{
    x = 4;
    fun(x/2+1);
    cout << x << endl;
}
```

# EXAMPLE OF USING VALUE PARAMETERS AND GLOBAL VARIABLES

```
#include <iostream.h>
int x; // Global variable
void fun(int x)
{
    cout << x << endl;
    x=x+5;
}
void main()
{
    x = 4;
    fun(x/2+1);
    cout << x << endl;
}
```

x

4



3

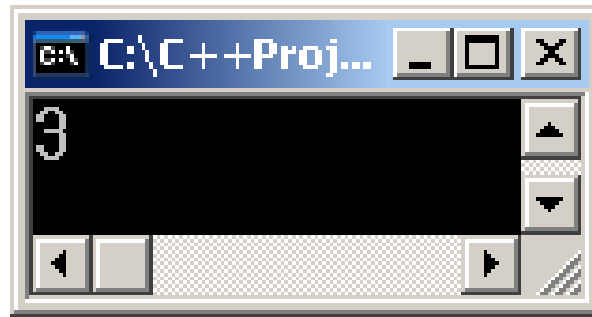
```
void fun(int x )
{
    cout << x << endl;
    x=x+5;
}
```

2

```
void main()
{
    x = 4;
    fun(x/2+1);
    cout << x << endl;
}
```

# EXAMPLE OF USING VALUE PARAMETERS AND GLOBAL VARIABLES

```
#include <iostream.h>
int x; // Global variable
void fun(int x)
{
    cout << x << endl;
    x=x+5;
}
void main()
{
    x = 4;
    fun(x/2+1);
    cout << x << endl;
}
```



x

4

4 →

```
void fun(int x 8 )
{
    cout << x << endl;
    x=x+5;
}
```

2 →

```
void main()
{
    x = 4;
    fun(x/2+1);
    cout << x << endl;
}
```

# EXAMPLE OF USING VALUE PARAMETERS AND GLOBAL VARIABLES

```
#include <iostream.h>
int x; // Global variable
void fun(int x)
{
    cout << x << endl;
    x=x+5;
}
void main()
{
    x = 4;
    fun(x/2+1);
    cout << x << endl;
}
```

x

4



```
void fun(int x 8 )
{
    cout << x << endl;
    x=x+5;
}
```

```
void main()
{
    x = 4;
    fun(x/2+1);
    cout << x << endl;
}
```

# EXAMPLE OF USING VALUE PARAMETERS AND GLOBAL VARIABLES

```
#include <iostream.h>
int x; // Global variable
void fun(int x)
{
    cout << x << endl;
    x=x+5;
}
void main()
{
    x = 4;
    fun(x/2+1);
    cout << x << endl;
}
```

x

4



6

```
void main()
{
    x = 4;
    fun(x/2+1);
    cout << x << endl;
}
```

# EXAMPLE OF USING VALUE PARAMETERS AND GLOBAL VARIABLES

```
#include <iostream.h>
int x; // Global variable
void fun(int x)
{
    cout << x << endl;
    x=x+5;
}
void main()
{
    x = 4;
    fun(x/2+1);
    cout << x << endl;
}
```

x

4



```
void main()
{
    x = 4;
    fun(x/2+1);
    cout << x << endl;
}
```

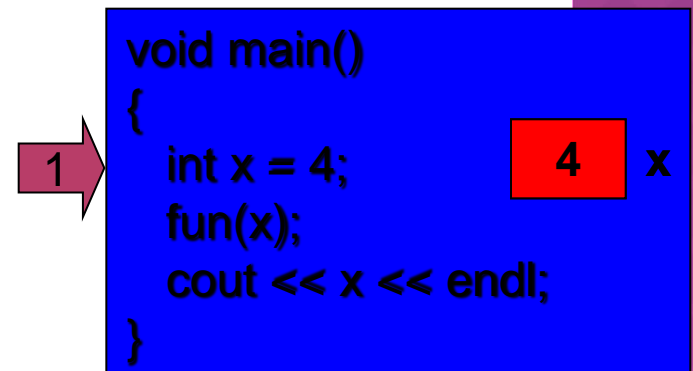
7

# REFERENCE PARAMETERS

- ⦿ As we saw in the last example, any changes in the value parameters don't affect the original function arguments
- ⦿ Sometimes, we want to change the values of the original function arguments or return with more than one value from the function, in this case we use reference parameters
  - A reference parameter is just another name to the original argument variable
  - We define a reference parameter by adding the & in front of the parameter name, e.g.  
double update (double & x);

# EXAMPLE OF REFERENCE PARAMETERS

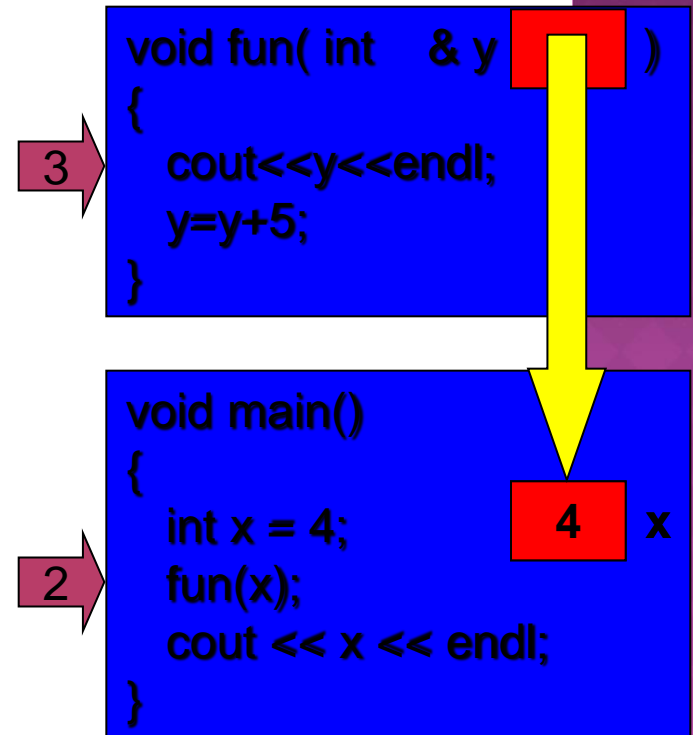
```
#include <iostream.h>
void fun(int &y)
{
    cout << y << endl;
    y=y+5;
}
void main()
{
    int x = 4; // Local
    variable
    fun(x);
    cout << x << endl;
}
```





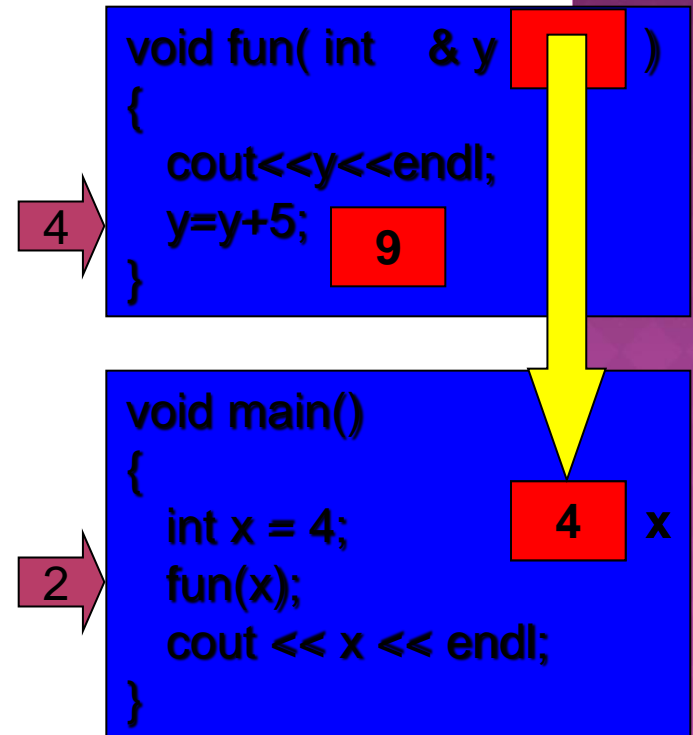
# EXAMPLE OF REFERENCE PARAMETERS

```
#include <iostream.h>
void fun(int &y)
{
    cout << y << endl;
    y=y+5;
}
void main()
{
    int x = 4; // Local
    variable
    fun(x);
    cout << x << endl;
}
```



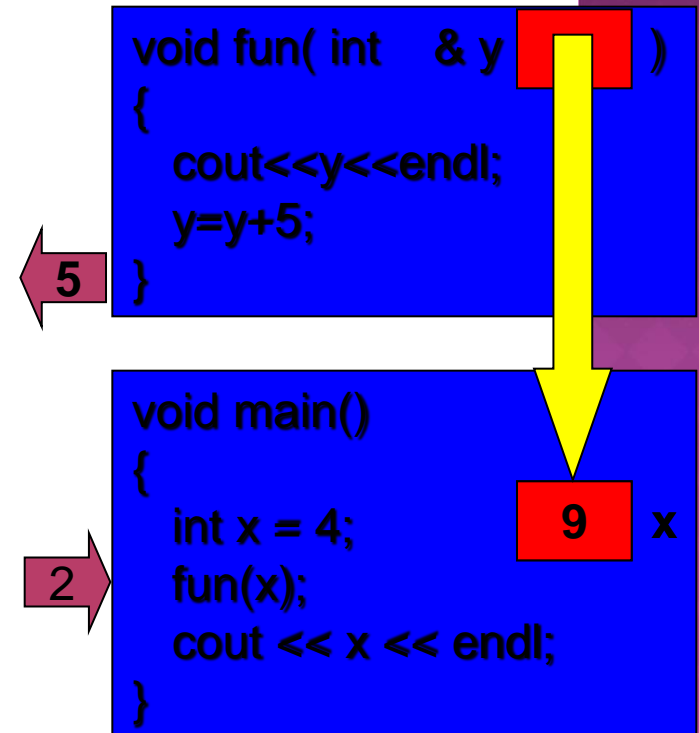
# EXAMPLE OF REFERENCE PARAMETERS

```
#include <iostream.h>
void fun(int &y)
{
    cout << y << endl;
    y=y+5;
}
void main()
{
    int x = 4; // Local
    variable
    fun(x);
    cout << x << endl;
}
```



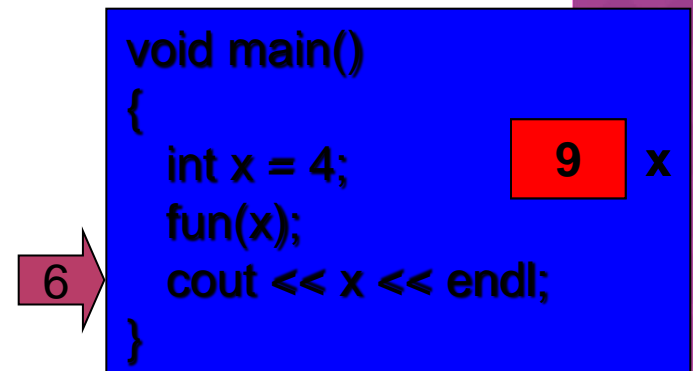
# EXAMPLE OF REFERENCE PARAMETERS

```
#include <iostream.h>
void fun(int &y)
{
    cout << y << endl;
    y=y+5;
}
void main()
{
    int x = 4; // Local
    variable
    fun(x);
    cout << x << endl;
}
```



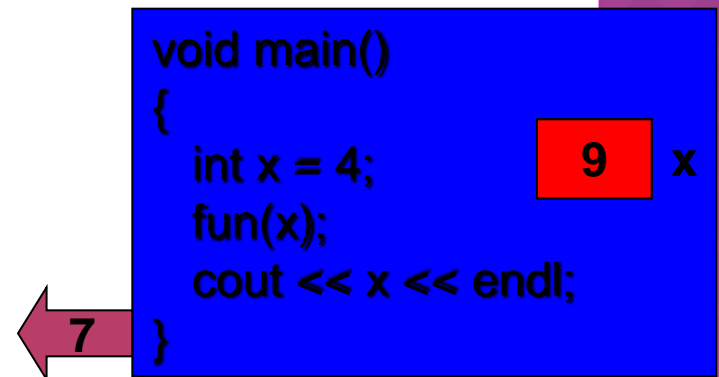
# EXAMPLE OF REFERENCE PARAMETERS

```
#include <iostream.h>
void fun(int &y)
{
    cout << y << endl;
    y=y+5;
}
void main()
{
    int x = 4; // Local
    variable
    fun(x);
    cout << x << endl;
}
```



# EXAMPLE OF REFERENCE PARAMETERS

```
#include <iostream.h>
void fun(int &y)
{
    cout << y << endl;
    y=y+5;
}
void main()
{
    int x = 4; // Local
    variable
    fun(x);
    cout << x << endl;
}
```



# CONSTANT REFERENCE PARAMETERS

- ◉ Constant reference parameters are used under the following two conditions:
  - The passed data are so big and you want to save time and computer memory
  - The passed data will not be changed or updated in the function body
- ◉ For example  
    void report (const string & prompt);
- ◉ The only valid arguments accepted by reference parameters and constant reference parameters are variable names
  - It is a syntax error to pass constant values or expressions to the (const) reference parameters