fit@hcmus

Functions

1

fit@hcmus

Contents

- Predefined functions
- User-defined functions
- Scope rules
- Parameters
- Function overloading
- Default arguments

3

fit@hcmus

Functions

- Functions are like building blocks
- They allow complicated programs to be divided into manageable pieces
- Some advantages of functions:
  - 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

4

fit@hcmus

Functions

- Other names:
  - Procedure
  - Subprogram
  - Method
- Types:
  - Pre-defined functions
  - User-defined (Programmer-defined) functions

5

fit@hcmus

Predefined Functions

Do not reinvent the wheel.

6

fit@hcmus

Predefined Functions

`<data_type returned/void>` **`Function_Name`** `(Argument_List)`

- `void` function: Function does not produce a value.
- Argument list: comma-separated list of arguments.

7

fit@hcmus

Some 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
- Some functions are in `cstdlib` header.

8

fit@hcmus

Some Predefined Functions

- `pow(x,y)` calculates x^y
  - `pow(2, 3) = 8.0`
  - Returns a value of type `double`
  - `x` and `y` are the **parameters** (or **arguments**). The function has two parameters.
  - `8.0` is **value returned**.

- `sqrt(x)` calculates the nonnegative square root of `x`, for `x>=0.0`
  - `sqrt(2.25)` is `1.5`
  - Returns the value of type `double`

9

fit@hcmus

Some Predefined Functions

- The `floor(x)` function calculates largest whole number not greater than `x`
  - `floor(48.79)` is `48.0`
  - Type `double`
  - Has only one parameter

- The `abs(x)`, `labs(x)`, `fabs(x)` functions calculate the absolute value of `x` (`x` is integer, long or floating-point number).

fit@hcmus

NAME	DESCRIPTION	TYPE OF ARGUMENTS	TYPE OF VALUE RETURNED	EXAMPLE	VALUE	LIBRARY HEADER
sqrt	Square root	double	double	sqrt(4.0)	2.0	cmath
pow	Powers	double	double	pow(2.0,3.0)	8.0	cmath
abs	Absolute value for int	int	int	abs(-7) abs(7)	7 7	cstdlib
labs	Absolute value for long	long	long	labs(-70000) labs(70000)	70000 70000	cstdlib
fabs	Absolute value for double	double	double	fabs(-7.5) fabs(7.5)	7.5 7.5	cmath
ceil	Ceiling (round up)	double	double	ceil(3.2) ceil(3.9)	4.0 4.0	cmath
floor	Floor (round down)	double	double	floor(3.2) floor(3.9)	3.0 3.0	cmath
exit	End program	int	void	exit(1);	None	cstdlib
rand	Random number	None	int	rand( )	Varies	cstdlib
srand	Set seed for rand	unsigned int	void	srand(42);	None	cstdlib

Some Predefined Functions

- `cos(x)`, `cmath`: return cosine of angle `x`.
- `tolower(c)`, `cctype`: return lowercase of `c`.
- `toupper(c)`, `cctype`: return UPPERCASE of `c`.

12

`void` Predefined Functions

- `exit(integer)`
  - Library `cstdlib`
  - Program ends immediately.
  - argument value:
    - 1: caused by error
    - 0: other cases.

```
1  #include <iostream>
2  #include <cstdlib>
3  using namespace std;

4  int main( )
5  {
6      cout << "Hello Out There!\n";
7      exit(1);

8      cout << "This statement is pointless,\n"
9           << "because it will never be executed.\n"
10          << "This is just a toy program to illustrate exit.\n";

11     return 0;
12 }
```

This is just a toy example. It would produce the same output if you omitted these lines.

13

fit@hcmus

User-defined Functions

14

fit@hcmus

Example

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

3  double totalCost(int numberParameter, double priceParameter);
4  //Computes the total cost, including 5% sales tax,
5  //on numberParameter items at a cost of priceParameter each.

6  int main( )
7  {
8      double price, bill;
9      int number;

10     cout << "Enter the number of items purchased: ";
11     cin >> number;
12     cout << "Enter the price per item $";
13     cin >> price;

14     bill = totalCost(number, price);

15     cout.setf(ios::fixed);
16     cout.setf(ios::showpoint);
17     cout.precision(2);
18     cout << number << " items at "
19          << "$" << price << " each.\n"
20          << "Final bill, including tax, is $" << bill
21          << endl;

22     return 0;
23 }
```

Function declaration; also called the function prototype

Function call

Function head

15

Example

```
10      cout << "Enter the number of items purchased: ";
11      cin >> number;
12      cout << "Enter the price per item $";
13      cin >> price;
14      bill = totalCost(number, price);

15      cout.setf(ios::fixed);
16      cout.setf(ios::showpoint);
17      cout.precision(2);
18      cout << number << " items at "
19           << "$" << price << " each.\n"
20           << "Final bill, including tax, is $" << bill
21           << endl;

22      return 0;
23  }

24  double totalCost(int numberParameter, double priceParameter)
25  {
26      const double TAXRATE = 0.05; //5% sales tax
27      double subtotal;

28      subtotal = priceParameter * numberParameter;
29      return (subtotal + subtotal*TAXRATE);
30  }
```

Function call

Function head

Function body

Function definition

16

Terminologies

- **Function declaration**/Function prototype:
  - determines the kind of function
  - tells the name of the functions
  - tells number and types of arguments
  - ends with a semicolon.
- **Function definition:** describes how the function works.
  - **Function header:** same as function declaration (without semicolon at the end).
  - **Function body:** consists of declaration and executable statements enclosed within a pair of braces.

17

fit@hcmus

Terminologies

- Function call/Function invocation
- **Formal argument** (argument): variable declared in function header, or function prototype.
- Actual argument: variable or expression listed in a function call.

18

fit@hcmus

Function Declaration

- Syntax:

```
Type_Returned/void FunctionName(Parameter_List);
```

  - `Parameter_List` can be empty (function with no arguments).
- Normally placed before the `main` function.

19

return Statement

fit@hcmus

- Once a value-returning function computes the value, the function returns this value via the `return` statement
  - It passes this value outside the function via the `return` statement
- Syntax:

```
return expression;
```


20

return Statement

fit@hcmus

- When a `return` statement executes
  - Function immediately terminates
  - Control goes back to the caller
- A `return` statement in `void` function simply ends the function call.
  - `void` function needs not contain
- When a `return` statement executes in the function `main`, the program terminates.

21

fit@hcmus

Examples

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

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

    return max;
}
```

You can also write this function as follows:

```
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;
}
```


fit@hcmus

Examples

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

double compareThree (double x, double y, double z)
{
    return larger(x, larger(y, z));
}
```

Function `main`

- The `main` part of a program is the definition of a function called `main`.
- When program runs, the function `main` is automatically called.
- Some compiler requires `return 0;` in the `main` function.

24

Recursive Functions

- The function calls itself.
- Example:

```
int Factorial(int N)
{
   if (N == 0)
         return 1;
   return N * Factorial(N-1);
}
```


25

fit@hcmus

Scope Rules

26

fit@hcmus

Scope of an Identifier

- The scope of an identifier refers to where in the program an identifier is accessible.
  - **Local identifier**: identifiers declared within a function (or block)
  - **Global identifier**: identifiers declared outside of every function definition
- C++ does not allow nested functions
  - The definition of one function cannot be included in the body of another function

27

Local Variables

- Variables are declared within the body of a function.
  - **Local** to that function.
- Two same name (local) variables in two different functions are different.

28

Local Variables

- Variables are declared within the body of a function.
  - **Local** to that function.
- Two same name (local) variables in two different functions are different.

29

fit@hcmus

Global Constants

- Use the `const` modifier to name constant value.

```
const double TAX_RATE = 0.1; //VAT tax: 10%
```

- If the declaration is outside of all functions, the named constant is **global named constant**.

30

fit@hcmus

Global Variables

- **Global variable**:
  - same as global named constant, without using `const` modifier.
  - accessible to all function definitions in a file.

31

Global Variables

- Some compilers initialize global variables to default values
- The operator `::` is called the scope resolution operator
- By using 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.

32

Global Variables | Example

```
1  #include <iostream>
2
3  int a;
4
5  void PrintA()
6  {
7        int a = 7;
8        std::cout << "Local variable: " << a << std::endl;
9        std::cout << "Global variable: " << ::a << std::endl;
10 }
11
12 int main()
13 {
14       a = 15;
15       PrintA();
16       return 0;
17 }
```


33

fit@hcmus

Side Effects

- Using global variables has 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
- Global named constants have no side effects.

34

fit@hcmus

Blocks

- A block is some C++ code enclosed in braces.
- Variables declared in a block are local to that block.

35

fit@hcmus

Blocks

```
#include <iostream>

int main()
{
	int a, b;
	a = 16;
	b = 8;
	std::cout << "Outside\t a - b: " << a << " - " << b << std::endl;
	if (b > 0)
	{
		int a;
		a = 90;
		std::cout << "Inside\t b - a: " << b << " - " << a << std::endl;
	}

	std::cout << "Outside\t b - a: " << b << " - " << a << std::endl;
	return 0;
}
```


36

fit@hcmus

Nested Blocks

- If one identifier is declared in two blocks (nested), they are different variables with the same name.

37

Nested Blocks

fit@hcmus

```
#include <iostream>

int main()
{
	int a, b;
	a = 16;
	b = 8;
	std::cout << "Outside\t a - b: " << a << " - " << b << std::endl;
	if (b > 0)
	{
		int a;
		a = 90;
		if (a % 2 == 0)
		{
			int a = 24;
			std::cout << "Nested\t b - a: " << b << " - " << a << std::endl;
		}
		else
			std::cout << a << " is odd\n";
		std::cout << "Inside\t b - a: " << b << " - " << a << std::endl;
	}

	std::cout << "Outside\t b - a: " << b << " - " << a << std::endl;
	return 0;
}
```


38

Static Local Variables

fit@hcmus

- Keyword `static` is used for specifying a static variable.
- Examples:

```
static int a;
static float b;
```


39

fit@hcmus

Static Local Variables

- A **static local variable** exists **only** inside a function where it is declared (like a **local variable**) but its lifetime **starts** when the function is called and **ends** only when the program ends.
- The main difference between local variable and static variable is that the value of static variable persists the end of the program.

40

fit@hcmus

Static Local Variables | Example

```
1  #include <iostream>
2
3  void Test()
4  {
5        static int count = 0;
6        count++;
7        std::cout << "Call of " << count << std::endl;
8  }
9  int main()
10 {
11       Test();
12       Test();
13       return 0;
14 }
```


41

Example | Classifying Numbers

- We use **functions** to write the program that determines the number of odds and evens from a given list of integers.
- Main algorithm remains the same:
  - Initialize variables, zeros, odds, evens to 0
  - Read a number.
  - If number is even, increment the even count
    - If number is also zero, increment the zero count; else increment the odd count
  - Repeat Steps 2-3 for each number in the list.

Example | Classifying Numbers

- The program functions include:
  - **`initialize:`** initialize the variables, such as zeros, odds, and evens
  - **`getNumber`**: get the number
  - **`classifyNumber`**: determine if number is odd or even (and whether it is also zero); this function also increments the appropriate count
  - **`printResults`**: print the results

fit@hcmus

Exercises

- Assignment 12 (Page 143).

fit@hcmus

Parameters

fit@hcmus

Value vs Reference Parameters

- **Call-by-Value parameter:** a formal parameter that receives **a copy of the content** of corresponding actual parameter.
  - Can be variables or expressions.
- **Call-by-Reference parameter:** a formal parameter that receives **the location (memory address)** of the corresponding actual parameter.
  - Only be variables.

46

fit@hcmus

Call-by-Value Parameters

- Is actually a local variable.
- The value of the corresponding actual parameter is copied into it.
- The parameter has its own copy of the data.
- During program execution
  - The parameter manipulates the data stored in its own memory space.

47

fit@hcmus

Call-by-Reference Parameters

- It receives the memory address of the corresponding actual parameter.
- The parameter stores the address of the corresponding actual parameter.
- During program execution to manipulate data
  - The address stored in the parameter directs it to the memory space of the corresponding actual parameter.

48

fit@hcmus

Call-by-Reference Parameters

- Indicating the call-by-reference parameters by attaching the ampersand sign **&** at the of the type name in formal parameter list.
- Example:

```
void getInput(double& N);
void sum(int N, int& s);
```


49

fit@hcmus

Call-by-Reference Parameters

- **Call-by-Reference** parameters can:
  - Pass one or more values from a function
  - Change the value of the actual parameter
- Call-by-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

50

fit@hcmus

Example

- Write a function to swap the value of two integer variables a, b.

53

fit@hcmus

Example

- Write a function to swap the value of two integer variables a, b.

- Version 01:

```
void swap(int a, int b)
{
   int temp;
   temp = a;
   a = b;
   b = temp;
}
```


fit@hcmus

Example

- Write a function to swap the value of two integer variables a, b.

- Version 02:

```
void swap(int& a, int& b)
{
   int temp;
   temp = a;
   a = b;
   b = temp;
}
```

fit@hcmus

Function Overloading

57

fit@hcmus

Overloaded Functions

- Two or more functions having **same name** but **different argument(s)** are known as overloaded functions.
- The signature of a function consists of the **function name** and its **formal parameter list**.
- Examples:

```
int test();
void test(int);
void test(float);
int test(float);
float test();
```


58

fit@hcmus

Overloaded Functions | Examples

```
1  void display(int var) {
2      std::cout << "Integer number: " << var << std::endl;
3  }
4
5  void display(float var) {
6      std::cout << "Float number: " << var << std::endl;
7  }
8
9  void display(int var1, float var2) {
10     std::cout << "Integer number: " << var1 << std::endl;
11     std::cout << " and float number:" << var2 << std::endl;
12 }
```


59

fit@hcmus

Overloaded Functions | Examples

```
1  int absolute(int var) {
2       if (var < 0)
3           var = -var;
4      return var;
5  }
6
7  float absolute(float var){
8      if (var < 0.0)
9          var = -var;
10     return var;
11 }
```


60

fit@hcmus

Default Arguments

61

fit@hcmus

Default Arguments

- In a function call, the **number of** actual parameters and formal parameters must **be the same**.
  - C++ relaxes this condition for functions with **default parameters**
- You specify the value of a default parameter when the function name appears for the first time (e.g., in the prototype).
- If you do not specify the value of a default parameter, the default value is used.

62

fit@hcmus

Default Arguments

- All default parameters must be the **rightmost** parameters of the function.
- Default values can be constants, global variables, or function calls
  - Cannot assign a **constant value** as a default value **to a reference parameter**.
- In a function call where the function has more than one default parameter and a value to a default parameter is not specified:
  - You must omit all of the arguments to its right

63

fit@hcmus

Examples

```
#include <iostream>

// A function with default arguments, it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z = 0, int w = 0)
{
	return (x + y + z + w);
}

int main()
{
	std::cout << sum(10, 15) << std::endl;
	std::cout << sum(10, 15, 25) << std::endl;
	std::cout << sum(10, 15, 25, 30) << std::endl;
	return 0;
}
```


64

fit@hcmus

Examples

- Illegal function prototypes:

```
void funcOne(int x, double z = 23.45, char ch, int u = 45);
int funcTwo(int length = 1, int width, int height = 1);
void funcThree(int x, int& y = 16, double z = 34);
```


65

fit@hcmus

Examples

Consider the following function prototype:

```
void testDefaultParam(int a, int b = 7, char z = '*');
```

Which of the following function calls is correct?

a. `testDefaultParam(5);`
b. `testDefaultParam(5, 8);`
c. `testDefaultParam(6, '#');`
d. `testDefaultParam(0, 0, '*');`

66

fit@hcmus

Function as an Argument

```
#include <iostream>

int add(int a, int b)
{
	return a + b;
}

int subtract(int a, int b)
{
	return a - b;
}

void Print(int Func(int, int), int a, int b, char c)
{
	std::cout << a << c << b << " is " << Func(a, b) << std::endl;
}

int main()
{
	int a = 16, b = 8;
	Print(add, a, b, '+');
	Print(subtract, a, b, '-');
	return 0;
}
```


67

fit@hcmus

Questions and Answers

68