

# CS2311 Computer Programming

## Functions

pass-by-value and pass-by-reference

# What is function?

- A collection of statements that perform a specific task.
- Functions are used to break a problem down into manageable pieces
- Write several small functions to solve a specific part of the problem
- A function can be invoked multiple times. No need to repeat the same code in multiple parts of the program.

# Function in C++

- The C++ standard library provides a rich collection of functions for
- Mathematical calculations (`#include <cmath>`)
- String manipulations (`#include <cstring>`)
- Input/output (`#include <iostream>`)

# How to use function written by others?

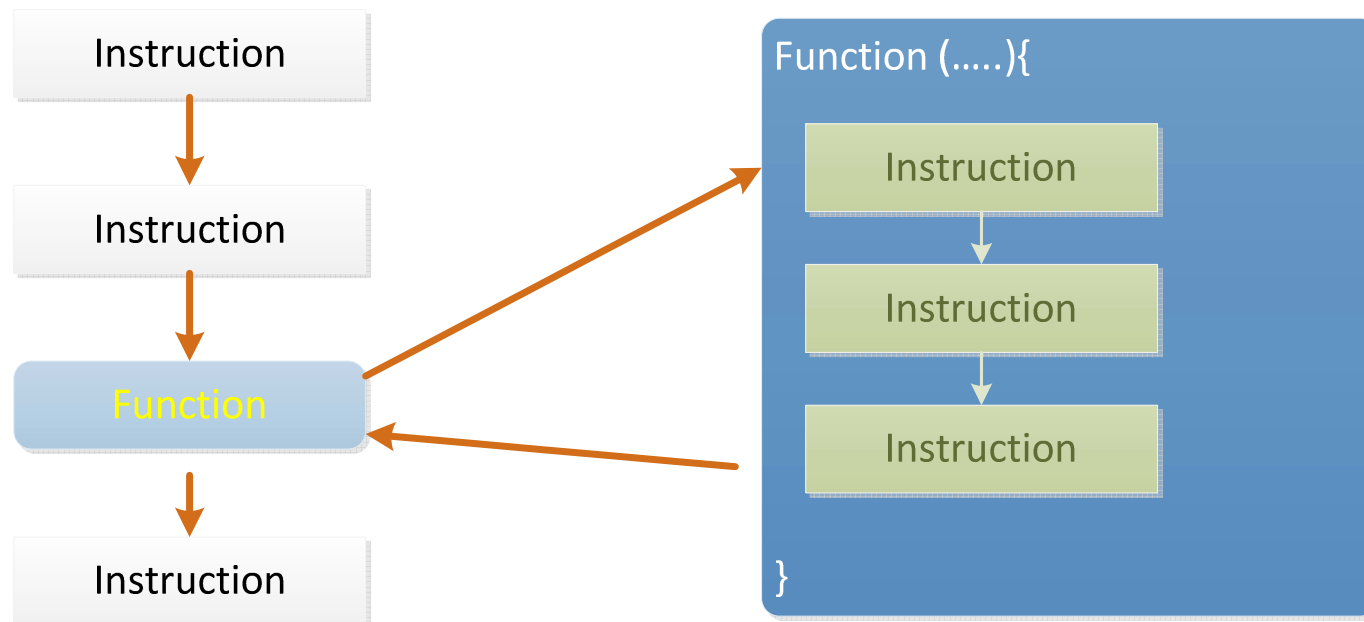
```
#include <iostream>
#include <cmath>
using namespace std;
void main()
{
    double area, side;
    cout << "Enter the area of a square:";
    cin >> area;
    side=sqrt(area);
    cout << "The square has perimeter: " << 4*side;
}
```

Tell compiler that you are going to use functions defined in **cmath** package

Pass area to the function **sqrt** which will return the square root of area

# Function invocation

- During program execution, when a function name followed by parentheses is encountered, the function is invoked and the program control is passed to that function; when the function ends, program control is returned to the statement *immediately after* the function call in the original function



## Write your own function (User defined functions)

- Define a function `printHello` which accepts an integer `n` as input
- The function should print "Hello" `n` times, where `n` is an integer

# Function Components

Return type

Function name

Input (Parameter)

```
void printHello (int n){  
    int i;  
    for (i=0; i<n; i++)  
        cout <<"Hello\n";  
}
```

Function body

n is defined as input, therefore there is **no need to declare n again** in the function body again

# Calling (*executing*) a function (I)

To make a function call, we only need to specify a function name and provide the parameters inside a pair of parenthesis ( )

| Function name               | Input |
|-----------------------------|-------|
| ↓                           | ↓     |
| <code>printHello(3);</code> |       |

Print "hello" 3 times

**IMPORTANT:** We don't write the return type (e.g. `int`) when calling a function.

**Syntax error:**

`int printHello(3);`

## Calling a function (II)

```
int x=4;  
printHello (x);  
printHello (x+2);
```

Print "hello" 4 times and then 6 times.

We don't need the parameter type when calling a function.

**Syntax error:**

```
printHello(int x );
```

Advantage of using a function: we don't need to write two loops, one to print "Hello" 4 times and the other to print "Hello" 6 times

# Flow of control

```
void main(){  
    int x=4;  
    printHello(x);  
    cout << "bye";  
}
```

```
void printHello (int n){  
    int i;  
    for (i=0; i<n; i++)  
        cout << "Hello\n";  
}
```

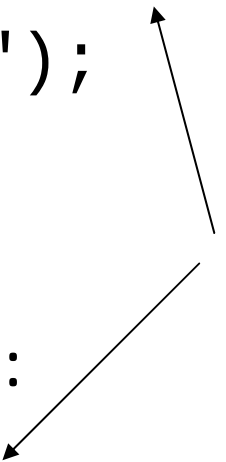
1. The program first start execution in main()
2. printHello(x) is called
3. The value of x is **copied** to the variable n. As a result, n gets a value of 4.
4. The loop body is executed.
5. After executing all the statements within printHello(), control go back to main() and "bye" is printed

# Function with no input

```
void printHello (){  
    print("hello");  
}
```

To call the function:  
`printHello();`

Note: The parenthesis ( ) is NOT optional. It cannot be omitted!



- It is also okay to write `void printHello(void)`
- Note that the first `void` (return type) is **compulsory**
- The second `void` (parameter) is **optional**

# Examples: Function w/o return value

Usually the output is displayed directly to screen

| Function    | Parameter            | return value | Examples:                                                                                                                  |
|-------------|----------------------|--------------|----------------------------------------------------------------------------------------------------------------------------|
| printHello  | nil                  | nil          | <pre>void printHello(){<br/>    cout &lt;&lt; "Hello\n";<br/>}</pre>                                                       |
| printHellos | int n                | nil          | <pre>void printHellos(int n){<br/>    int i;<br/>    for (i=0;i&lt;n;i++)<br/>        cout &lt;&lt; "Hello\n";<br/>}</pre> |
| printMax    | float n1<br>float n2 | nil          | <pre>void printMax(float n1,float n2){<br/>    cout &lt;&lt; (n1&gt;n2)?n1:n2 &lt;&lt; endl;<br/>}</pre>                   |

# Examples: Function with return value

The **return value** can be user input, result of processing / calculation ...etc

| Function | Parameter            | return value | Examples:                                                                                                                           |
|----------|----------------------|--------------|-------------------------------------------------------------------------------------------------------------------------------------|
| getX     | nil                  | int          | <pre>int getX (){<br/>    int d;<br/>    cin &gt;&gt; d;<br/>    return d;<br/>}</pre>                                              |
| calMax   | float f1<br>float f2 | float        | <pre>float calMax (float f1,float f2){<br/>    if (f1&gt;f2)<br/>        return f1;<br/>    else<br/>        return f2;<br/>}</pre> |

# Parameters Passing: Pass-by-Value

- When a function is invoked, the arguments within the parentheses *by default* are passed using *pass-by-value*
- As only the *value* is passed, the parameter can be:
  - Constant: e.g. `myFunc(10);`
  - Expression: e.g. `myFunc(1+2*3*y);`
  - Variable: e.g. `myFunc(num);`
- Consider the 3<sup>rd</sup> case above (*i.e. argument is a variable*)  
Since only the *value* is passed, so even if you try to change the function's *local* variable, *the original is not affected!*
- In other word, when you call `myFunc(num)`, the function can not change the value of the `num` enclosed inside parenthesis.  
(*unless you use the non-default, pass-by-reference mechanism in p32*)

# Parameters Passing: Pass-by-Value

```
void f (int x){  
    x=4; //we modify the value x to 4...  
        //but do we modify y (of main) at the same time? NO  
    y=4; //syntax error: y is local to main  
}  
  
void main(){  
    int y=3;  
    f(y);  
    cout << y; //print 3, y remains unchanged  
}
```

The variable x and y are local variables.

x is local to f(), so we cannot use x in main().

y is local to main (), so we cannot use y in f().

x and y are two independent variables. When x is modified, y will not be affected.

## What if we *rename* x to y in f()?

(also possible, it's not syntax error!)

```
void f (int y){  
    y=4; //modify y in f(), not the one in main()  
}  
  
void main(){  
    int y=3;  
    f(y);  
    cout << y; //print 3, y remains unchanged  
}
```

In this program, there are two (*unrelated*) variables, both called y. One is defined in f() and another one is defined in main(). In f(), y in f() is modified. However, y in main() is not affected.

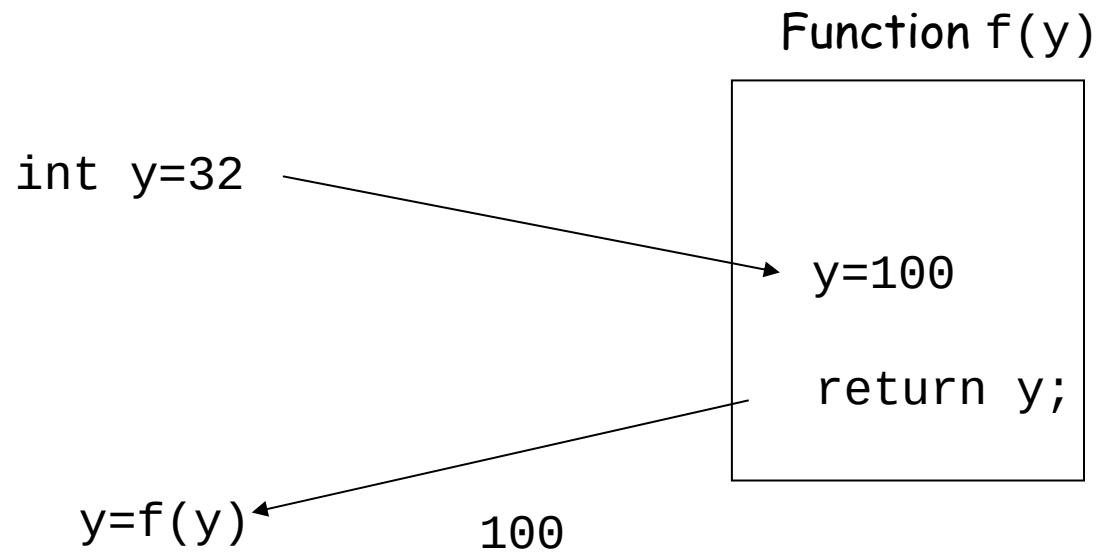
## Then how to let f ( ) modify y?

```
int f (int x){  
    x=4; //we modify the value x to 4  
    //Do we modify y at the same time? NO  
    return x;  
}  
  
void main(){  
    int y=3;  
    y=f(y);  
    cout << y; //print 4  
}
```

If pass-by-value is used, the only way is to assign the *return value* of f( ) to y

After the function call, y gets a value of 4

# Return value



We assign the return value of `f(y)` to the variable `y`.  
What is `y`? Ans: 100

# The return statement

- When a `return` is encountered, the value of the (optional) expression after the keyword `return` is sent back to the calling function
- Also, all statements after `return` are skipped (*i.e. exit the function immediately*)

- Syntax:

```
return expression;  
return;
```

- Example:

```
return (a+b*2);
```

- Note: you can only write `return` (*without the following expression*) if the function is of type `void`

## Example: findMax

Suppose we need to write a function `findMax`, which **return** the larger value among the two input integers.

E.g.

When  $x > y$ , the expression `findMax(x, y)` should evaluate to a value of  $x$

```
cout << findMax(4, 3); //print 4 (first parameter)
```

When  $y > x$ , the expression `findMax(x, y)` should evaluate to a value of  $y$

```
cout << findMax(3, 4); // print 4 (second parameter)
```

# Function definition

```
int findMax (int n1, int n2) {  
    if (n1 > n2)  
        return n1;  
    else  
        return n2;  
}
```

Also ok to write: *(without else)* why?

```
int findMax (int n1, int n2) {  
    if (n1 > n2) return n1;  
    return n2;  
}
```

The return type of the variable is **int**

There are two arguments: **n1** and **n2**, separated by a comma.

IMPORATNT: The type of EACH variable should be specified individually.

Error: `int findMax (int a, () b);`

IMPORATNT: When using branching like if/switch, let's make sure that all paths give a return value!

## Calling findMax()

```
int max;  
int x=3;  
int y=4;  
max=findMax(x, y);
```

The value of this expression is 4. Assign 4 to max

The variable max will hold the value of x when  $x > y$ .  
Otherwise, max will hold the value of y.

## Flow of control

```
void main(){  
    int max;  
    int x=3;  
    int y=4;  
    max=findMax(x, y);  
}
```

```
int findMax (int n1, int n2) {  
    if (n1>n2)  
        return n1;  
    else  
        return n2;  
}
```

When findMax() is called,  
the value of x(3) is copied to the variable n1  
the value of y(4) is copied to the variable n2

## Finding the max of 3 numbers, i, j, k

```
int i,j,k;  
int max;  
cin >> i >> j >> k;  
  
//find the max of i, j, k  
___ = findMax(____, ____);  
___ = findMax(____, ____);  
  
cout << "max is " << max;
```

## Answer

```
int i,j,k;  
int max;  
cin >> i >> j >> k;  
  
//find the max of i, j, k  
max = findMax(i, j);  
max = findMax(max, k);  
  
cout << "max is " << max;
```

Alternatively, just chain them up and write as:

```
cout << "max is " << findMax(i, findMax(j,k) );
```

## What is the output of the following program?

Variables in function are unrelated to the variables in main

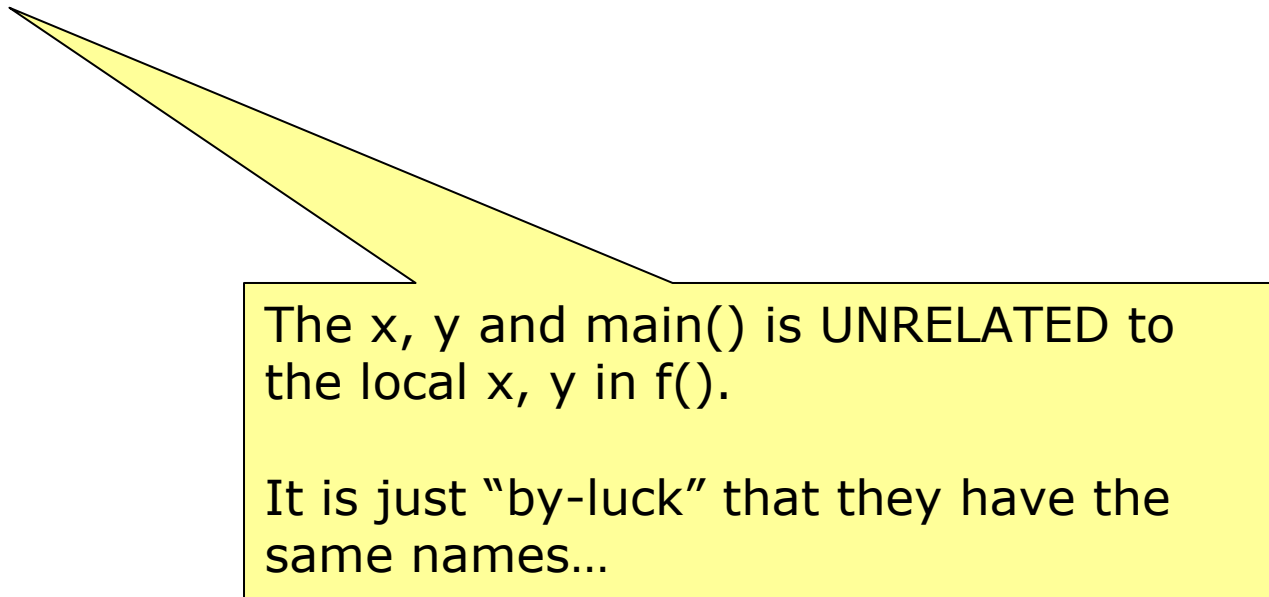
```
void f(int y, int x){  
    cout << "x=" << x << endl;  
    cout << "y=" << y << endl;  
}
```

```
void main(){  
    int x=3, y=4;  
    f(x, y);  
}
```

# Answer

x=4;

y=3;



The x, y and main() is UNRELATED to the local x, y in f().

It is just "by-luck" that they have the same names...

## Exercise

Rewrite the `radius.cpp` (*on next page*) by defining two additional functions.

`areaOfCircle :`

Given a radius `r`, return the area of a circle

`PrintResult :`

Given a radius `r` and area `a`, generate the output

# Original radius.cpp

```
#include <iostream>
#include <iomanip>
using namespace std;
const double PI=3.141592653589793;
void main()
{
    int radius;
    float area;
    cout << "Radius:";
    cin >> radius;
    area = PI * radius * radius;
    cout << fixed << setprecision(2);
    cout << "Area = " << PI << "*" << radius << "*" << radius;
    cout << "=" << area << endl;
}
```

## Sample input/output

```
Radius:5
Area = 3.14 * 5 * 5 = 78.54
```

# Answer

```
#include <iostream>
#include <iomanip>
using namespace std;
const double PI=3.141592653589793;

//define the function areaOfCircle

//define the function printResult

void main(){
    int radius;
    float area;
    cout << "Radius:";
    cin >> radius;
}
```

# Answer

```
#include <iostream>
#include <iomanip>
using namespace std;
const double PI=3.141592653589793;
float areaOfCircle(int r){ /*r=radius*/
    float result;
    result= PI * r* r;
    return result;
}

void printResult(int r, float a){ /*r=radius; a=area*/
    cout << "Area = " << PI << "*" << r << "*" << r << "=" << a <<endl;
}

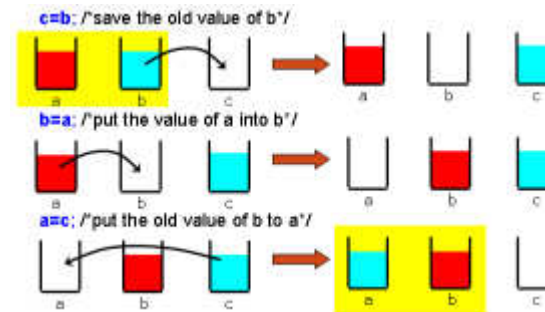
void main(){
    int radius;
    float area;
    cout << "Radius:";
    cin >> radius; /*input 5*/
    area=areaOfCircle(radius);
    printResult(radius, area);
}
```

# Parameters Passing: Pass-by-Reference

- Parameters pass to a function **can be updated** inside the function
- In function **definition**: Add '**&**' in front of the parameter(s)
- When **calling** the function: As usual... '**&**' is not needed, but..
- Argument must be variable, it cannot be constant / expression
- More details will be given in later weeks (Pointer)

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

```
void main(){  
    int x=1,y=3;  
    cout << "x:"<<x <<" ,y:"<<y<<endl;  
    swap(x,y);  
    cout << "x:"<<x <<" ,y:"<<y<<endl;  
}
```



# Parameters Passing: arrays

- When passing an array to a function, we only need to specify the array name

```
void f(int x[20]){  
    ...  
}
```

```
void main(){  
    int y[20];  
    f(y); //Only the name y is needed!
```

```
    f(y[0]); //ERROR!, type mismatch:  
              // y[0] is a single integer, not array!!  
}
```

*(Note: writing f(y[]) is also WRONG!)*

# Parameters Passing: arrays

if the content of `a[i]` is modified in function, the modification **will persist** even after the function returns (*i.e. it **automatically** uses pass-by-reference without asking you!*) *Rationale: reduce copying of gigantic data*

```
void f(int a[3]){
    cout << a[0] <<endl; //1 is printed
    a[0]=10;
}

void main (void) {
    int a[3]={1,2,5}; //an array with 3 elements
    f(a); //calling f() with array a
    cout << a[0] <<endl; //10 is printed
}
```

**Only need to input the array name!**

# Defining and calling functions

Correct

```
void f(){  
}  
  
void main(){  
    f();  
}
```

Syntax **Error**

```
void main(){  
    f(); //f() is undefined (yet)  
}  
  
void f(){  
}
```

A function should be defined before use.

But what if *func1()* internally calls *func2()* and *func2()*, in turns, internally calls *func1()* ?

In what sequence should the functions be defined?

# Function Prototype

- C++ language allows us to define the function prototype first and then write the function later
- Function prototype
  - Specifies the function **name**, **input** and **output** type.
  - Names of parameters are optional

e.g. `int f(int a, int b);`  
e.g. `int f(int, int);`
- The function **body** of the f() function can be defined separately later

# Function Prototype

```
void f (void);
```

```
void main(){  
    f();  
}
```

```
void f(){  
    //define f() here  
}
```

```
int findMax (int, int);
```

```
void main(){  
    int x=findMax (3,4);  
}
```

```
int findMax (int n1, int n2){  
    //define findMax() here  
}
```

Function prototype

Function body

# Summary

- Functions simplifies the logic by breaking down the big problem into smaller parts
- Syntax: `return_type Function_name(paramaters) {...}`
- The keyword `return` is used to:
  - Quit a function, and
  - Report the answer to the caller
- Function prototype/body must declared before it can be used.
- Parameters by default uses `pass-by-value`, therefore updating the variable in function *do not affect the original*.
- A function may optionally uses `pass-by-reference (&)`, which directly uses the original variable.