

CS319: Scientific Computing (with C++)

## More on functions. Recursion.

Week 3: 9am and 4pm, 25 Jan 2017



Source: <http://xkcdsw.com/1105>

- 1 Recall... Functions
  - E.g, Prime or Composite?
- 2 void functions
- 3 Pass-by-value
- 4 Function overloading
- 5 A detailed example
- 6 Default values
- 7 Bitwise operators
- 8 Recursion

- Every C++ program has at least one function: `main()`
- A function consists of two main parts:
  - ▶ Function “header” or **prototype** which gives
    - the function’s return value data-type, or `void` if there is no return value; and
    - the parameter list data-types, or `void` if there are no parameters.
  - ▶ **Function definition**: the code statements that are executed when the function is called. These are delimited by “curly” braces `{...}`.
- **Function definition syntax**:

```
ReturnType FnName (type1 var1, type2 var2, ...)
{
    statements
}
```

argument list.

Eg

int

Add (int a,  
int b)

return type.

- ▶ `ReturnType` is the data-type of the value returned by the function.
- ▶ `FnName` the identifier by which the function is called.
- ▶ A parameter list, with data-types (`int`, `float`, etc) and variable names. The variables are local to the function.

To demonstrate this, we'll revisit the last example from Week 2:

`01IsComposite.cpp`. This program has a function called `IsComposite`

- It takes a single `int`, `i`, as its only parameter;
- In turn, checks  $k = 2, 3, \dots, k = i$ , to see if the remainder on dividing  $i$  by  $k$  is zero, i.e., if  $(i\%k) == 0$ . If that is true, for any  $k$ , then  $k$  is returned.
- If there is no such  $k$ , then return 0.

`01IsComposite.cpp`

```
30 int IsComposite(int i)
31 {
32     int k;
33
34     for (k=2; k<i; k++)
35         if ( (i%k) == 0)
36             return(k);
37
38     // If we get to here, then i has no divisors between 2 and i-1
39     return(0);
40 }
```

this is the name of the variable  
within the function

## Calling the IsComposite function:

01IsComposite.cpp

```
12 int main(void )
13 {
14     int i;
15
16     std::cout << "Enter a natural number: ";
17     std::cin >> i;
18
19     if (IsComposite(i)) evaluates as 'true' if not zero.
20         std::cout << i << " is a composite number." << std::endl;
21     else
22         std::cout << i << " is a prime number." << std::endl;
23
24     return(0);
25 }
```

## void functions

"Void" = nothing.

Most functions will return some value. In rare situations, they don't, and so have a `void` argument list.

02Kth.cpp

```
#include <iostream>
void Kth(int i),
int main(void )
14 {
    int i;

    std::cout << "Enter a natural number: ";
18     std::cin >> i;

    std::cout << "That is the ";
20     Kth(i);
22     std::cout << " number." << std::endl;

24     return(0);
}
```

## void functions

notice that there is no "return" statement-

02Kth.cpp (continued)

```
26 // FUNCTION KTH
27 // ARGUMENT: single integer
28 // RETURN VALUE: void (does not return a value)
29 // WHAT: if input is 1, displays 1st, if input is 2, displays 2nd,
30 // etc.
31 void Kth(int i)
32 {
33     std::cout << i;
34     i = i%100;
35     if ( ((i%10) == 1) && (i != 11))
36         std::cout << "st";
37     else if ( ((i%10) == 2) && (i != 12))
38         std::cout << "nd";
39     else if ( ((i%10) == 3) && (i != 13))
40         std::cout << "rd";
41     else
42         std::cout << "th";
43 }
```

} notice composite  
if  
statements.

## Pass-by-value

In C++ we need to distinguish between

- a variable
- the value stored in it.

The classic example is function that

- takes two **integer** inputs, **a** and **b**;
- after calling the function, the values of **a** and **b** are swapped.

### 03SwapByValue.cpp

```
4 #include <iostream>
   void Swap(int a, int b);
   int main(void )
8 {
    int a, b;

    std::cout << "Enter two integers: ";
12    std::cin >> a >> b;

14    std::cout << "Before Swap: a=" << a << ", b=" << b << std::endl;
    Swap(a,b);
16    std::cout << "After Swap: a=" << a << ", b=" << b << std::endl;

18    return(0);
}
```

*This does not pass the variables a & b to Swap, but just their values.*

## Pass-by-value *add & before x & y*

```
void Swap(int x, int y)
{
    int tmp;

    tmp=x;
    x=y;
    y=tmp;
}
```

### This won't work.

We have passed only the *values* stored in the variables *a* and *b*. In the `swap` function these values are copied to local variables *x* and *y*. Although the local variables are swapped, they remained unchanged in the calling function.

What we really wanted to do here was to use **Pass-By-Reference** where we modify the contents of the memory space referred to by *a* and *b*. This is easily done...

...we just change the declaration and prototype from

```
void Swap(int x, int y) // Pass by value
```

to *x is a variable, but &x is its address.*

```
void Swap(int &x, int &y) // Pass by Ref
```



## Function overloading

C++ has certain features of **polymorphism** – for example, where two different functions can have the same name, so long as they have different argument lists.

This is called **function overloading**.

As a simple example, we'll write two functions with the same name: one that swaps the values of a pair of **ints**, and that other that swaps a pair of **floats**. (Later in the course, we'll see how to do this with **templates**.)

04Swaps.cpp

```
6 #include <iostream>
8 // We have two function prototypes!
  void Swap(int &a, int &b);
10 void Swap(float &a, float &b);
```

argument lists  
are different.

## Function overloading

04

~~12~~ Swaps.cpp (continued)

```
12 int main(void)
13 {
14     int a, b;
15     float c, d;
16
17     std::cout << "Enter two integers: ";
18     std::cin >> a >> b;
19     std::cout << "Enter two floats: ";
20     std::cin >> c >> d;
21
22     std::cout << "a=" << a << ", b=" << b <<
23         ", c=" << c << ", d=" << d << std::endl;
24     std::cout << "Swapping ...." << std::endl;
25
26     Swap(a,b);
27     Swap(c,d);
28
29     std::cout << "a=" << a << ", b=" << b <<
30         ", c=" << c << ", d=" << d << std::endl;
31
32     return(0);
33 }
```

## Function overloading

02Swaps.cpp (continued)

```
34 void Swap(int &a, int &b)
35 {
36     int tmp;
37
38     tmp=a;
39     a=b;
40     b=tmp;
41 }
42
43 void Swap(float &a, float &b)
44 {
45     float tmp;
46
47     tmp=a;
48     a=b;
49     b=tmp;
50 }
```

definitely a  
candidate  
for templates !

## Function overloading Each function has a signature.

What does the compiler take into account to distinguish between overloaded functions?

C++ takes the following into account:

- Type of arguments. So `void Sort(int, int)` is different from `void Sort(char, char)`.
- The number of arguments. So `int Add(int a, int b)` is different from `int Add(int a, int b, int c)`.

But not

- Return values. For example, we cannot have two functions `int Convert(int)` and `float Convert(int)` since they have the same argument list.
- user-defined types (using `typedef`) that are in fact the same. See, for example, `050verloadedConvert.cpp`.

Signature: function name + Arg list.

## A detailed example

---

In the following example, we combine two features of C++ functions:

- Pass-by-reference,
- Overloading,

We'll write two functions, both called `Sort`:

- `Sort(int &a, int &b)` – sort two integers in ascending order.
- `Sort(int list[], int n)` – sort the elements of a list of length *n*.

The program will make a list of length 8 of random numbers between 0 and 39, and then sort them using **bubble sort**.

(See notes from lecture for full description).

## A detailed example

06Sort.cpp (i)

```
6 #include <iostream>
  #include <stdlib.h> → Standard C library : for rand()
8 const int N=8;
10 void Sort(int &a, int &b);
   void Sort(int list[], int length); } overloaded swap functions
12 void PrintList(int x[], int n);
```

output the values  
in the array x,  
with some formatting.

## A detailed example

06Sort.cpp (ii)

```
14 int main(void )  
15 {  
16     int i, x[N];  
17  
18     for (i=0; i<N; i++)  
19         x[i]=rand()%40;  
20  
21     std::cout << "The list is:\t\t";  
22     PrintList(x, N);  
23     std::cout << "Sorting..." << std::endl;  
24  
25     Sort(x,N);  
26  
27     std::cout << "The sorted list is:\t";  
28     PrintList(x, N);  
29     return(0);  
30 }
```

*N is fixed to be 8.*

*returns a (pseudo) random int  
between 0*

*& RAND\_MAX, a  
system variable.*

## A detailed example

06Sort.cpp (iii)

```
32 // Arguments: two integers
33 // return value: void
34 // Does: Sorts a and b so that a<=b.
35 void Sort(int &a, int &b)
36 {
37     if (a>b)
38     {
39         int tmp;
40         tmp=a;
41         a=b;
42         b=tmp;
43     }
44 }

45 // Arguments: an integer array and its length
46 // return value: void
47 // Does: Sorts the 1st n elements of x
48 void Sort(int x[], int n)
49 {
50     int i, k;
51     for (i=n-1; i>1; i--)
52         for (k=0; k<i; k++)
53             Sort(x[k], x[k+1]);
54 }
```

Compare with Swap



## A detailed example

```
62 void PrintList(int x[], int n)
63 {
64     for (int i=0; i<n; i++)
65         std::cout << x[i] << " ";
66     std::cout << std::endl;
67 }
```

Finished here 100m.

## Default values

In C++, one can also define functions that have assigned default values:

```
int mult(int a, int b=1, int c=1) // from 07Mult.cpp
{
    return(a * b * c);
}
```

This means that, if the user fails to provide the second and third arguments to the function, it is assumed that they are both 1.

### Example

```
std::cout << "mult(1) = " << mult(1); a set to 1; b & c default to 1
std::cout << "mult(1,2) = " << mult(1,2);
std::cout << "mult(1,2,3) = " << mult(1,2,3);
```

note: if a default value is given for argument  $k$ , it must also be given for argument  $k+1, k+2$ , etc..

a set to 1, b to 2  
& c defaults to 1.

Our next example is in `08Binary.cpp`, and uses some *bitwise operators*. These relate to the logical operators you may have seen in CS304.

First we'll look at a function to convert from decimal to binary:

returns the binary form of a (decimal) integer  
08Binary.cpp

```
std::string Int_to_Binary(int a)
{
    std::string A=""; //empty string .
    for (int i=(int)log2(a); i>=0; i--)
    {
        if ( a >= pow(2,i))
        {
            A=A+"1";
            a=a-pow(2,i);
        }
        else
            A=A+"0";
    }
    return(A);
}
```

of 1s & 0s.  
check if coef of  $2^i$  should be zero or 1.

We'll return to a recursion-based implementation later...

## Bitwise operators

Then we'll look at the calling part (modified from the actual code to simplify formatting):

### 08Binary.cpp

```
int main(void)
{
    int a, b, c;

    std::cout << "Input two integers: ";
    std::cin >> a >> b;
    std::cout << "You entered: " << a << " and " << b << std::endl;

    std::cout << a << " = " << Int_to_Binary(a) << std::endl;
    std::cout << b << " = " << Int_to_Binary(b) << std::endl;

    c = a^b;
    std::cout << "XOR: a^b = " << c << " = " << Int_to_Binary(c);
    c = a&b;
    std::cout << "AND: a&b = " << c << " = " << Int_to_Binary(c);
    c = a|b;
    std::cout << " OR: a|b = " << c << " = " << Int_to_Binary(c);

    return(0);
}
```

## Recursion Also: Merge Sort...

Many problems in scientific computing can be solved by replacing the problem by a similar but simpler one, and solving that instead.

Here are a few very simplistic examples:

- Suppose we want to compute  $x = a^b$ , where  $b$  is a positive integer. We could first compute  $a^{b-1}$ , and then set  $x = \underline{(a)(a^{b-1})}$ . The process can be repeated:

$$x = a (a (a^{b-2})) = a (a (a (a^{b-3})))$$

etc. with  $a^0 = 1$

- Suppose we want to compute  $x = n!$ , where  $n$  is a positive integer. We could first compute  $(n-1)!$ , and then compute  $x = (n)(n-1)!$ .

Both these are candidates for computation by recursion.

Better Examples: Binomial Coefs:  $\binom{a}{b} = \frac{a!}{b!(a-b)!}$

Use  $\binom{a}{b} = \binom{a-1}{b} + \binom{a-1}{b-1}$ . Pascal's Identity-

# Recursion

## 09Power.cpp

```
float Power(float a, unsigned int b); // compute a to the power of b

int main()
4 {
    float a, c;
6    int b;

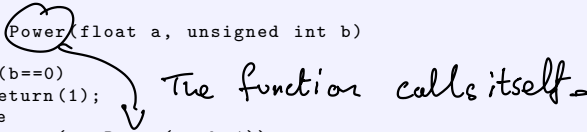
    std::cout << "Input a float, a, and nonnegative integer, b: ";
    std::cin >> a >> b;
10    std::cout << "You entered: a=" << a << " and b=" << b << std::endl;

12    c = Power(a,b);

14    std::cout << a << " to the power of " << b << " is " << c << std::endl;
    return(0);
16 }

18 float Power(float a, unsigned int b)
    {
20     if (b==0)
        return(1);
22     else
        return( a*Power(a, b-1));
24 }
```

The function calls itself



## Recursion

As mentioned above, we can write a recursive decimal-to-binary converter. Here it is below. *Can you work out how it works?* ?

### 10BinaryAgain.cpp

```
4 // A simple example of a recursive algorithm:  
4 // converting from decimal to binary  
4 // Based on Shapira "Solving PDEs in C++", Section 1.18  
  
8 #include <iostream>  
8 #include <math.h>  
  
10 int Binary(int a); // return the binary representation of a  
  
12 int main(void)  
13 {  
14 .  
15 .  
16 .  
17 }  
  
20 int Binary(int a)  
21 {  
22     if (a<=1) If a is 0 or 1 it is the same in binary & decimal.  
23         return(a);  
24     else  
25         return(10*Binary(a/2) + a%2);  
26 }
```