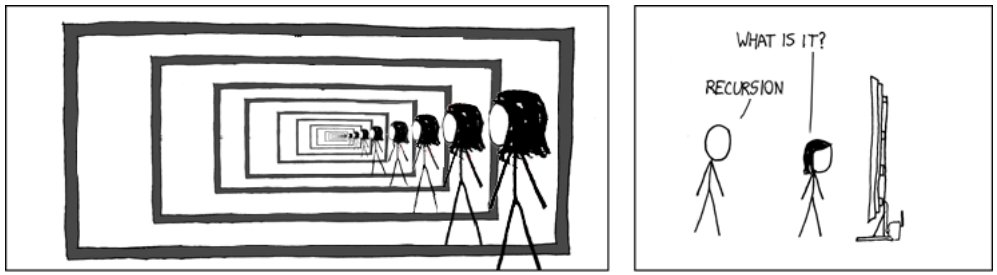
CS319: Scientific Computing (with C++)

More on functions. Recursion.

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



Source: http://xkcdsw.com/1105

Recall... Functions

- 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
}
```

  - `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 $k = 2, 3, \ldots, k - i$, 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)
   {
32   int k;

34   for (k=2; k<i; k++)
       if ( (i%k) == 0)
36       return(k);

38   // If we get to here, then i has no divisors between 2 and i-1
     return(0);
40 }
```

Calling the `IsComposite` function:

01IsComposite.cpp

```
12 int main(void )
   {
14   int i;

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

     if (IsComposite(i))
20     std::cout << i << " is a composite number." << std::endl;
     else
22     std::cout << i << " is a prime number." << std::endl;

24   return(0);
   }
```

void functions

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 )
{
  int i;

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

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

  return(0);
}
```

void functions

02Kth.cpp (continued)

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

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

```
#include <iostream>
void Swap(int a, int b);

int main(void )
{
  int a, b;

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

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

  return(0);
}
```

Pass-by-value

```
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

```
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 `int`s, and that other that swaps a pair of `float`s. (Later in the course, we'll see how to do this with `template`s.)

04Swaps.cpp

```
#include <iostream>

// We have two function prototypes!
void Swap(int &a, int &b);
void Swap(float &a, float &b);
```

Function overloading

04Swaps.cpp (continued)

```
12 int main(void)
   {
14     int a, b;
       float c, d;

       std::cout << "Enter two integers: ";
18     std::cin >> a >> b;
       std::cout << "Enter two floats: ";
20     std::cin >> c >> d;

22     std::cout << "a=" << a << ", b=" << b <<
         ", c=" << c << ", d=" << d << std::endl;
24     std::cout << "Swapping ...." << std::endl;

26     Swap(a,b);
       Swap(c,d);

       std::cout << "a=" << a << ", b=" << b <<
30       ", c=" << c << ", d=" << d << std::endl;

32     return(0);
   }
```

Function overloading

04Swaps.cpp (continued)

```
34 void Swap(int &a, int &b)
   {
36   int tmp;

38   tmp=a;
     a=b;
40   b=tmp;
   }

   void Swap(float &a, float &b)
44 {
     float tmp;

     tmp=a;
48   a=b;
     b=tmp;
50 }
```

Function overloading

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, `05OverloadedConvert.cpp`.

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)

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

const int N=8;

void Sort(int &a, int &b);
void Sort(int list[], int length);
void PrintList(int x[], int n);
```

A detailed example

06Sort.cpp (ii)

```
14 int main(void )
   {
16    int i, x[N];

18    for (i=0; i<N; i++)
        x[i]=rand()%40;

      std::cout << "The list is:\t\t";
22    PrintList(x, N);
      std::cout << "Sorting..." << std::endl;

      Sort(x,N);

      std::cout << "The sorted list is:\t";
28    PrintList(x, N);
      return(0);
30 }
```

A detailed example

06Sort.cpp (iii)

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

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

A detailed example

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

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);
std::cout << "mult(1,2) = " << mult(1,2);
std::cout << "mult(1,2,3) = " << mult(1,2,3);
```

Bitwise operators

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:

08Binary.cpp

```
   std::string Int_to_Binary(int a)
46 {
      std::string A="";

      for (int i=(int)log2(a); i>=0; i--)
50    {
         if ( a >= pow(2,i))
52       {
            A=A+"1";
54          a=a-pow(2,i);
         }
56       else
            A=A+"0";
58    }
      return(A);
60 }
```

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

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 $x = (a)(a^{b-1})$. The process can be repeated:

- 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.

Recursion

09Power.cpp

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

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

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

  c = Power(a,b);

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

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

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

```
// A simple example of a recursive algorithm:
// converting from decimal to binary
// Based on Shapira "Solving PDEs in C++", Section 1.18

#include <iostream>
#include <math.h>

int Binary(int a);   // return the binary representation of a

int main(void)
{
.
.
.
}

int Binary(int a)
{
  if (a<=1)
    return(a);
  else
    return(10*Binary(a/2) + a%2);
}
```