CS101 Computer Programming and Utilization

Milind Sohoni

May 13, 2006

The story so far ...

- We have written some non-trivial programs
- We have seen various control flows.
- We have seen multi-dimensional arrays and the `char` data type.
- Finally, we saw how to get formatted output.

Functions

We come now to an important conceptual step called `functions`. Again www.cplusplus.com/doc/tutorial for reference.

Motivation for Functions

In programming, functions usually arise from three basic conceptual requirements.

- As a piece of code which appears to be repeated.
- As a utility which should be viewed as an independent task.
- As a conceptual understanding leading to a solution to the problem.

We will see examples of all three.

Problem 1

Write a program to solve the equation $Ax = b$, when A is an invertible 2×2-matrix.

```
#include <iostream.h>
float det(float a,float b,
               float c,float d)
{
    return (a*d-b*c);
}
int main()
{
  float a11,a12,a21,a22,b1,b2,d1
  cin >> a11 >> a12 >> a21 >> a2
  cin >> b1 >> b2;
  d=det(a11,a12,a21,a22);
  if (d==0)
    cout<< "error";
  d1=det(b1,a12,b2,a22);
  d2=det(a11,b1,a21,b2);
  cout << d1/d << " " << d2/d <<
}
```

Motivation for Functions

We use Kramer's rule:

$$x_1 = \frac{det\left(\begin{bmatrix} b1 & a12 \\ b2 & a22 \end{bmatrix}\right)}{det\left(\begin{bmatrix} a11 & a12 \\ a12 & a22 \end{bmatrix}\right)}$$

$$x_2 = \frac{det\left(\begin{bmatrix} a11 & b1 \\ a12 & b2 \end{bmatrix}\right)}{det\left(\begin{bmatrix} a11 & a12 \\ a12 & a22 \end{bmatrix}\right)}$$

Input/Output

```
Input
1 2 1 3
3 4
Output
1 1
```

```
#include <iostream.h>
float det(float a,float b,
                float c,float d)
{
    return (a*d-b*c);
}
int main()
{
  float a11,a12,a21,a22,b1,b2,d1
  cin >> a11 >> a12 >> a21 >> a2
  cin >> b1 >> b2;
  d=det(a11,a12,a21,a22);
  if (d==0)
    cout<< "error";
  d1=det(b1,a12,b2,a22);
  d2=det(a11,b1,a21,b2);
  cout << d1/d << " " << d2/d <<
}
```

AxB.c Execution Flow

float det(float a,....)

{

body

}

prevline;

d=det(a11,a12,a21,a22)

nextline;

The variables are copied in order and the output copied back.

```
#include <iostream.h>
float det(float a,float b,
               float c,float d)
{
    return (a*d-b*c);
}
int main()
{
  float a11,a12,a21,a22,b1,b2,d1
  cin >> a11 >> a12 >> a21 >> a2
  cin >> b1 >> b2;
  d=det(a11,a12,a21,a22);
  if (d==0)
    cout<< "error";
  d1=det(b1,a12,b2,a22);
  d2=det(a11,b1,a21,b2);
  cout << d1/d << " " << d2/d <<
}
```

```
#include <iostream.h>
float det(float a,float b,
               float c,float d)
{
    return (a*d-b*c);
}
int main()
{
  float ...
  cin >> ...
  cin >> b1 >> b2;
  d=det(a11,a12,a21,a22);
  if (d==0)
    cout<< "error";
  d1=det(b1,a12,b2,a22);
  d2=det(a11,b1,a21,b2);
  cout << ...
}
```

- Note that the function is specified before the main and used after its specification.
- The function det has four inputs and one output. Each input has a given data-type and so does the output. When called, the correct order and type must be used.

```
#include <iostream.h>
float det(float a,float b,
               float c,float d)
{
    return (a*d-b*c);
}
int main()
{
  float ...
  cin >> ...
  cin >> b1 >> b2;
  d=det(a11,a12,a21,a22);
  if (d==0)
    cout<< "error";
  d1=det(b1,a12,b2,a22);
  d2=det(a11,b1,a21,b2);
  cout << ...
}
```

- Note that the function is specified before the main and used after its specification.
- The function det has four inputs and one output. Each input has a given data-type and so does the output. When called, the correct order and type must be used.
- Control temporarily goes to the function. Upon the return statement, control returns to the line after the calling statement. Thus, for each call,
  - The point of return, is stored.
  - The input arguments are copied out, and
  - upon, return, the output argument copied into the calling variable.

Rootfinding again

`rootfinding.c`

We modify the earlier `cubicroot.c` to find the roots of $\sin(x)$ or for that matter any function.

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

float f(float x)
{ // ANY FUNCTION HERE
  return(sin(x));
}
```

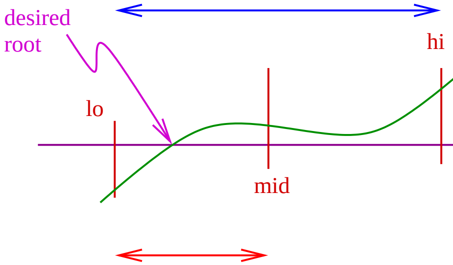
Rootfinding again

`rootfinding.c`

We modify the earlier `cubicroot.c` to find the roots of $\sin(x)$ or for that matter any function.

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

float f(float x)
{ // ANY FUNCTION HERE
  return(sin(x));
}
```



Rootfinding again

rootfinding.c

We modify the earlier `cubicroot.c` to find the roots of $\sin(x)$ or for that matter any function.

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

float f(float x)
{ // ANY FUNCTION HERE
  return(sin(x));
}
```

INPUT
3 4 0.00001
OUTPUT
3.1416

```
int main()
{
  float lo,hi,mid,fhi,fmid, flo,
  cout << "low high tolerance" <
  cin >> lo >> hi >> tol;
  mid=(lo+hi)/2;
  flo=f(lo);fhi=f(hi);fmid=f(mid
  while (fabs(fmid)>tol)
  {
   if (flo*fmid >0)
   {
     lo=mid; flo=fmid;
   }
   else
   {
     hi=mid; fhi=fmid;
   };
   mid=(lo+hi)/2;fmid=f(mid);
 }; // end of while
 cout << mid << "\n";
 return 0;
```

Recursion

The function achieved a separation of the evaluation of the function from its root finding. Thus the two activities can be separately implemented.
We have seen the use of function to

- Avoid duplication of code.

  `AxB.c`

- Separate two concepts.

  `rootfinding.c`

Recursion

The function achieved a separation of the evaluation of the function from its root finding. Thus the two activities can be separately implemented.
We have seen the use of function to

- Avoid duplication of code.

  `AxB.c`
- Separate two concepts.

  `rootfinding.c`

  AND NOW
- think differently!

Compute N!

`factorial.c`

Recursion

The function achieved a separation of the evaluation of the function from its root finding. Thus the two activities can be separately implemented.
We have seen the use of function to

- Avoid duplication of code.

  `AxB.c`

- Separate two concepts.

  `rootfinding.c`

  AND NOW

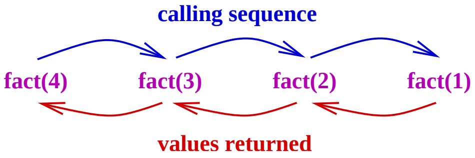
- think differently!

Compute N!

`factorial.c`

```
#include <iostream.h>
#include <math.h>
int fact(int x)
{
  if (x==1) return(1);
  else return(x*fact(x-1));
}
int main()
{
  int N;
  cout << "N?";
  cin >> N;
  cout << fact(N);
}
```

- The function `fact` calls itself, but with a smaller argument.
- It is clear that $N! = N * ((N-1)!)$ and the code imitates that.
- Note that `fact` has one part which stops the recursion, i.e, when `x==1`. The other calls `fact(x-1)`.
- The calling sequence is the order in which `factorial` are executed and the input arguments.
- The values are returned in the reverse order. Thus the call to `fact(5)` is complete only after `fact(4)` has returned a value.




```
#include <iostream.h>
#include <math.h>
int fact(int x)
{
  if (x==1) return(1);
  else return(x*fact(x-1));
}
int main()
{
  int N;
  cout << "N?";
  cin >> N;
  cout << fact(N);
}
```

Old Problem

Count the number of sequences of length n over 0-1 with NO consecutive zeros.

a_n = strings as above but ending in 0
b_n = strings as above but ensing in 1

Our interest is in $a_n + b_n$. We have:

$$\begin{aligned} a_n &= b_{n-1} \\ b_n &= a_{n-1} + b_{n-1} \end{aligned}$$

Old Solution

Using Arrays `int A[10], B[10]`.

AnBn.c

Old Problem

Count the number of sequences of length n over 0-1 with NO consecutive zeros.

a_n = strings as above but ending in 0
b_n = strings as above but ensing in 1

Our interest is in $a_n + b_n$. We have:

$$\begin{aligned} a_n &= b_{n-1} \\ b_n &= a_{n-1} + b_{n-1} \end{aligned}$$

Old Solution

Using Arrays `int A[10], B[10]`.

```
#include <iostream.h>
#include <math.h>
int B(int x);
int A(int x)
{
  if (x==1) return(1);
  else return(B(x-1));
}
int B(int x)
{
  if (x==1) return(1);
  else return(A(x-1)+B(x-1));
}
int main()
{
  int N;
  cin >> N;
  cout << A(N)+B(N);
}
```

AnBn.c

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

int B(int x);

int A(int x)
{
  if (x==1) return(1);
  else return(B(x-1));
}
int B(int x)
{
  if (x==1) return(1);
  else return(A(x-1)+B(x-1));
}
int main()
{
  int N;
  cin >> N;
  cout << A(N)+B(N);
```

Many things to note here:

- The programs for A and B mimic their mathematical definitions.
- There are two functions calling each other recursively.
- Note the peculiar single line header of B . If this were absent, the program would not compile.

```
AnBn.c: In function ‘int A(in
AnBn.c:6: error: ‘B’ undeclar
         (first use this func
AnBn.c: In function ‘int B(in
AnBn.c:9: error: ‘int B(int)
         prior to declaratio
```

- This just means that B occurs in A but its identity is not declared beforehand.

AnBn.c

WARNING: Recursion is simpler to implement but

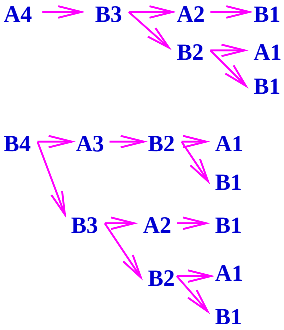
- Harder to debug.
- Generally Inefficient.

In this case:

- A4 calls B3 which will call A2, B2 and so on.
- B4 will call A3,B3. However, the A4 call of B3 is forgotten and cannot be re-used.

```
#include <iostream.h>
#include <math.h>
int B(int x);
int A(int x)
{
  if (x==1) return(1);
  else return(B(x-1));
}
int B(int x)
{
  if (x==1) return(1);
  else return(A(x-1)+B(x-1));
}
int main()
{
  int N;
  cin >> N;
  cout << A(N)+B(N);
}
```

In Total



We see that there are:

- 5 calls to B1, 3 calls to A1.
- 3 calls to B2 and 2 calls to A2.
- 2 calls to B3 and 1 call to A3.

Thus, there is a lot of duplication in effort. The array code is much much more efficient.

AnBn.c

```
#include <iostream.h>
#include <math.h>
int B(int x);
int A(int x)
{
  if (x==1) return(1);
  else return(B(x-1));
}
int B(int x)
{
  if (x==1) return(1);
  else return(A(x-1)+B(x-1));
}
int main()
{
  int N;
  cin >> N;
  cout << A(N)+B(N);
}
```

Summary

- Functions have three typical uses:
  - save code repetition.
  - separate distinct parts of the code
  - conceptualize mathematical definitions
- The function must be specified before the main program. It must have input arguments and an output value.
- The calling program must respect these attributes.
- Control temporarily passes to the function and returns to the next statement.

Problems

- Let R_1 and R_2 be two rectangles in a plane. Show that there is a line which will cut both rectangles into equal halves. Write a program to input two sets of four points. Then (i) check that each set marks a rectangle, and (ii) compute the cut above.
- Write a program which takes in a positive integer and prints one factorization of it into primes.