Functions

Ajit Rajwade

Refer: Chapter 9 of the book by Abhiram Ranade

Revisiting our bisection method code

- We considered a function `f(x) = x^3-5`.
- We had to evaluate the function for many different values: `xL, xR, xM`.
- We had to each time write the same formula with `xL, xR, xM` respectively.
- See next slide which contains the code.
- This may seem okay, but consider what would happen if `f(x)` were a much more complicated function, potentially requiring conditional statements if different formulae were applied to different portions of the domain.
- Moreover, you could have needed to use this formula not just 3 times, but say 50 times.
- How do you make your code more compact?

Bisection method for roots of f(x)=x^3-5

```
main_program{
double xL = 1, xR = 2; // initial interval
double epsilon = 0.001; // precision of the root’s value
bool flagL, flagR;
double xM;
bool flagM;
flagL = xL*xL*xL-5 > 0; flagR = xR*xR*xR-5 > 0; // signs of f(xL) and f(xR)
if (flagL != flagR)
{
     while (xR - xL > epsilon) // until the required precision is reached
     {
          xM = (xR+xL)/2; // interval midpoint
          flagM = xM*xM*xM-5 > 0;
          if (flagM != flagL) xR = xM; // signs of f(xM) and f(xL)
          else if (flagM != flagR) xL = xM;
     }
}
cout << “the root is ” xL <<;
}
```

C++ functions to the rescue

- We need to write a C++ function to compute the value of `f(x)` given `x`.
- We can call it multiple times from the main program.
- This reduces code redundancy, and reduces scope for errors.
- The function will have the following form:

```
double f ( double x)
```

- `f` is the name of the function and it takes parameter `x` of type double as input. It also returns a value of type double.
- With this modification, the same code will look like what you will see on the next slide.

```
double f (double x) { return x*x*x-5; } // function declaration

main_program{
double xL = 1, xR = 2; // initial interval
double epsilon = 0.001; // precision of the root’s value
bool flagL, flagR, flagM; double xM;
flagL = f(xL) > 0; flagR = f(xR) > 0; // signs of f(xL) and f(xR)
if (flagL != flagR)
{
     while (xR - xL > epsilon) // until the required precision is reached
     {
          xM = (xR+xL)/2; // interval midpoint
          flagM = f(xM) > 0;
          if (flagM != flagL) xR = xM; // signs of f(xM) and f(xL)
          else if (flagM != flagR) xL = xM;
     }
}
cout << “the root is ” xL <<;
}
```

General function syntax

- The general syntax of a function declaration is:

```
typename_return_value function-name (typename1 parameter1, typename2
parameter2, …, typenameN parameterN)

{

Body;  // includes a statement of the type return return_value;

}
```

- A function may take as input some `N` parameters, which may be of different data-types. `N` could be equal to 0.
- The body of the function can contain a sequence of statements but it must contain a `return` statement, returning a variable of type `typename_return_value`.
- A declared function is called from another function including `main` using the syntax: `returnval = function-name (parameter1, parameter2, …, parameterN);` where `returnval` has type `typename_return_value`.

Function for GCD

```
int gcd (int m, int n){

while (m%n != 0) {

    int remainder = m%n;

    m = n; n =
remainder;

}

return n;

}
```

```
main_program{

int a=10, b=4, c = 39,
d=72;

cout << gcd(a,b) << endl;

cout << gcd(c,d) << endl;

}
```

Function for Factorial

```
int factorial (int n){

if (n == 0) return 1;

int prod=1;

for (int i = n; i > 0; i=i-1){

    prod *= i;

}

return prod;

}
```

```
main_program{

int a=10, b=4;

cout << factorial(a) << endl;

cout << factorial(b) << endl;

}
```

`void` functions

- A function that **does not return any quantity** is called a `void` function.
- Functions that draw shapes are often written as `void` functions because they do not need to return anything. See next slide for the function `void drawPolygon (int numsides)` which can be called from the main program.
- Another example: a function which takes as input a number and which just prints out the digits of the number.

`void` functions: examples

```
void drawPolygon (int numsides){

turtleSim();

for(int i = 0; i < num_sides; i++){

forward(50); left(360/num_sides);

}

wait(5);

}

int main(){

drawPolygon (15);

}
```

```
void printLastDigit(int n)

{

cout << n%10;

}

int main(){

printLastDigit(10);

}
```

Note that `turtleSim()`, `forward()`, `left()`, `wait()` are all functions – in fact they are all `void` functions!

Function execution

- Refer to the `gcd` function. What happens when the program comes to `gcd(a,b);` inside main?
- The arguments to the call are evaluated. Here it implies just getting the values of `a` and `b` from memory, but evaluation would be necessary if you had instead supplied expressions as parameters to `gcd`, for example if you had called `gcd(a+b, c+d);`
- The calling function (`main` in this case - note that `main` is also a function!) gets suspended, and will be resumed later, and at that point it will continue from the point where it was suspended.
- The variables local to the `main` function are stored in area of memory, called the **activation frame**.
- The called function (`gcd` here) gets its own **activation frame**, where it can store its own variables (called **local variables**). In the case of `gcd(int m, int n)`, the stack will store local variables `m` and `n`. The values of `a` and `b` respectively will be copies of `m` and `n` in memory locations **different** from those of `m` and `n`.
- The activation frame of all functions are part of a larger area of memory called the **stack**. We say that the AF of the calling function is *pushed onto the stack*, before control passes to the called function.

Function execution

- The values of `a` and `b` are copied to `m` and `n` respectively.
- Sample activation frames for `main`, `gcd(a,b)` and `gcd(c,d)` are shown on the next slide.
- The body of the function is executed. Any space to be reserved later on for variables of the function, must be given inside the activation frame. For example, the variable `remainder`.
- The statements of body get executed until you encounter the `return` statement. The value of `return_expression` (if any) is computed and sent back to the calling function. This value is in fact copied into the appropriate variable of the calling function. For example, for the call `gcd(a,b)`, the value 12 is returned.
- The activation frame of the called function is destroyed, i.e. that portion of memory is marked available for usage by other functions (also called *"popped off the stack"*).
- The calling function resumes from the point where it was suspended. The returned value (if any) is used as required in this function.
- The values of the local variables of the calling function are obtained from the stack.

Activation frame of main	**Activation frame of gcd(a,b)**
a: 36	m:36
b: 24	n:24
c: 99	
d:47	

After copying arguments

Activation frame of main	**Activation frame of gcd(a,b)**
a: 36	m:24
b: 24	n:12
c: 99	remainder: 12
d:47	

At the end of the first iteration of the loop in gcd

Note: even though the values of `m` and `n` inside `gcd` change, the values of `a` and `b` in the main function remain intact. This is desirable, but also has limitations, which we will see later.

You can similarly work out activation records for a few other functions, or for gcd(c,d) in the same example.

Nested function calls

- Any given function can call other functions, which in turn can call more functions as well.
- In fact, `main_program` itself is a function. If you remove simplecpp and move to traditional C++, the main function looks like the following:

```
int main (){

// body

return 0;

}
```

- An example of nested function calls for the bisection method is presented on the next slide. You have function `f`, and you have another function which implements the bisection method, and which is called from `main`.
- Note: each program must have a `main` function. You cannot call `main` from within itself or any other function.
- The g++ compiler allows you not to return anything from `main`, even if it is declared as `int main () {}`. This special leeway is not given to any other function.

```
double f (double x) { return x*x*x-5; } // function declaration
double bisection_root (double xL, double xR, double epsilon){
// initial interval is given by [xL, xR], given as input
// precision of the root's value is also given as input
bool flagL, flagR, flagM; double xM;
flagL = f(xL) > 0; flagR = f(xR) > 0; // signs of f(xL) and f(xR)
if (flagL != flagR)
{
    while (xR - xL > epsilon) // until reqd. precision is reached
    {
        xM = (xR+xL)/2; // interval midpoint
        flagM = f(xM) > 0;
        if (flagM != flagL) xR = xM; // signs of f(xM) and f(xL)
        else if (flagM != flagR) xL = xM;
    }
}
return xL;
}
```

```
// for C++
int main (){
double xL = 1, xR = 2,
epsilon=0.001;
double rootval;

rootval =
bisection_root(xL,xR,e
psilon);

return 0;
}
```

```
// for simplecpp
main_program () {
double xL = 1, xR = 2,
epsilon=0.001;
double rootval;

rootval =
bisection_root(xL,xR,e
psilon);
}
```

Call by value: and its limitations

- All function calls we have seen so far fall under the category "**call by value**".
- That is, the **values** of the parameters are passed on by the calling function to the function being called.
- You saw this in the activation tables drawn out for the `gcd` function.
- Now consider the following function which attempts to swap the values of two numbers `a` and `b`:

```
void myswap (int a, int b){

int temp = a; a = b; b = temp;
return;

}
```

```
int main () {

int x = 10, y = 20;

myswap(x,y);

cout << x <<”,”<<y;

return 0;

}
```

Call by value: and its limitations

- What do you think will be the output of the program?
- The values of `x` and `y` will remain **unchanged** even after the call to `myswap`, because those values were copied into the parameters `a` and `b`.
- But the values of a and b were never copied back into `x` and `y`.
- This is one **limitation** of call by value.
- As it stands, a function can return only one value at a time.
- What if you need to **return two or more values**? There is no provision for this.
- For example, if you wanted to write a function to take (x,y) coordinates of a point as input and produce their polar coordinates.

To the rescue: Call by reference

- If you want changes in the function parameters made during a function call to reflect even after the function call is over, you have to use **call by reference**.
- This is done by specifying an ampersand (`&`) before the parameter name when the function is written out.
- When a parameter (say `p`) is called by reference, no separate storage for `p` is allocated in the activation frame of the function.
- Instead the reference parameter directly refers to the **original** memory location of that variable.
- Hence changes made to a parameter passed by reference within a function reflect in the original variable itself.

Call by reference: swap

```
void myswap (int& a, int& b){

int temp = a; a = b; b = temp;
return;

}

// int &a, int &b is also valid
// syntax
```

```
int main () {

int x = 10, y = 20;

myswap(x,y);

cout << x <<”,”<<y;

return 0;

}
```

Output: 20,10
If a and b were called by value, the output would have been 10,20

Call by reference: Cartesian to polar

```
void cartesianToPolar (double x, double y, double &r, double
&theta){
    r = sqrt(x*x + y*y);
    theta = atan2(y/x);
}
int main (){
double x = 1.0, y = 1.0, theta, r;
cartesianToPolar (x,y,r,theta); cout << r << " " << theta;
}
```

Since `r` and `theta` are passed by reference, their values will be available inside `main` **even after** a call to `cartesianToPolar` is completed.

Call by reference: some remarks

- The manner in which a function is called remains the same, whether or not one or more of the function parameters were called by value or called by reference.
- The change is only in the definition of the function, except for the following point.
- If you had to call by value, you can use constant values (literals) as arguments. For example, you could call a function: `myswap(10,20)` if the arguments of myswap were by value.
- You cannot use such a call with literals, if those arguments were by reference.

Global Variables

- A global variable is a variable that is common to all functions in a program, including `main`.
- It should be declared before the `main` function, for example as follows:

```
#include <iostream>
using namespace std;
int x = 10, y; double z, w = 4.5; char a, b = ‘X’, c;
int main (){
// body of main function
}
```

- Global variables can be used or modified inside any function.

Global Variables

- Consider a local variable `int x;` declared inside a function (say `main()`, but could be any other).
- Let us suppose `x` is also a global variable.
- When you use `x` inside `main`, you are using the local variable `x`, and not the global variable.
- If you want to use a global variable with the same name as a local variable, you use `::x` instead of `x` (but such name conflicts should ideally be avoided).

```
#include <iostream>
using namespace std;
int x = 10, y; double z, y = 4.5; char a, b = 'X', c;
void f(){
    x = x+1; // updates the global value of x
    cout << x; // prints the global value of x
}
int main (){
int x = 5;
x = x+1; // updates the local value of x
cout << x; // prints the local value of x
f();
::x = ::x+1; // update the global value of x
cout << ::x;
}
```

The output of this program is 61112

Global Variables

- The use of global variables is generally **discouraged**, as it is bad programming practice and makes large programs difficult to maintain.
- They should be used **rarely**, only when certain variables are genuinely used across a large number of functions.
- For example `const double pi = 3.142857;` is a good example of a global variable which can be used in all mathematical functions.
- Another example: `double square_root_two = 1.414;` can be used in many geometrical functions.

Functions Example: Virahanka Numbers

- A sequence of numbers where the n-th number $V_n = V_{n-1} + V_{n-2}$, i.e. sum of (n-1)th and (n-2)the numbers.
- This is called a **recurrence relation**.
- Commonly called Fibonacci numbers, but discovered earlier by Virahanka.
- The first two Virahanka numbers are equal to 1.
- Applying the recurrence relation, we have $V_3 = V_2 + V_1 = 2$, $V_4 = V_3 + V_2 = 3$, $V_5 = V_4 + V_3 = 5$, and so on.
- Given the recurrence relation, we now write a function to compute the `n`-th Virahanka number.
- We will study later on as to why someone thought of such a sequence of numbers.

```
int virahanka (int n){
if ( n == 1) return 1;
if (n == 2) return 1;
int prev = 1, curr = 1,val;
for(int i = 0; i < n-2;i++){
val = prev + curr;
prev = curr;
curr = val;
}
return val;
}
```

```
int main(){
int n,v;
cout << "Which Virahanka
number do you want to
calculate?";
cin >> n;
v = virahanka(n);
cout << "The Virahanka
number at position " << n <<
"is " << v;
}
```

Here, we have followed the convention that the first two Virahanka numbers are 1,1. In some books, the convention followed is that the first two Virahanka numbers are either 0,1 or else 1,2. The essence of the Virahanka sequence remains the same in all cases, and the code will change very slightly.