An Introduction to Programming though C++

Abhiram G. Ranade

Lecture 6.3

Ch. 12: Functions: Advanced Topics

Outline

- Passing functions as arguments to other functions
  - Function parameters
  - Lambda expressions
- Default values to parameters
- Function overloading

The function ***bisection*** to find square root of 2

```
double bisection(double xL, double xR, double epsilon){
// Input precondition: sign f(xL) != sign f(xR)
  bool xLisPos = (xL*xL — 2) > 0;
  while(xR — xL >= epsilon){
    double xM = (xL+xR)/2;
    bool xMisPos = (xM*xM — 2) > 0;
    if(xLisPos == xMisPos) xL = xM;
    else xR = xM;
  }
  return xL;
}
```

- If we want to find cube root of 3 should we have to write everything again?
- We should really only need to write again whatever is different for cube root of 3.

A C++ function to find the root of any mathematical function

- Natural idea:
  - pass an extra argument to ***bisection***
  - argument specifies the mathematical function whose root we want to find.
- How do we represent a mathematical function?
  - Natural idea: As a C++ function computing the required value.

```
double f(double x){return x*x – 2;}
double g(double x){return x*x*x – 3;}
```

- "Passing a mathematical function to another function":
  - Can we pass one C++ function as argument to another C++ function?

What we would like to write

```
double f(double x){
  return x*x — 2;
}
double g(double x){
  return x*x*x - 3;
}
int main(){
  cout << bisection(1,2,0.0001,f) << endl;
  cout << bisection(1,3,0.0001,g) << endl;
}
// should print out square root of 2,
// and cube root of 3.
```

How to pass a function ***h*** to another function ***B***

- Suppose ***h*** has declaration**:**

  return-type h(parameter1-type, ...);
- In parameter list of B we need to put the type of ***h.***
- The type of ***h*** is

  function<return-type(parameter1-type,...)>
- The functions that we want to pass to ***bisection*** take a single ***double*** argument and return a ***double***.
- Hence their type is ***function<double(double)>***
- Thus bisection will have the declaration:

double bisection(double xL, double xR, double epsilon,
function<double(double)> h);

- Inside bisection we can write calls to h.

The function **bisection**

```
#include <functional> //Defines std::function<...>
double bisection(double xL, double xR, double epsilon,
                 function<double(double)> h){
  bool xLisPos = (h(xL) > 0);
  while(xR - xL >= epsilon){
    double xM = (xL+xR)/2;
    bool xMisPos = h(xM) > 0;
    if(xLisPos == xMisPos) xL = xM;
    else xR = xM;
  }
  return xL;
}
```

Demo

- manyRoots.cpp

Exercise

The following function somehow accumulates the numbers 1 through n. Write a main program which calls this twice with appropriate functions so that the sum of the numbers and then the product is returned.

```
int accumulate(int n, function<int(int,int)> f){
  int res = 1;
  for(int i=2; i<=n; i++)
    res = f(res,i);
  return res;
} // works only for positive n.
```

Exercise

- Write a functions ***plot*** which plots a given function ***f*** on the graphics screen.

```
void plot(std::function<double(double)> f,
          double x0, double y0,
          double x1, double y1);
```

- The function plot must leave 10 % margin on all sides of the canvas. It should plot the part of the function which lies in the rectangle with diagonal (x0,y0) and (x1,y1).

What we discussed

- We often want to write functions that operate on functions.
  - E.g. functions that find roots of other functions or plot functions.
- Such functions can be written conveniently because C++ provides a mechanism to pass a function to another function.
- Must include header file ***<functional>*** in order to use the feature we discussed.
- Next: Lambda Expressions

🏓

Lambda expressions (from Lisp)

- Lambda expression = an expression which represents a (nameless) function.

Example: **[](double x){return x*x – 2;}**

General form: **[](parameter list){body}**

- You can pass arguments to the function and it is evaluated like an ordinary function call.

```
[](double x){return x*x – 2;}(3.5)
```

- This evaluates to 12.25 – 2 = 10.25.
- You can also pass it as a parameter to functions such as **bisection**:

```
cout << bisection(1,2,0.0001,
                  [](double){return x*x – 2;}) << endl;
```

- Key benefit: you do not need to create a function f just to pass to bisection.
- What you are passing to bisection is written in the call itself – better readability.

Remarks

General form: ***[](parameter-list){body}***

- Parameter list is comma separated as usual.
- Body is like the body of any function.
- Return type is not stated explicitly. It is inferred by the C++ compiler.
- In case it is not possible to infer the type, you can specify it explicitly too

[](parameter-list)->return type {body}

Example: ***[]()-> int {return 1;}***

- 1 could be short, long, or int. So must specify return type.

More general lambda expressions

“Write a program that reads a number from the keyboard and prints its square root using the bisection method.”

- Can be written by modifying the bisection function.
- But suppose we do not have the code of bisection.
- We can write this as follows:

```
double z; cin >> z;
cout << bisection(0,z+1,0.0001,
      [z] (double x){return x*x – z;}) << endl;
```

- The z in [] says that the lambda expression will capture the value of z from the function in which the lambda expression is written.
  - Without this, the body of the lambda expression cannot refer to variables defined outside.
- You can capture many variables by putting a comma separated list in [].

Demo

- sqrtZ.cpp

Exercise

What does the code below do?

```
double ssum(function<double(int)> f, int n){
  double sum = 0;
  for(int i=0; i<=n; i++) sum = sum + f(i);
  return sum;
}
int main(){
  cout << ssum([](int i){return i*i*i;},
               100)<<endl;
}
```

Exercise

The following function is meant to draw a square, but instead of using the normal forward command, it uses a command supplied as an argument.

```
void square(function<void()> fd){
  repeat(4){fd(); right(90);}
}
```

Write a main program to from which you supply a function which will cause a dashed square of side length 100 do be drawn.

Basically, this will let you draw “decorated” squares. By supplying the appropriate function, you should be able to draw a square in which the turtle goes off from the line, draws something, but gets back on track again.

More complex variable capture

- By placing names of variables in [], you enable the values of the variables to be used inside the body of the lambda expression.
- But you can also allow the lambda expression to access the value of the variable at the time the lambda expression is evaluated.
- Discussed in the book, but will not be considered in this course.

What we discussed

- Lambda expressions evaluate to nameless functions
- We can evaluate these by supplying arguments, or pass them to other functions.
- Lambda expressions can also capture the values of variables defined in the function in which the lambda expression appears.
- Next: Giving default values to parameters

🏓

Default values to parameters

- Suppose we are drawing lots of squares, most of which are black, but for some we want to specify a colour.
- Wouldnt it be nice if we can say,
  - "if I do not tell you what colour to use, make it black?"
- C++ allows this:
  - One or more parameters occurring at the end of the parameter list can be given default values.
  - Suppose there are n parameters, and you have specified default values for last m.
  - In the call you must give at least n-m arguments.
  - If you give n-m+r arguments, then the last m-r will take default values.

Imprinting a disk

```
void disk(double x, double y, double r=10,
          Color col=COLOR("black"),
          bool fill=true){
  Circle c(x,y,r);
  c.setColor(col);
  c.setFill(fill);
  c.imprint();
}
```

Demo

- disk.cpp

Exercise

The k-norm of a math vector (x,y,z,...) is defined to be the kth root of $x^k + y^k + \ldots$ Most commonly the 2 norm is used.

Define a function norm for 2 dimensional vectors (x,y). The call norm(x,y,k) should give the kth root of $x^k + y^k$. The call norm(x,y) should give the 2 norm i.e. square root of $x^2 + y^2$.

You may note that the function pow(x,r) returns x^r for any r.

What we discussed

- How to give default values to the last parameters in the parameter list.
- Note that if you want to specify a default value for the rth parameter, you must specify a default value for all subsequent parameters as well.
- This often provides some convenience.
- Next: Overloading functions, conclusion of lecture sequence

🏓

Overloading functions

- C++ allows you to define multiple functions with the same name, provided they have different argument lists.
- Consider a function to compute the area of a graphical object.
- It will be nice to give the name Area to the function even though the arguments could be a circle or a rectangle.
- Just do it! It is allowed in C++.

Function overloading demo

- Area.cpp

Exercise

Write an additional ***Area*** function so that if one ***double*** argument is given it returns the area of a circle with that radius; if two ***double*** arguments are given it returns the are of a rectangle with those side lengths.

Concluding Remarks

- It is often useful to pass functions as arguments in a function call.
  - Common example: writing a single function that calculates the roots, or the numerical integral, ... of math functions.
  - Functions can be passed by specifying their name, or by giving a lambda expression.
  - There is a C style method also, discussed in book.
- C++ allows default values for the last arguments in a function.
- C++ also allows defining many functions with same name, provided the parameter types are different.

🏓🏓