CS 1002 Programming Fundamentals
Lecture 13-14
(05, 10th Oct 2022)

Functions

National University
Of Computer and Emerging Sciences

We've seen already that…

- C++ programs can have
  - Variables – by declaring and using them
    - char c; int i; float cost; double money;
  - Flow control – code is executed conditionally
    - if, if-else, multiway if-else if-else, switch
  - Flow control – loops are repeatedly executed
    - Also conditionally
    - while, do-while, for (will cover later)
  - Flow control statements allow us to do one thing if the condition is true. But what if we need multiple things?

Quiz # 1

- Write a function that converts degrees to Radians
  - float radians(int degree)

- // 3.14 (actually pi) radians per 180 degrees

Blocks of statements

Statements in a program are grouped:

- with curly braces `{ }` for `if`, `else`, and even loops
- Blocks are treated like a single thing after a flow control statement.
- Blocks define a new scope, so local variables defined in the block, stay in the block.
- Imagine using named blocks.

This program has one long, complex function containing all of the statements necessary to solve a problem.


```
int main()
{
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
}
```

In this program the problem has been divided into smaller problems, each of which is handled by a separate function.


```
int main()
{
   statement;
   statement;
   statement;
}
```

main function

```
void function2()
{
   statement;
   statement;
   statement;
}
```

function 2

```
void function3()
{
   statement;
   statement;
   statement;
}
```

function 3

```
void function4()
{
   statement;
   statement;
   statement;
}
```

function 4

Function
Define Once, Use Many Times

A named block of code to perform a function

- May return an answer
- or just run a group of statements that perform a task

Some functions are available 'for free' with 'the system'

These functions are available in libraries and are brought into programs using #include directive

Pre-defined functions

Import functions with
`#include<cmath>`

Example:

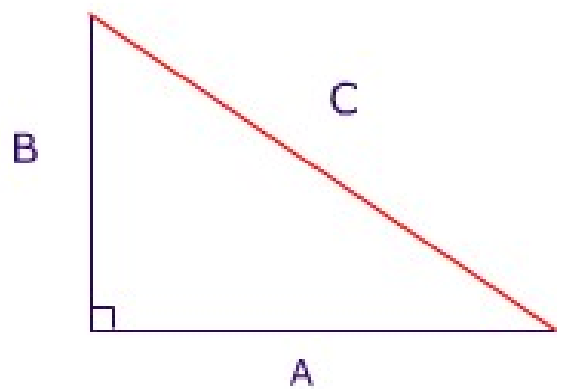
```
float y = sqrt(9);
```

- `sqrt(x)` is a function that returns $\sqrt{x}$
- `abs(x)` is a function that returns |x|
- `ceil(x)` is a function that returns $\lceil x \rceil$
  - 'Round up'. If decimal, next higher int, otherwise x.
- `floor(x)` is a function that returns $\lfloor x \rfloor$
- `pow(x,y)` is a function that returns x^y

Pythagorean Theorem

- Given A and B, how do we get C (using C++)?
- Let's do it now – what is the expression?

$$A^2 + B^2 = C^2$$

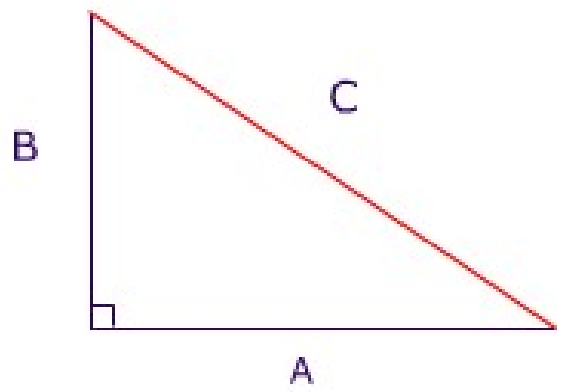


- How do we solve for C?

Pythagorean Theorem

- Given A and B, how do we get C (using C++)?
- Let's do it now – what is the expression?

$$A^2 + B^2 = C^2$$



- double C = sqrt(pow(A, 2) + pow(B, 2));

More pre-defined functions: Random numbers

Import function with

```
#include<cstdlib>
```

`rand()` returns a 'pseudo-random' number between `0` and `RAND_MAX`

`RAND_MAX and rand() are defined in <cstdlib>`

`RAND_MAX` = $2^{31} - 1$.

rand( )

Every call to rand() will give a new result

```
//Remember: RAND_MAX == 2,147,483,647
cout << "Picking 3 random numbers (0 to "<< RAND_MAX << "):" << endl;
cout << "Rand#1 = " << rand()  << endl;
cout << "Rand#2 = " << rand()  << endl;
cout << "Rand#3 = " << rand() << endl;
```

Output:

```
Picking 3 random numbers (0 to
2,147,483,647):
Rand#1 = 1804289383
Rand#2 = 846930886
Rand#3 = 1681692777
```

How many of you got the same results???

Pseudo-random

- Do you get the same results as your neighbors?
- How do we fix this?
- srand(time(0)); // initialize with a 'seed' based on the current time. (always different)
  - seconds since Jan 1, 1970 UTC

```
#include  <cstdlib> /* time & rand */
```

Guess My Number

```
#include <iostream>
using namespace std;
#include  <cstdlib> /* time & rand */

int main ()
{
  int iSecret, iGuess;

  srand (time(NULL));                          /* initialize random seed: */

  iSecret = rand() % 2 + 1;                   /* generate secret number */

    cout << "Guess the number : ";
    cin >> iGuess;

   if (iSecret==iGuess)
                    cout <<  "Congratulations!";
   else if (iSecret>iGuess)
                    cout << "The secret number is higher" << endl;
           else
                     cout << "The secret number is lower" << endl;

   return 0;
}
```

Fun with rand()

Let's write a program to 'flip a coin' 5 times.

Let's use the first half of numbers for 'heads'

Else 'tails'

Use Mid to find the middle (= RAND_MAX/2)

How many are heads and how many are tails?

Hints:

Variables to keep: headcount, tailcount, flipValue

Statement to call rand() pre defined function 5 times

If-then to decide if flip is heads or tails.

Smaller random numbers

- Use % and + to scale to desired number range
- Simulate rolling of die:

```
int roll = (rand() % 6) + 1;
```

- Simulate picking 1 of 26 students in our class:

```
int studentNum = ???
```

Making a function
When to do it & why

When

- What might we (or someone) use again?
- A set of logic that might be complicated
- Something we type over & over – think 'function'
  - Saves typing

Why

- Often makes the code more readable
- 'Complicated' logic is in one place – if it has to be corrected (i.e. not in a dozen places)
- Makes sure behavior is consistent

Example: average

```
float average(float x, float y) {
        float result; // good practice
        result = x + y / 2;
        return(result);
}
main( ) {
        int a,b;
        cout << "Give me 2 values: " ;
        cin >> a >> b;  // example  a = 5; int b = 8;
        cout << "Average of " << a << " and " << b ;
        cout << " = " <<  average(a,b) << endl; // call
}
```

How to make our own function

- Identify a set of statements with a single name
  - You name it. Pick something that makes sense!
  - Make it legal. **Same rules as for a variable.**
- Use the 'function name' to run the larger set of statements anywhere in your code
- Determine the 'type' that the function will return.
  - If it is nothing, you can use void.
  - However, often int, float, bool, double (Twice the bits as float! More digits to right of decimal.)

How to <u>use</u> our own function

```
#include <cstdlib> // rand library
#include <iostream>
using namespace std;
int flip ( )
{    … from prior page
}
int main()
{
      int heads=0;
      int numFlips = 5;

      heads = heads + flip();
      heads = heads + flip();
      heads = heads + flip();
      heads = heads + flip();
      heads = heads + flip();

      cout << "Fraction of heads was " << heads / numFlips << endl;
}
```

Professional programmers instead say:

<u>call</u> a function

Challenge

- Write a function “flip” that returns 0 or 1 to imitate a cion flip.
- Use rand() from Cstdlib

More Pre-defined functions

Import functions with
`#include <cmath>`

- `sin(R); //R is radians`
- `cos(R); // and lots more…`

What a hassle! We think in degrees!

Let's make our own function to convert degrees to radians so we never have to think about it again!

Convert Degrees to Radians

```
float toRadians(float degrees)   // note the input is degrees!
                                 // type has to be declared
                                 // can use it inside our function
{
        // given degrees, returns radians
        // 3.14 (actually pi) radians per 180 degrees
        // so each degree  is 3.14/180 radians
        return(degrees * 3.14 / 180);      // For more accuracy, use <cmath> M_PI constant instead.
}
int main()
{
        // print the sin values for angles between 0 and 360 degrees in increments of 5 degrees

        //Example call to our sin function
        cout << "sin(90 degrees) is " << sin(toRadians(90))<<endl;

        return 0;
}
```

drawline

```
int drawline(char c)
// print 20 c characters in a row
{



}
int main()
{
        drawline ('-');   // calling drawline in main
        return(0);
}
```

drawline

```
int drawline(char c)
// print 20 c characters in a row
{



}
// At the beginning, the end, and AFTER every
// multiple of 90 degrees, draw a line
```

Vocabulary: Parameter

In the function <u>declaration</u>

```
double rand( );// 0 parameters
double sin(double radians);    // 1 parameter
double pow(double x, double y);        //2 parameters
// may have MANY parameters.
// Typically 0-4 though
```

Vocabulary: Argument

In the function <u>call</u>

```
rand( );   // 0 arguments
sin(3.14/2);  // 1 argument
pow(2,3);          // 2 arguments
// may have MANY.
// Typically, 0-4 though

drawline('-');    ??
```

ONE Return Value

Are the results of our functions

```
int      rand( );
double sin(R);
double pow(x,y);
```

```
??   drawline('-');
```

Return Type – in Action

```
int  rand100( ) // 0 parameters
{
        int myNum;
        myNum = rand()%100;
        return (myNum);
}

int main()
{
        int oneGuess = rand100(); // Limited use? Can we improve it?
        cout << "I guess you are " << oneGuess << " years old" << endl;
        return(0);
}
```

Function Returns void

```
#include <iostream>
using namespace std;

void drawline(char c)
// print 20 c characters in a row
{
            // guts of function go here
}

int main ()
{
  // print a line with stars !
  drawline('*');
}
```

```
********************
```

Functions – terminology

- Return_type – A function may return a value. The return_type is the type of the return value. Only one value can be returned
- Function_name – actual name of the function.
- Parameters – A parameter is a variable. Values can be passed to functions in an ordered list. The values passed are arguments, the variables receiving them are parameters.
- Arguments – An argument is a value, expression or variable passed to a function when called. Function input.
- Function_body – A block of statements that perform the required task. May have local variables, may have 0 or more return statements depending on return_type.
- Function call – Calling the function runs the function.

Syntax

```
// Function definition
Return_type Function_name ( parameter_list )
{
    // code to implement function
    return Expression_of_return_type;
}
```

Practice: Sin (and drawline) Program

Create 2 functions:

Degrees2Radians: which has input Degrees, and returns Radians

Drawline: which takes a character and numRepetitions and prints the character numRepetitions times, followed by a newline

main

For values between 0 and 360, in 5 degree increments:

calculate and print the value of sin (radians),

After every 90 degrees, print out a line of dashes (minus signs '-')

A Better Function

```
int  randUpTo(int maxNum )       // 1 parameter. Initialized when calling
{
        int myNum = rand() % maxNum;
        return (myNum);
}
int main()
{
        int oneGuess = randUpTo(100); // range is always from 0.
        cout << "I guess you are " << oneGuess << " years old" << endl;
        oneGuess = randUpTo(25);
        cout << "I guess your cat is " << oneGuess << " years old" <<
endl;
        return(0);
}
// What if we want a function that finds a number between min and max.
```

An Even Better Function

```
int  randBetween(int minNum, int maxNum) // 2 parameter.
{
        int myNum = (rand( ) % (maxNum-minNum)) + minNum;
        return (myNum);
}
int main()
{
        int oneGuess = randBetween(40,90);
        cout << "I guess a Professor is " << oneGuess << " years old" <<
endl;
        oneGuess = randBetween(16,70);
        cout << "I guess a Student is " << oneGuess << " years old" <<
endl;
        return(0);
}
```

Parameter Order MATTERS!

```
int  randBetween(int minNum, int maxNum)
{
        …
}

int main()
{
        int oneGuess = randBetween(40,90);

        return(0);
}
```

Example addition Function

```
#include <iostream>
using namespace std;

int addition (int a, int b)
{
  int r;
  r=a+b;
  return r;
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
}
```

```
The result is 8
```

Use it like any number

```
#include <iostream>
using namespace std;

int addition (int a, int b)
{
  int r;
  r=a+b;
  return r;
}

int main ()
{
  int z;
  z = addition (5,3) + 100;
  cout << "The result is " << z;
}
```

```
The result is 108
```

Other functions we can build

Commonly needed, useful code

Perform a function

Example: generate an answer

run a group of statements

Circle area: 3.14 x r x r

Define Once, Use Many Times

Area of a Circle : $A = \pi r^2$

Good name for a function that returns area of circle?

What is the parameter?

What is the return value type?

Function Location

- Must be defined before it is used. (for now).
- Above main()
- We do this so that the compiler knows about the function before it is used.
  - Otherwise, it sees the name of the function but doesn't recognize that it is a function and gives an error.

A "Heads Up" to the Compiler

- Everything in C++ must be declared before it is used.
- A declaration tells the compiler about a symbol. What is it? (e.g. variable, function)
  - Declarations come first
- A definition tells the compiler how it behaves. What does it do? (e.g. statements to execute)
  - Defintions come at the end

Defining a function

Similar to variable

- function <u>declaration</u>
  - must be declared before it is used
  - declaration tells the compiler what it is.
- function <u>definition</u>
  - provides the statements performed by the function
  - definition tells the compiler what it does.

Functions in your C++ file

```
#include<iostream>
using namespace std;

float circleArea(float radius);  // declaration

int main () {
    . . .
    float area_R2=circleArea(2); // usage
    . . .
}

float circleArea(float radius) {  // definition
    float area=3.14*radius*radius;
    return area;
}
```

Function declaration

Establish:

- function name
- output type
- input types and names

<u>Syntax:</u>

```
return_type function(parameter_list);
```

<u>Example:</u>

```
float circleArea(float radius);
// computes area of circle
```

Function definition

Provides the statements performed when function is used

Syntax:

```
return_type fcn_name(input_list){
   statement1;
   . . .
   statementN;
}
```

Example:

```
float circleArea(float radius){
   float area=3.14*radius*radius;
   return area;
}
```

Function use – “function call”

- (If appropriate) can assign output

```
float area_R2 = circleArea(2);
```

- Call types must be consistent with declaration and definition

The return statement

- When function is “called”, information may be expected back

```
float area_R2 = circleArea(2);
```

- `return` specifies what value to give the caller
- Syntax:

```
variable = function(arguments);
function(arguments);
```

Practice: TimeGreeting

```
> ./timeGreetings
What is your name? Joe
What time is it? 0900
Good morning, Joe.
> ./timeGreetings
What is your name? Laura
What time is it? 1400
Good afternoon, Laura.
>
```

Starting Code for timeGreetings.cpp

Get name and time

```
cout << "What is your name? ";
cin >> name;
cout << "What time is it? ";
cin >> time;
```

Practice: Make a function for timeGreetings.cpp

- Make a function TimeToGreeting that has time as a parameter and prints Good Morning, Good Afternoon, Good Evening
- Stretch goal: modify TimeToGreeting to include a character to indicate the language (e.g. 'E' -English, 'S'- Spanish, 'F' - French. (you can used Google translate to get the proper equivalent, or just say "French version of Good Morning"

Modify timeGreetings.cpp

- Stretch goal: modify TimeToGreeting tohave 2 parameters:
  1. A name
  2. A character to indicate the language (e.g. 'E' -English, 'S'-Spanish, 'F' - French. (you can used Google translate to get the proper equivalent, or just say "French version of Good Morning"

  3. Example output:
     1. Bonjour, Marie

Legal Alternate (but WORSE) function declaration

```
float circleArea (float);//poor, but works!
float circleArea(float radius);// much better
```

- Only argument <u>types</u> are absolutely **required** in the declaration // **NOBODY SENSIBLE DOES THIS. Do not do it. Ever.**
- Argument names **highly** recommended
- <u>Parameter names **required** in this class</u>

Call-declaration consistency

- Compiler forces match between call and declaration

```
float final_price(int numItems, float single_cost);
x = final_price(3.43,10); // numItems*single_cost
```

Will force type-conversion: 3.43->3, 10->10.000

- Does not check logical ordering of arguments

```
int sum_range(int min, int max);
a = sum_range(10,3);
```

Will not re-order input: min=10, max=3

Variable scope

Variables declared in a function

- are **local** to that function
- are invisible to all other functions

`int main()` is a function

Remember: What's <u>defined</u> in the function stays in the function.

(Just like blocks { })

```
int newFunc(int a);

int main() {
  int a=5, b, c=5;
  b = newFunc(a);
  cout << a << " " << b << " "
       << c << endl;
  return 0;
}

int newFunc(int a) {
  int c=12;
  return a*5+c;
}
```

What does this code do?

Formal parameters

"Formal parameters" are the variables in the function header

```
float triple(float inNum)
{
    float tripledNum;
    tripledNum=3*inNum;
    return tripledNum;
}
```

Function head

Function body

Formal parameters

- **Local** to the function
- Used as if they were declared in function body – **do not** re-declare in function body
- When function is called, parameters initialized to the values of the arguments in the function call

```
float triple(float inNum)
{
    float tripledNum;
    tripledNum=3*inNum;
    return tripledNum;
}
```

Formal parameter names

- Typically, argument names do not match parameter names

Example declaration:

int mymin(int **oneNum**, int **anotherNum**);

..

Example call:

int x=6; int y=4;

mymin(x,y);

Broader scope: global variables

- Global variables visible to all functions
- Declared outside of all functions
- Must be declared prior to first use

```
#include<iostream>
using namespace std;
const float PI=3.14;
    // visible to main and to areaCircle

// compute area of circle
float areaCircle(float radius);

int main() { ...}
float areaCircle(float radius) {...}
```

More on global variables

- Useful to define global constants
- Very risky to define non-constant global variables.
  - try to keep track of what functions change the variable
- Local variables are not automatically initialized. They must be initialized by programmer.
- Global variables (not constants) are automatically initialized to `0` (numeric) or `NULL` (character) when the variable is defined.

Program 6-19

```
// This program calculates gross pay.
#include <iostream>
#include <iomanip>
using namespace std;

// Global constants
const double PAY_RATE = 22.55;     // Hourly pay rate
const double BASE_HOURS = 40.0;    // Max non-overtime hours
const double OT_MULTIPLIER = 1.5;  // Overtime multiplier

// Function prototypes
double getBasePay(double);
double getOvertimePay(double);

int main()
{
   double hours,           // Hours worked
          basePay,         // Base pay
          overtime = 0.0,  // Overtime pay
          totalPay;        // Total pay
```

Global constants defined for values that do not change throughout the program's execution.

The constants are then used for those values throughout the program.

```
29      // Get overtime pay, if any.
30      if (hours > BASE_HOURS)
31         overtime = getOvertimePay(hours);
```

```
56    // Determine base pay.
57    if (hoursWorked > BASE_HOURS)
58       basePay = BASE_HOURS * PAY_RATE;
59    else
60       basePay = hoursWorked * PAY_RATE;
```

```
75      // Determine overtime pay.
76      if (hoursWorked > BASE_HOURS)
77      {
78         overtimePay = (hoursWorked - BASE_HOURS) *
79                  PAY_RATE * OT_MULTIPLIER;
```