

Introduction to Programming using C++

Lecture Three: Functions

Carl Gwilliam

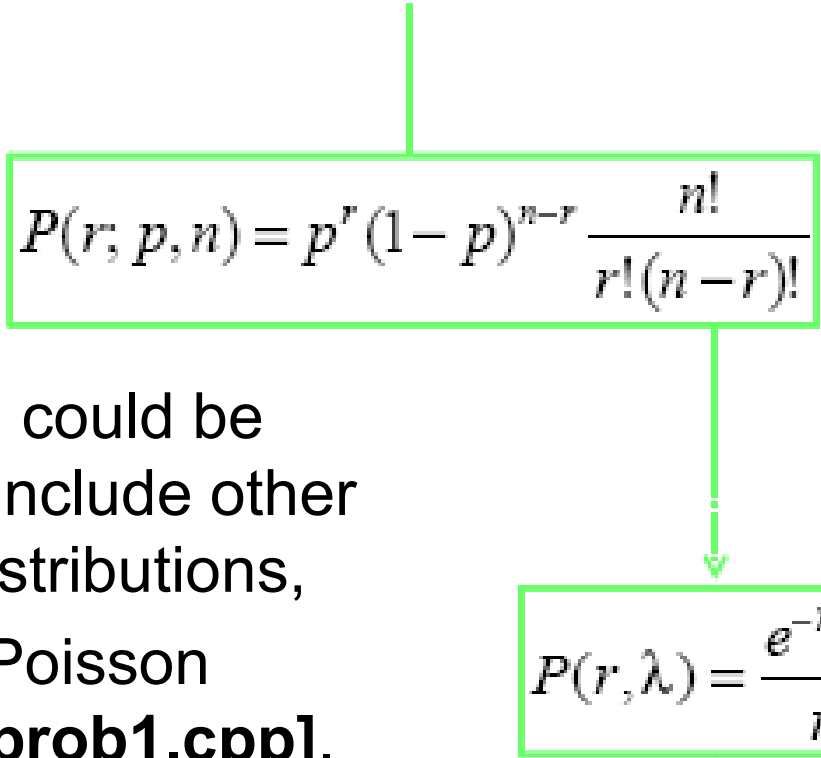
gwilliam@hep.ph.liv.ac.uk

<http://hep.ph.liv.ac.uk/~gwilliam/cppcourse>

Calculating Binomial and Poisson Probability

- Consider question 3(b) from Tutorial 1:

“Generalise the program to allow the user to find the probability of tossing i heads from j throws, where i and j are determined by the user.”


$$P(r, p, n) = p^r (1 - p)^{n-r} \frac{n!}{r!(n-r)!}$$

Binomial

- The program could be extended to include other probability distributions, such as the Poisson distribution [**prob1.cpp**].

$$P(r, \lambda) = \frac{e^{-\lambda} \lambda^r}{r!}$$

Poisson

Function Declaration

```
return_type function_identifer (argument_list)
```

- A function has to be declared before it is called. This is also known as the functions prototype

```
double sin (double x); //declaration before main (in math.h)

int main() {

    double pi = 3.14;

    double y = sin(pi); //variable name need not be the same

} //return value assigned to variable of same type
```

- A function declaration acts as a contract between the author and the user and a function **definition** is an implementation of that declaration.
- Functions in C++ are **strongly** typed.

Writing Functions

- It is desirable to delegate work outside the main block.
- To identify a function use case, looked for a repeated sequence of statements that share some form of commonality.
- I don't care how the work is done, I'm only interested in return value.

Calculate
combinatorics:

$$\frac{n!}{r!(n-r)!}$$

```
double triesfct = 1;
for (int i = 1; i<=tries; i++) triesfct *= i;
double winsfct = 1;
for (int i = 1; i<=wins; i++) winsfct *= i;
float losses = tries - wins;
double lossesfct = 1;
for (int i = 1; i<=losses; i++) lossesfct *= i;
float comb = triesfct / (winsfct * lossesfct);
```

Writing a factorial function

- An $n!$ function can be added to the program (this is needed 3 times from the main for the combinatorics):
- Variable type 'Number' is an example of a **typedef** (see later).

[prob2.cpp]

```
// function prototype (outside main)
double factorial(int n);

int main() {
    // calculate combinations
    Number triesfct = factorial(tries);
    Number winsfct = factorial(wins);
    Number lossesfct = factorial(tries-wins);
    float comb = triesfct / (winsfct * lossesfct);
:
}

double factorial(int n) {
    int fact = 1;
    for (int i = 1; i<=n; i++) fact *= i;
    return fact;
}
```

Switch Statements

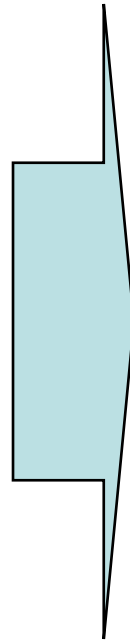
- The **switch** statement evaluates an expression and executes a set of statements depending upon its **value**
- The result of the expression in the switch statement must be of type **integer**.
- If the result does not match any of the **case** labels the **default** block is executed.
- A **break** must be included at the end of each case otherwise the next statement will be executed

```
switch (expression) {  
  case value1:  
    statement1;  
    break;  
  case value2:  
    statement1;  
    :  
    statementN;  
    break;  
  :  
  case valueN:  
    :  
    break;  
  default:  
    :  
}
```

Switch statements for a menu

- If the number of menu options will increase then it is preferable to use a **switch** statement instead of an **if – else if** chain:

```
if (request == 1) {  
    :  
} else if (request == 2) {  
    :  
} else if (request == 3) {  
    :  
} else {  
    :  
}
```



```
switch (request) {  
    case 1:  
        :  
        break;  
    case 2:  
        :  
        break;  
    case 3:  
        :  
        break;  
    default:  
        :  
}
```

- Statements shouldn't be too long or code loses readability

Enumerations

```
enum label {name1 = value1, name2 = value2, .. nameN = valueN}  
enum label {name1, name2, .. nameN};
```

- For a large enough menu system, it can be difficult to recall the correspondence between the menu options and the routines they refer to.
- For this problem **enumerations** can be applied which map an **integer** value to a user defined name.
- If the values are not explicitly defined in the **enum** statement then the first argument will be assigned 0, the second 1 and so on

Enumerations

```
enum DaysOfWeek {mon=1,tue,wed,thr,fri,sat,sun};  
cout << "Enter day (mon=1 to sun=7)" << endl;  
int day;  
cin >> day;  
if (day == thr) {  
    cout << "C++ Lecture at 11am!\n";  
}
```

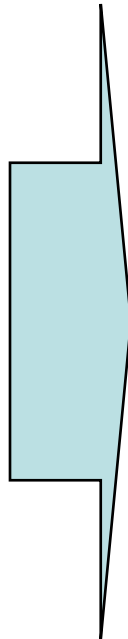
- If you only want to use the enum to refer to constants (but don't plan to use the type to declare variables, function arguments, etc) you can use an **unnamed** enum

```
enum {name1, name2, .. nameN};
```

Enumerations for a menu

- In a menu routine, enumerations can be used in the switch statement to clarify the case expressions.

```
switch (request) {  
  case 1:  
    :  
    break;  
  case 2:  
    :  
    break;  
  case 3:  
    :  
    break;  
  default:  
    :  
}
```



```
enum Menu {mean=1,stdev=2,order=3};  
switch (request) {  
  case mean:  
    :  
    break;  
  case stdev:  
    :  
    break;  
  case order:  
    :  
    break;  
  default:  
    :  
}
```

Typedefs

```
typedef name type;
```

- It is sometimes uncertain which type will be needed for a set of variables

e.g. An **integer** variable storing the value of a factorial calculation will only be useful up to **12!**

- Use **typedef** to aid redefinition of a variable type across an entire program.

```
typedef long double Number;

int main() {

    // calculate combinations

    Number triesfct = 1;

    for (int i = 1; i<=tries; i++) triesfct *= i;

    Number winsfct = 1;

    for (int i = 1; i<=wins; i++) winsfct *= i;

    :

}
```

Writing probability functions

- When identifying function use cases, look for reusable chunks of code that exist to calculate a single value.
- Functions to calculate Binomial and Poisson values can be added to the program.

```
float binomial(float prob, int wins, int tries) {  
    // calculate combinations and then binomial  
    Number triesfct = factorial(tries); //function calling function  
    Number winsfct = factorial(wins);  
    Number lossesfct = factorial(tries-wins);  
    float comb = triesfct / (winsfct * lossesfct);  
    return(pow(prob,wins) * pow(1-prob,tries-wins) * comb);  
} // can perform calculations within the return statement  
float poisson(int r, float mean) {  
    return (exp(-mean) * pow(mean,r))/factorial(r);  
}
```

Calling the probability functions

```
int main() {  
    // get probability, successes and tries  
    // get binomial  
    float binom = binomial(prob,wins,tries);  
    cout << "Binomial probability: " << binom << endl;  
    // get r and mean  
    // get poisson  
    cout << "Poisson probability: " << poisson(r,mean) << endl;  
    return 0;  
} [prob3.cpp]
```

- Now that the probability calculations have been taken outside **main()** these functions have the potential to be called from other programs.

Some questions about functions

- We will spend the rest of the lecture and the next lecture answering the following questions concerning the use of functions:
 - Can my functions be used by other programs?
 - How do I supply a default value for a function argument?
 - Can a function call itself?
 - Do I have to write separate functions for each type, even though the function will be exactly the same?
 - Is it possible to call a function that modifies the input arguments?
 - How do I return more than one value from a function?
 - Can the value of a function variable be made constant throughout the entire program?

Creating function libraries

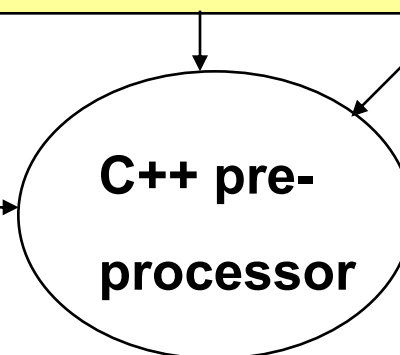
- The pre-processor copies the relevant contents of the **header file** into the translation unit ...

```
// function declarations [proplib.h]
float binomial(float prob, int wins, int tries);
float poisson(int r, float mean);
```

```
#include "proplib.h" [proplib.cpp]
// function definitions
float binomial(float prob, int wins, int tries) {
    // calculate combinations
    // calculate binomial
}
float poisson(int r, float mean) {
    return (exp(-mean) * pow(mean,r))/factorial(r);
}
```

```
#include "proplib.h"
// main program
int main() {
    :
}
```

[prob.cpp]



Translation units:

```
// func declarations
// func definitions
```

```
// func declarations
// main program
```


The C++ program

- In general, a C++ program has source code distributed across multiple files.
- It is useful to separate source code larger than 500+ lines into separate **translation units**.
- To compile code distributed across multiple files include all the translation units in the compilation command.

g++ -Wall -o prob.exe prob.cpp problib.cpp

- The program will not compile unless the translation units defining all the necessary functions are included

Default arguments

```
function_identifier (argument1 = value1, .. ,argumentN = valueN)
```

- It is often useful to supply a default value for a function argument.
- The default is entered in the function prototype.

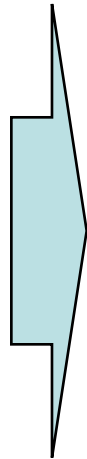
```
double getAngle(double vect1[],double vect2[],int radorDeg = 1);  
double getArea(double PI = 3.14,double radius);  
int main() {  
    double angle = getAngle(vect1,vect2); // get angle in radians  
    double angle = getAngle(vect1,vect2,2); // get angle in degrees  
    // get area of circle  
    double Area1 = getArea(3.14157,3); // ok  
    double Area2 = getArea(,3); // error  
}
```

- If a function takes multiple arguments it is not possible to have a default value for the first argument.

Recursion

- Recursion routines are a powerful and elegant tool in programming.

```
Number factorial(int n) {  
    int fact = 1;  
    for (int i = 1; i<=n; i++)  
        fact *= i;  
    return fact;  
}
```



```
Number factorial(int n) {  
    if (n > 1) {  
        return (n * factorial(n-1));  
    } else {  
        return 1;  
    }  
} // function calls itself
```

- Recursive functions are commonly used in computing mathematical algorithms (factorials, series) and in manipulating large data structures.

Changing function argument types

- The compiler forbids calling a function with different input and return types than declared in the function prototype.

```
int conversion(int celsius);
int main() {
    int celsius;
    cout << "Temperature (C)?" << endl;
    cin >> celsius;
    float fahr = conversion(celsius);
    //error: float returned but assigned to int
}
// func converts celsius into fahrenheit
int conversion(int celsius) {
    return (int) (((9.0/5.0) * celsius) + 32);
}
```

- It seems necessary to declare & define separate funcs to accept input and return values of different types.

Changing function argument types

- Functions **i_conversion** and **f_conversion** are added to the code to account for the two different types.

```
int i_conversion(int celsius);    //return int
float f_conversion(float celsius); //return float
int main() {
    if (precis == 2) { //what precision is required
        float celsius;
        cout << "Temperature (C)?\n";
        cin >> celsius;
        float fahr = f_conversion(celsius);
    } else {
        int celsius;
        cout << "Temperature (C)?\n";
        cin >> celsius;
        int fahr = i_conversion(celsius);
    }
}
```

Function Overloading

- A function is distinct if the function name OR the number and type of arguments is unique.

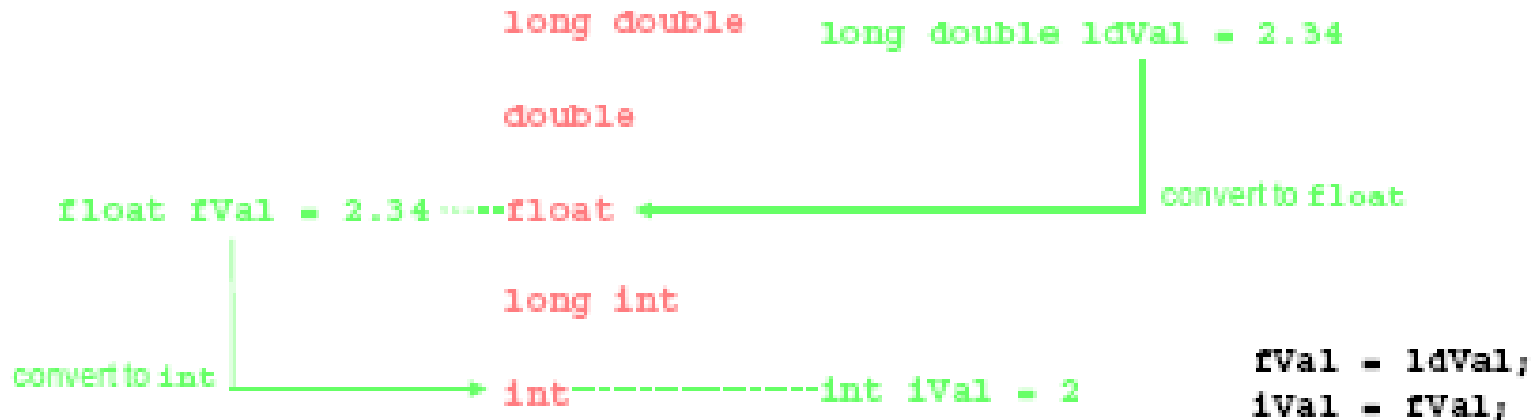
```
int conversion(int celsius);    // same identifier but different
float conversion(float celsius); // argument and return types
int main() {
    float celsius;
    float fahr = conversion(celsius); // celsius is a float so float
    :                               // version of function called
}
// functions convert celsius into fahrenheit
int conversion(int celsius) { return (int) (((9.0/5.0) * celsius) + 32); }
float conversion(float celsius) { return (((9.0/5.0) * celsius) + 32); }
```

- Although separate conversion functions have to be defined, these functions have the same identifier. This is referred to as **function overloading**.
- Templates allow the use of the same func for multiple types (see later)

Type conversion routines

Note: You don't really have to write a separate function for each input and routine type, but it's good practice!

- The compiler will use type conversion routines if there is no exact type match:



- The conversion routine will work here but do not rely on this method to convert all types.
- A variable type is not just a label to indicate the precision of a value. In general, a type is much more than a single value.

Function Overloading

- Function overloading is not just used for grouping identical functions. Overloading can be used when two functions have a similar purpose.

```
float probdist(float prob, int wins, int tries);  
float probdist(int r, float mean);  
int main() {  
    :  
    float binom = probdist(prob,wins,tries); // get binomial  
    float poisson = probdist(r,mean);        // get poisson  
    :    // the func to call depends only on input args  
}  
float probdist(float prob, int wins, int tries) {...}  
float probdist(int r, float mean) {...}
```

- This is the first example of **polymorphism!** In this case the ability to implement a function in different ways.

Polymorphism

- DEFINITION:

*“In object-oriented programming, **polymorphism** (from the Greek meaning ‘having multiple forms’) is the characteristic of being able to assign a different meaning or usage to something in different contexts - specifically, to allow an entity such as a variable, a function, or an object to have more than one form. There are several different kinds of polymorphism.”*

http://searchsmb.techtarget.com/sDefinition/0,,sid44_gci212803,00.html

The swap function

“I want to write a function that takes two integers and transposes their values.”

- Swapping the values in the function will have no effect on the variables in **main()**
- Calling a function in this way is referred to as **passing by value**.
- In fact, the object (variable) within the function is actually a **copy** not the original (just has the same label).

```
int main() {  
    int x = 2, y = 5;  
    cout << x << ", " << y << endl;  
    swap(x,y);  
    cout << x << ", " << y << endl;  
}  
  
void swap(int x, int y) {  
    //not same x and y (copies)  
    int tmp = y; y = x; x = tmp;  
    cout << x << ", " << y << endl;  
    return;  
}
```

2, 5

5, 2

2, 5

Is there an alternative method that allows the modification of the input variables?