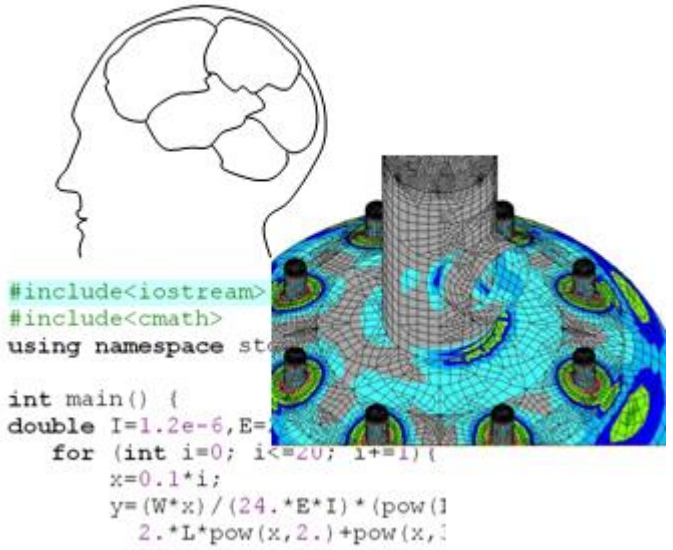
ME 110
Computation for
Mechanical
Engineering



Functions II

Part 2

Contents

This week we will look at some more details of functions:

- Arguments passed by value and by reference
- Default parameters
- Overloading functions (polymorphism)
- Putting functions in a header file

Arguments passed by value and by reference

In the previous week, we have seen that function arguments are passed by *value*.

That is when we call a function with arguments we pass values to the function and not the variables themselves.

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

void sum(double, double); // prototype

int main() {
        double x=3.4, y=6.1;
        sum(x, y);
}
                3.4          6.1

void sum(double a, double b) {
  cout << a+b << endl;
  return;
}
```

For example, in this program the values 3.4 and 6.1 are passed to the function.

The variables **x** and **y** are not passed.

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

void increment(int, int);

int main() {

   int x=4, y=7;
   cout << "Before increment(): " << x << "  " << y << endl;
                                          4          7

   increment (x, y);
   cout << "After  increment(): " << x << "  " << y << endl;
                                          4          7
}                  4        7

void increment(int a, int b) {
   a++; b++;
   cout << "Inside increment(): " << a << "  " << b << endl;
                                          5          8
}
```

Arguments passed by value.
After a call to the function, variables **x** and **y** are unchanged.
*(**values** are passed, not variables)*

Output

```
Before increment(): 4  7
Inside increment(): 5  8
After  increment(): 4  7
```

Here, **`increment()`** has no affect on variables **`x`** and **`y`** because variables **`a`** and **`b`** are local to the function and so do not reference variables **`x`** and **`y`** directly.

The aim of the function is to increment the two variables; so we need to pass the variables themselves, not only their values.

This is achieved by **passing arguments by *reference***; to do this in C++ we add the **`&`** symbol to each parameter in the function definition (and to the prototype) – this indicates that the argument is to be passed by reference instead of by value.

So `void increment(int a, int b)` Pass by value.

becomes

`void increment(int& a, int& b)` Pass by reference.

or `void increment(int &a, int &b)` Pass by reference.

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

void increment(int&, int&);

int main() {

   int x=4, y=7;
   cout << "Before increment(): " << x << "  " << y << endl;
                                          4          7
   increment (x, y);
   cout << "After  increment(): " << x << "  " << y << endl;
                                          5          8
}                  x          y

void increment(int& a, int& b) {
   a++; b++;
   cout << "Inside increment(): " << a << "  " << b << endl;
                                          5          8
}
```

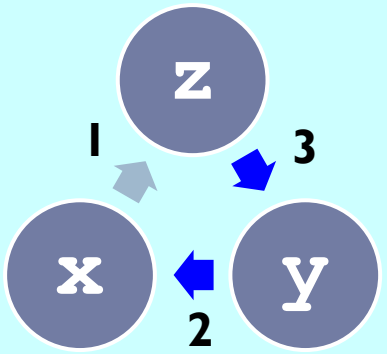
Arguments passed by reference.
After a call to the function,
variables **x** and **y** have changed.
*(**variables** are passed, not just the values)*

Output

```
Before increment(): 4  7
Inside increment(): 5  8
After  increment(): 5  8
```

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

void swap(int&, int&);

int main() {

   int x = 22, y = 33;
   cout << "Values before swapping: " << x << " " << y << endl;

   swap (x, y);
   cout << "Values  after swapping: " << x << " " << y << endl;

   return 0;
}

void swap(int& x, int& y) {

   int z = x;
   x = y;
   y = z;
   return;

}
```

In this example the **swap(x,y)** function swaps (exchanges) the values of the two variables.

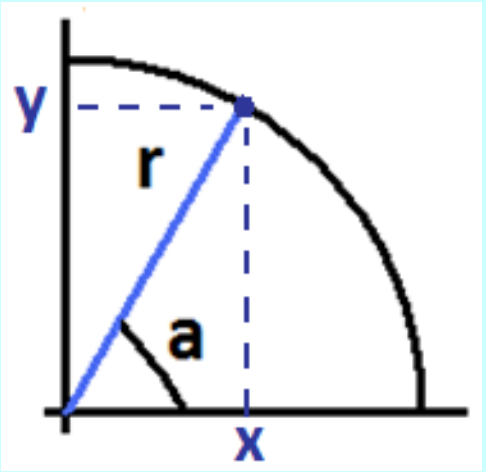


Output

```
Values before swapping: 22 33
Values after  swapping: 33 22
```

In this example, the *radius* and *angle* of a point are passed to the **`getxy()`** function that assigns referenced variables **`x`** and **`y`** the corresponding x,y coordinate values.

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

void getxy( double,  double,
            double&, double& );

int main() {
  double r, a, x, y;
  cout << "Input the radius: "; cin >> r;
  cout << "Input  the angle: "; cin >> a;
  getxy(r, a, x, y);
  cout << "x = " << x << endl;
  cout << "y = " << y << endl;
}

void getxy( double r, double a, double &x, double &y ) {
  x = r*cos(a*M_PI/180.);
  y = r*sin(a*M_PI/180.);
}
```



Output

```
Input the radius: 10
Input the angle: 60
x = 5
y = 8.66025
```

Default Parameters

A *default* value can be defined for each parameter of a function.
If a parameter is not supplied then the default value is used.

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

double cylinderVolume(double, double = 2.5);

int main() {
  double r = 1.1, h = 1.3;
  cout << cylinderVolume(r) << endl;
  cout << cylinderVolume(r,h) << endl;
  return 0;
}

double cylinderVolume(double radius, double height) {
  return 3.141592653589793 * radius * radius * height;
}
```

Output

```
9.50332
4.94173
```

If the function is defined before **`main()`** (i.e. without a prototype) then the default parameter definition looks like this:

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

double cylinderVolume(double radius, double height = 2.5) {
  return 3.14159265358979 * radius * radius * height;
}

int main() {
  double r = 1.1, h = 1.3;
  cout << cylinderVolume(r) << endl;
  cout << cylinderVolume(r,h) << endl;
  return 0;
}
```

Output

```
9.50332
4.94173
```

In this example, a third-order polynomial is implemented using a function with default parameters.

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

double poly(double, double,
            double = 0., double = 0., double = 0.);

int main() {

  double x;
  cout << "Input x: ";
  cin >> x;

  cout << "2.3 - 5.4 x + 1.4 x^2 = "
       << poly(x,2.3,-5.4,1.4) << endl;

  cout << "2.3 - 5.4 x + 1.4 x^2 + 0.8 x^3 = "
       << poly(x,2.3,-5.4,1.4,0.8) << endl;
}

double poly(double x, double a, double b, double c, double d) {
  return a + b*x + c*x*x + d*x*x*x;
}
```

Output

```
Input x: 1.5
2.3 - 5.4 x + 1.4 x^2 = -2.65
2.3 - 5.4 x + 1.4 x^2 + 0.8 x^3 = 0.05
```

Overloading functions (Polymorphism)

C++ allows the creation of *more than one function* with the *same name*.

The aim here is to allow the programmer to create functions of the same name (normally performing the same task) that accept:

- different *types of parameters* and/or
- different *numbers of parameters*.

Different *return values* are also permitted.

Many intrinsic functions are overloaded, for example **`sin(x)`** returns

- a type *float* if **`x`** is type *float*
- a type *double* if **`x`** is type *double*
- a type *long double* if **`x`** is type *long double*

This is called function overloading or polymorphism.
(poly means many, morph means form: polymorph is many-formed).

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

int max(int, int);
double max(double, double);

int main() {
  cout << max(9,7) << endl;
  cout << max(3.1,4.7) << endl;
}

int max(int x, int y) {
  if (x>y) return x;
  else     return y;
}

double max(double x, double y) {
  if (x>y) return x;
  else     return y;
}
```

Overloading with parameter types:

Example of overloading a function called **`max()`**.

The first form takes type integer arguments and returns an integer value.

The second form takes type double arguments and returns a double value.

Output

```
9
4.7
```

Overloading with numbers of parameters:

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

double oPlus(double, double);
double oPlus(double, double, double);

int main() {
  cout << oPlus(6.3,4.9) << endl;
  cout << oPlus(6.3,4.9,8.7) << endl;
}

double oPlus(double x, double y) {
  return sqrt(x*x+y*y);
}

double oPlus(double x, double y, double z) {
  return sqrt(x*x+y*y+z*z);
}
```

Output

```
7.98123
11.8064
```

Putting functions in a header file

```
#include <iostream>
#include <cmath>
#include "mylib.h"
using namespace std;

int main() {
   cout << oPlus(6.3,4.9) << endl;
   cout << oPlus(6.3,4.9,8.7) << endl;
}
```

"mylib.h" is included like others

```
double oPlus(double x, double y) {
   return sqrt(x*x+y*y);
}

double oPlus(double x, double y, double z) {
   return sqrt(x*x+y*y+z*z);
}
```

Functions are located in a new source file that is saved as "mylib.h" in the same path with main program file.