Computing and Statistical Data Analysis
Lecture 3

Type casting: `static_cast`, etc.

Basic mathematical functions

More i/o: formatting tricks

Scope, namspaces

Functions

Type casting

Often we need to interpret the value of a variable of one type as being of a different type, e.g., we may want to carry out floating-point division using variables of type `int`.

Suppose we have: `int n, m; n = 5; m = 3;` and we want to know the real-valued ratio of `n/m` (i.e. not truncated). We need to "type cast" `n` and `m` from `int` to `double` (or `float`):

```
double x = static_cast<double>(n) /
           static_cast<double>(m);
```

will give `x = 1.666666...`

Will also work here with `static_cast<double>(n)/m;` but `static_cast<double>(n/m);` gives `1.0.`

Similarly we can use `static_cast<int>(x)` to turn a `float` or `double` into an `int`, etc.

Digression #1: **`bool`** vs. **`int`**

C and earlier versions of C++ did not have the type **`bool`**. Instead, an **`int`** value of zero was interpreted as false, and any other value as true. This still works in C++:

```
int num = 1;
if ( num ) {
  ...             // condition true if num != 0
```

It is best to avoid this. If you want true or false, use **`bool`**. If you want to check whether a number is zero, then use the corresponding boolean expression:

```
if ( num != 0 ) {
  ...             // condition true if num != 0
```

Digression #2: value of an assignment and == vs. =

Recall `=` is the assignment operator, e.g., `x = 3;`

`==` is used in boolean expressions, e.g., `if ( x == 3 ) { ...`

In C++, an assignment statement has an associated value, equal to the value assigned to the left-hand side. We may see:

```
int x, y;
x = y = 0;
```

This says first assign `0` to `y`, then assign its value (`0`) to `x`. This can lead to very confusing code. Or worse:

```
if ( x = 0 ) { ... // condition always false!
```

Here what the author probably meant was

```
if ( x == 0 ) { ...
```

Standard mathematical functions

Simple mathematical functions are available through the standard C library **`cmath`** (previously **`math.h`**), including:

```
abs    acos   asin   atan   atan2  cos    cosh   exp
fabs   fmod   log    log10  pow    sin    sinh   sqrt
tan    tanh
```

Most of these can be used with **`float`** or **`double`** arguments; return value is then of same type.

Raising to a power, $z = x^y$, with **`z = pow(x,y)`** involves log and exponentiation operations; not very efficient for **`z = 2, 3`**, etc. Some advocate e.g. **`double xSquared = x*x;`**

To use these functions we need: **`#include <cmath>`**

Google for C++ cmath or see **`www.cplusplus.com`** for more info.

A simple example

Create file `testMath.cc` containing:

```
// Simple program to illustrate cmath library
#include <iostream>
#include <cmath>
using namespace std;
int main() {

  for (int i=1; i<=10; i++){
    double x = static_cast<double>(i);
    double y = sqrt(x);
    double z = pow(x, 1./3.);    // note decimal pts
    cout << x << "   " << y << "   " << z << endl;
  }

}
```

Note indentation and use of blank lines for clarity.

Running testMath

Compile and link: `g++ -o testMath testMath.cc`

Run the program: `./testMath`

```
1   1   1
2   1.41421   1.25992
3   1.73205   1.44225
4   2   1.5874
...
```

The numbers don’t line up in neat columns -- more later.

Often it is useful to save output directly to a file. Unix allows us to redirect the output:

```
./testMath > outputFile.txt
```

Similarly, use `>>` to append file, `>!` to insist on overwriting.

These tricks work with any Unix commands, e.g., `ls`, `grep`, ...

Improved i/o: formatting tricks

Often it's convenient to control the formatting of numbers.

```
cout.setf(ios::fixed);
cout.precision(4);
```

will result in 4 places always to the right of the decimal point.

```
cout.setf(ios::scientific);
```

will give scientific notation, e.g., `3.4516e+05`. To undo this, use `cout.unsetf(ios::scientific);`

`cout.width(15)` will cause next item sent to `cout` to occupy 15 spaces, e.g.,

```
cout.width(5); cout << x;
cout.width(10); cout << y;
cout.width(10); cout << z << endl;
```

To use `cout.width` need `#include <iomanip>` .

More formatting: **`printf`** and **`scanf`**

Much of this can be done more easily with the C function **`printf`**:

```
   printf ("formatting info" [, arguments]);
```

For example, for **`float`** or **`double x`** and **`int i`**:

```
   printf("%f %d \n", x, i);
```

will give a decimal notation for **`x`** and integer for **`i`**.
`\n` does (almost) same as **`endl;`**

Suppose we want 8 spaces for **`x`**, 3 to the right of the decimal point, and 10 spaces for **`i`**:

```
   printf("%8.3f %10d \n", x, i);
```

For more info google for printf examples, etc.

Also **`scanf`**, analogue of **`cin`**.

To use **`printf`** need **`#include <cstdlib>`** .

Scope basics

The scope of a variable is that region of the program in which it can be used.

If a block of code is enclosed in braces `{ }`, then this delimits the scope for variables declared inside the braces. This includes braces used for loops and if structures:

```
int x = 5;
for (int i=0; i<n; i++){
  int y = i + 3;
  x = x + y;
}
cout << "x = " << x << endl;  // OK
cout << "y = " << y << endl;  // BUG -- y out of scope
cout << "i = " << i << endl;  // BUG -- i out of scope
```

Variables declared outside any function, including **`main`**, have ‘global scope’. They can be used anywhere in the program.

More scope

The meaning of a variable can be redefined in a limited ‘local scope’:

```
int x = 5;
{
  double x = 3.7;
  cout << "x = " << x << endl;  // will print x = 3.7
}
cout << "x = " << x << endl;    // will print x = 5
```

(This is bad style; example is only to illustrate local scope.)

In general try to keep the scope of variables as local as possible. This minimizes the chance of clashes with other variables to which you might try to assign the same name.

Namespaces

A namespace is a unique set of names (identifiers of variables, functions, objects) and defines the context in which they are used.

E.g., variables declared outside of any function are in the global namespace (they have global scope); and can be used anywhere.

A namespace can be defined with the **`namespace`** keyword:

```
namespace aNameSpace {
  double x = 1.0;
}
```

To refer to this **`x`** in some other part of the program (outside of its local namespace), we can use

```
aNameSpace::x
```

`::` is the scope resolution operator.

The **std** namespace

C++ provides automatically a namespace called `std`.

It contains all identifiers used in the standard C++ library (lots!), including, e.g., `cin`, `cout`, `endl`, ...

To use, e.g., `cout`, `endl`, we can say:

```
using std::cout;
using std::endl;
int main(){
  cout << "Hello" << endl;
  ...
```

or we can omit `using` and say

```
int main(){
  std::cout << "Hello" << std::endl;
  ...
```

`using namespace std;`

Or we can simply say

```
using namespace std;
int main(){
  cout << "Hello" << endl;
  ...
```

Although I do this in the lecture notes to keep them compact, it is not a good idea in real code. The namespace **`std`** contains thousands of identifiers and you run the risk of a name clash.

This construction can also be used with user-defined namespaces:

```
using namespace aNameSpace;
int main(){
  cout << x << endl;          // uses aNameSpace::x
  ...
```

Functions

Up to now we have seen the function **`main`**, as well as mathematical functions such as **`sqrt`** and **`cos`**.   We can also define other functions, e.g.,

```
const double PI = 3.14159265;      // global constant
double ellipseArea(double, double);  // prototype
int main() {
  double a = 5;
  double b = 7;
  double area = ellipseArea(a, b);
  cout << "area = " << area << endl;
  return 0;
}

double ellipseArea(double a, double b){
  return PI*a*b;
}
```

The usefulness of functions

Now we can ‘call’ **`ellipseArea`** whenever we need the area of an ellipse; this is modular programming.

The user doesn’t need to know about the internal workings of the function, only that it returns the right result.

‘Procedural abstraction’ means that the implementation details of a function are hidden in its definition, and needn’t concern the user of the function.

A well written function can be re-used in other parts of the program and in other programs.

Functions allow large programs to be developed by teams (as is true for classes, which we will see soon).

Declaring functions

Before we can use a function, we need to declare it at the top of the file (before `int main()`).

```
   double ellipseArea(double, double);
```

This is called the 'prototype' of the function. It begins with the function's 'return type'. The function can be used in an expression like a variable of this type.

The prototype must also specify the types of the arguments, in the correct sequence. Variable names are optional in the prototype.

The specification of the types and order of the arguments is called the function's signature.

Defining functions

The function must then be defined, i.e., we must say what it does with its arguments and what it returns.

```
double ellipseArea(double a, double b){
  return PI*a*b;
}
```

The first word defines the type of value returned, here **`double`**.

Then comes a list of parameters, each preceded by its type.

Note the scope of **`a`** and **`b`** is local to the function **`ellipseArea`**. We could have given them names different from the **`a`** and **`b`** in the main program (and we often do).

Then the body of the function does the necessary computation and finally we have the **`return`** statement followed by the corresponding value of the function.

Return type of a function

The prototype must also indicate the return type of the function, e.g., **`int, float, double, char, bool`**.

```
double ellipseArea(double, double);
```

The function's return statement must return a value of this type.

```
double ellipseArea(double a, double b){
  return PI*a*b;
}
```

When calling the function, it must be used in the same manner as an expression of the corresponding return type, e.g.,

```
double volume = ellipseArea(a, b) * height;
```

Return type **`void`**

The return type may be ‘void’, in which case there is no return statement in the function (like a FORTRAN subroutine):

```
void showProduct(double a, double b){
  cout << "a*b = " << a*b << endl;
}
```

To call a function with return type void, we simply write its name with any arguments followed by a semicolon:

```
showProduct(3, 7);
```

Putting functions in separate files

Often we put functions in a separate files. The declaration of a function goes in a 'header file' called, e.g., **ellipseArea.h**, which contains the prototype:

```
#ifndef ELLIPSE_AREA_H
#define ELLIPSE_AREA_H

// function to compute area of an ellipse

double ellipseArea(double, double);

#endif
```

The directives **#ifndef** (if not defined), etc., serve to ensure that the prototype is not included multiple times. If **ELLIPSE_AREA_H** is already defined, the declaration is skipped.

Putting functions in separate files, continued

Then the header file is included (note use of `" "` rather than `< >`) in all files where the function is called:

```
#include <iostream>
#include "ellipseArea.h"
using namespace std;
int main() {
  double a = 5;
  double b = 7;
  double area = ellipseArea(a, b);
  cout << "area = " << area << endl;
  return 0;
}
```

(`ellipseArea.h` does not have to be included in the file `ellipseArea.cc` where the function is defined.)

Wrapping up lecture 3

We've now seen all of the important control structures and enough i/o to do some useful work.

We know how to reinterpret e.g. a **`double`** as an **`int`** (type casting) and we've seen the standard C library of mathematical functions (**`cmath`**).

We've started off with declaring and defining our own functions, and seen how to put these in separate files.

Next: how arguments are passed to functions, etc.