Computing and Statistical Data Analysis
Lecture 2

Variables, types: `int, float, double, bool,` ...

Assignments, expressions

Simple i/o; `cin` and `cout`.

Basic control structures: `if, else`

Loops: `while, do-while, for, ...`

C++ building blocks

All of the words in a C++ program are either:

Reserved words: cannot be changed, e.g.,

`if, else, int, double, for, while, class`, ...

Library identifiers: default meanings usually not changed, e.g., **`cout, sqrt`** (square root), ...

Programmer-supplied identifiers:

e.g. variables created by the programmer,

`x, y, probeTemperature, photonEnergy`, ...

Valid identifier must begin with a letter or underscore ("_") , and can consist of letters, digits, and underscores.

Try to use meaningful variable names; suggest lowerCamelCase.

Data types

Data values can be stored in variables of several types.

Think of the variable as a small blackboard, and we have different types of blackboards for integers, reals, etc.
The variable name is a label for the blackboard.

Basic integer type: **`int`** (also **`short`**, **`unsigned`**, **`long int`**, ...)

Number of bits used depends on compiler; typically 32 bits.

Basic floating point types (i.e., for real numbers):

`float` usually 32 bits

`double` usually 64 bits ← best for our purposes

Boolean: **`bool`** (equal to **`true`** or **`false`**)

Character: **`char`** (single ASCII character only, can be blank),
no native 'string' type; more on C++ strings later.

Declaring variables

All variables must be declared before use.

Usually declare just before 1st use.

Examples

```
int main(){
  int numPhotons;         // Use int to count things
  double photonEnergy;    // Use double for reals
  bool goodEvent;         // Use bool for true or false
  int minNum, maxNum;     // More than one on line
  int n = 17;             // Can initialize value
  double x = 37.2;        // when variable declared.
  char yesOrNo = 'y';     // Value of char in ' '
      ...
}
```

Assignment of values to variables

Declaring a variable establishes its name; value is undefined (unless done together with declaration).

Value is assigned using = (the assignment operator):

```
int main(){
  bool aOK = true;  // true, false predefined constants
  double x, y, z;
  x = 3.7;
  y = 5.2;
  z = x + y;
  cout << "z = " << z << endl;
  z = z + 2.8;      // N.B. not like usual equation
  cout << "now z = " << z << endl;
       ...
}
```

Constants

Sometimes we want to ensure the value of a variable doesn't change.

Useful to keep parameters of a problem in an easy to find place, where they are easy to modify.

Use keyword **`const`** in declaration:

```
const int numChannels = 12;
const double PI = 3.14159265;

// Attempted redefinition by Indiana State Legislature
PI = 3.2;              // ERROR will not compile
```

Old C style retained for compatibility (avoid this):

```
#define PI 3.14159265
```

Enumerations

Sometimes we want to assign numerical values to words, e.g.,

January = 1, February = 2, etc.

Use an 'enumeration' with keyword **enum**

```
enum { RED, GREEN, BLUE };
```

is shorthand for

```
const int RED = 0;
const int GREEN = 1;
const int BLUE = 2;
```

Enumeration starts by default with zero; can override:

```
enum { RED = 1, GREEN = 3, BLUE = 7 }
```

(If not assigned explicitly, value is one greater than previous.)

Expressions

C++ has obvious(?) notation for mathematical expressions:

operation	symbol
addition	`+`
subtraction	`-`
multiplication	`*`
division	`/`
modulus	`%`

Note division of `int` values is truncated:

```
int n, m;  n = 5;  m = 3;
int ratio = n/m;       // ratio has value of 1
```

Modulus gives remainder of integer division:

```
int nModM = n%m;           // nModM has value 2
```

Operator precedence

`*` and `/` have precedence over `+` and `-`, i.e.,

```
x*y + u/v  means  (x*y) + (u/v)
```

`*` and `/` have same precedence, carry out left to right:

```
x/y/u*v  means  ((x/y) / u) * v
```

Similar for `+` and `-`

```
x - y + z  means  (x - y) + z
```

Many more rules (google for C++ operator precedence).

Easy to forget the details, so use parentheses unless it's obvious.

Boolean expressions and operators

Boolean expressions are either true or false, e.g.,

```
int n, m; n = 5; m = 3;
bool b = n < m;                  // value of b is false
```

C++ notation for boolean expressions:

```
greater than                     >
greater than or equals           >=
less than                        <
less than or equals              <=
equals                           ==
not equals                       !=
```

not =

Can be combined with `&&` (“and”), `||` (“or”) and `!` (“not”), e.g.,

```
(n < m) && (n != 0)              (false)
(n%m >= 5) || !(n == m)          (true)
```

Precedence of operations not obvious; if in doubt use parentheses.

Shorthand assignment statements

full statement	shorthand equivalent
`n = n + m`	`n += m`
`n = n - m`	`n -= m`
`n = n * m`	`n *= m`
`n = n / m`	`n /= m`
`n = n % m`	`n %= m`

Special case of increment or decrement by one:

full statement	shorthand equivalent
`n = n + 1`	`n++` (or `++n`)
`n = n - 1`	`n--` (or `--n`)

`++` or `--` before variable means first increment (or decrement), then carry out other operations in the statement (more later).

Getting input from the keyboard

Sometimes we want to type in a value from the keyboard and assign this value to a variable. For this use the iostream object **`cin`**:

```
int age;
cout << "Enter your age" << endl;
cin >> age;
cout << "Your age is " << age << endl;
```

When you run the program you see

```
Enter your age
23                    ← you type this, then "Enter"
Your age is 23
```

(Why is there no "**`jin`**" in java? What were they thinking???)

if and else

Simple flow control is done with **`if`** and **`else`**:

```
if ( boolean test expression ){
  Statements executed if test expression true
}
```

or

```
if (expression1 ){
  Statements executed if expression1 true
}
else if ( expression2 ) {
  Statements executed if expression1 false
  and expression2 true
}
else {
  Statements executed if both expression1 and
  expression2 false
}
```

more on if and else

Note indentation and placement of curly braces:

```
if ( x > y ){
  x = 0.5*x;
}
```

Some people prefer

```
if ( x > y )
{
  x = 0.5*x;
}
```

If only a single statement is to be executed, you can omit the curly braces -- this is usually a bad idea:

```
if  ( x > y )  x = 0.5*x;
```

Putting it together -- `checkArea.cc`

```
#include <iostream>
using namespace std;
int main() {
  const double maxArea = 20.0;
  double width, height;
  cout << "Enter width" << endl;
  cin >> width;
  cout << "Enter height" << endl;
  cin >> height;
  double area = width*height;
  if ( area > maxArea ){
    cout << "Area too large" << endl;
  else {
    cout << "Dimensions are OK" << endl;
  }
  return 0;
}
```

"while" loops

A **while** loop allows a set of statements to be repeated as long as a particular condition is true:

```
while( boolean expression ){
  // statements to be executed as long as
  // boolean expression is true


}
```

For this to be useful, the boolean expression must be updated upon each pass through the loop:

```
while (x < xMax){
  x += y;
  ...
}
```

Possible that statements never executed, or that loop is infinite.

“do-while” loops

A **do-while** loop is similar to a **while** loop, but always executes at least once, then continues as long as the specified condition is true.

```
do {
  // statements to be executed first time
  // through loop and then as long as
  // boolean expression is true

} while ( boolean expression )
```

Can be useful if first pass needed to initialize the boolean expression.

"for" loops

A `for` loop allows a set of statements to be repeated a fixed number of times. The general form is:

```
for ( initialization action ;
      boolean expression ; update action ){
  // statements to be executed

}
```

Often this will take on the form:

```
for (int i=0; i<n; i++){
  // statements to be executed n times

}
```

Note that here `i` is defined only inside the `{ }`.

Examples of loops

A for loop:

```
int sum = 0;
for (int i = 1; i<=n; i++){
  sum += i;
}
cout << "sum of integers from 1 to " << n <<
     " is " << sum << endl;
```

A do-while loop:

```
int n;
bool gotValidInput = false;
do {
  cout << "Enter a positive integer" << endl;
  cin >> n;
  gotValidInput = n > 0;
} while ( !gotValidInput );
```

Nested loops

Loops (as well as if-else structures, etc.) can be nested, i.e., you can put one inside another:

```
// loop over pixels in an image

for (int row=1; row<=nRows; row++){
  for (int column=1; column<=nColumns; column++){
    int b = imageBrightness(row, column);
    ...

  }      // loop over columns ends here
}        // loop over rows ends here
```

We can put any kind of loop into any other kind, e.g., **`while`** loops inside **`for`** loops, vice versa, etc.

More control of loops

`continue` causes a single iteration of loop to be skipped (jumps back to start of loop).

`break` causes exit from entire loop (only innermost one if inside nested loops).

```
while ( processEvent ) {

  if ( eventSize > maxSize ) { continue; }

  if ( numEventsDone > maxEventsDone ) {
    break;
  }

//  rest of statements in loop ...

}
```

Usually best to avoid **`continue`** or **`break`** by use of **`if`** statements.

Wrapping up lecture 2

We've seen some basic elements of a C++ program:

variables, e.g., **`int`**, **`double`**, **`bool`**, etc.;

how to assign values and form expressions;

how to get values from the keyboard and write values to the monitor;

how to control the flow of a program with **`if`** and **`else`**;

how to control flow with loops (**`while`**, **`do-while`**, **`for`**, etc.).

Next we will look at some library functions, and then move on to user defined functions.