Part II

Basic Mathematics

Outline

Basic mathematics

The basic operators +, -, *, / exist and act more-or-less as you would expect.

```
int a = 5;
int b = 3;
int c = a*b + 5;
std::cout << "c = " << c << std::endl;
c = c * 2;
std::cout << "c = " << c << std::endl;
```

Modulus operator: `7 % 3 == 1`.
This has the same precedence as * and /.
These operators may not quite act as you expect if the numbers involved overflow their types. Of which, more later...

Literals

- Decimal integers are specified as:
  `10, 42, 0, 12, 1e3`
- Double-precision floating-point numbers are specified as:
  `1.23e4, 1e-3`
- Binary integers are specified as:
  `0b101010`
- Hexadecimal integers are specified as:
  `0xa, 0x100, 0xffff`
- Octal integers are specified with a leading zero:
  `010, 077, 0123`
- Single characters are specified as:
  `‘a‘, ‘b‘`
- “Strings” of characters are specified as:
  `"Hello"`

Short-hand operators

There are short-hand versions of some operate-and-assign operations:
+= -= *= /= %=
Both the following lines do the same thing (on basic arithmetic types)

```
a = a + 5;
a += 5;
```

Also the increment and decrement operators **++ --**:
Both the following lines do the same thing (on basic arithmetic types).

```
i++; j--;
i = i+1; j = j-1;
```

(Hence the name C++)

Pre- and post-increment

- There is a difference between `i++` and `++i`:
- The pre-increment `++i` increments `i` and evaluates to the result after the increment.
- The post-increment `i++` evaluates `i` and then increments it

```
int i=2;
int j=(i++); // j=2, i=3
int k=(++i); // k=4, i=4
```

- You are advised not to write code that relies subtly on the distinction between the two.

Bitwise operators

- C++ also has operators that act bitwise
- Left-shift `<<` and right-shift `>>`
- These shift the binary representation of an integer to the left or right:

```
int a = 43; // 101011 in binary
int b = a << 2; // b = 172 (or 10101100 in binary)
int c = a >> 2; // c = 10 (or 1010 in binary)
```

- Bitwise NOT: `int a = ~5;`
- Now `a = -6` or `111...1010`
- Bitwise OR: `int a = 11 | 12; // a = 15`
- Bitwise AND: `int a = 11 & 12; // a = 8`

Outline

Basic types

Definition

The kind of information stored in a variable is referred to as its “type”.

- Examples of fundamental types in C++ are: `int`, `float`, `double`, `bool`, `unsigned int`
- C++ performs static type-checking: the types used must be known at compile-time so that the function to be called can be determined.
- This allows for higher-performance than other dynamically-typed languages that check types at run-time.
- If a function cannot be called with precisely the given types, then either types may be converted (using standard conversions such as `int` $\mapsto$ `float`) or the compilation may fail.

Basic types ctd.

- Integral types:
  - `bool`
  - `char`, `int`, `short int`, `long int`
    (can be preceded by signed (the default) or unsigned)
- Floating point types:
  - `float` single precision - usually 32-bit
  - `double` double precision - usually 64-bit
  - `long double` extended double precision
- `void` - absence of information

Type ranges

Typical for current 64-bit computers - not mandated by the C++ standard

Type	Bits	Range
`signed char`	8	-128 to 127
`unsigned char`	8	0 to 255
`signed short int`	16	-32768 to 32767
`unsigned short int`	16	0 to 65535
`signed int`	32	$-(2^{31})$ to $2^{31}-1$
`unsigned int`	32	0 to $2^{32}-1$
`signed long int`	64	$-(2^{63})$ to $2^{63}-1$
`unsigned long int`	64	0 to $2^{64}-1$
`float`	32	$\pm 3.4 \times 10^{38}$ (~ 7 s.f.)
`double`	64	$\pm 1.7 \times 10^{308}$ (~ 15 s.f.)

Programmatically discovering numeric type properties

- You may wish to write a portable program that can check the properties of the numeric types at run-time.
- For all built-in integral types, such as `int`, the following are available:

```
#include <limits>
int std::numeric_limits<int>::digits; // Binary digits
bool std::numeric_limits<int>::is_signed;
bool std::numeric_limits<int>::is_integer;
int std::numeric_limits<int>::min();
int std::numeric_limits<int>::max();
```

- For floating-point types, the following are available:

```
int std::numeric_limits<float>::digits; // Binary digits in
    the mantissa
float std::numeric_limits<float>::min();
float std::numeric_limits<float>::max();
float std::numeric_limits<float>::epsilon();
```

where ϵ is the smallest s.t. $1 + \epsilon > 1$.

- See a reference manual for all available values.

Variable declaration

- It is possible to declare multiple variables on a single line:

```
int a,b,c=5;
double x=0.9, y=1.1, z;
```

- will declare integers `a, b, c`,
- initializing `c` to be 5,
- and double-precision variables `x,y,z`,
- initializing `x` and `y`, but not `z`.

Integer division in C++

- A common problem found when learning C++ is that `1/3 == 0`
- Integer division always yields an integer
- `(-4) / 3` is *probably* equal to `-1` (i.e. round-towards-zero)
- but could be `-2` depending on the particular implementation (compiler)
- Note that modulo arithmetic is defined such that for integers

  ```
  (a/b)*b + a%b == a
  ```

  so modular arithmetic is always consistent with division.
- If you need moduli of negative numbers, you should check what `(-4) % (-3)` gives.

Integer overflow/wrapping in C++

- Integers are represented using a fixed number of bits
- When the result (or intermediate result) of an operation cannot be represented in this, then what happens?
- `unsigned int` will wrap around, i.e.

```
unsigned int i = 4294967295; // 2^32 - 1
unsigned int j = i + 1; // Now j == 0
```

- Overflow on signed arithmetic is not defined

```
int i = 2147483647; // 2^31 - 1;
int j = i + 1;
```

- The value of `j` is not defined by the C++ standard
- It may be -2^{31}, but it need not be.
- Your program could even crash at this point from integer overflow. (Misleadingly unlikely.)

Avoiding integer overflow

- Do not assume that all compilers/systems do the same as the one you're currently using
- Further, do not assume that your compiler will give the same answer in all circumstances - may be affected by optimization
- To avoid issues, you would need:

```
signed int a, b, c = 0;
// Initialize a and b
if( b == 0 || (b != 0 &&
abs(std::numeric_limits<signed int>::max() / b) < abs(a)) )
{
  c = a * b;
}
```

- However, this is probably overkill and too expensive for everything except software needing extensive robustness or security checks.
- It's worth bearing in mind, though.

Floating-point arithmetic

- Need to store finite-precision approximations to real numbers
- Floating-point arithmetic is not exact
- Floating-point arithmetic is commutative `x * y == y * x`
- Floating-point arithmetic is not associative:
  `(x+y)+z` not necessarily the same as `x+(y+z)`.
- You should (almost) never test for exact equality of floating-point numbers: 3*(1.0/3.0) may not be equal to 1.
- Floating-point exceptions occur upon division by zero, square-roots of -ve numbers, and similar
- These can be caught and may help in tracking down bugs, but may not be reliable.

Floating-point arithmetic ctd

- `NaN` stands for Not A Number.
- This can arise from `0/0`, ∞/∞ and similar computations.
- For full details, see "What every computer-scientist should know about floating-point numbers" (available online)

Type conversion

- When assigning an integer to a floating-point value or performing an operation on an integer and a floating-point number:

  ```
  double a = 2 + 5.3;
  ```

  then the 2 is converted to double precision before the addition is performed.
- When performing a division of two integers, any of the following will work:

  ```
  double oneThird = 1.0/3.0;
  double oneThird = 1/(double)3;
  double oneThird = (double)1/3;
  ```

- Note that specifying a floating-point number gives double precision by default.
- To specify single-precision, use a suffix `f`: `3.2f`.

Exponential notation

- Larger/smaller numbers can be specified by using exponential notation:
  `float b = 1.3e10f; float c = -9e-13f;`
- Note that specifying a number out of range leads to undefined behaviour:
  `double s = 1.3e400;` // Contents of s at run-time are undefined
- In theory, your program could crash here. Again, misleadingly unlikely.

Truncation

- Assigning a `float` to `int` truncates the value:

```
float a = 5.2f;
int b = a; // Now b == 5
float c = -3.141f;
int d = c; // Now d == -3
```

- Assigning a value larger than that which can be contained by the destination type gives undefined results.

```
float e = 1e10; // Bigger than 2^32
int f = e; // result undefined
```

Mathematical functions

Include these using `#include <cmath>`

- Various standard functions exist, in both single-precision and double-precision forms.
- `sin`, `cos`, `tan`, `sqrt`, `log`, `exp`, `asin`, `acos`
- `fabs` - floating-point absolute-value
- `abs` - integer absolute-value
- `ceil`, `floor` - round up/down
- `atan2(y,x)` $= \tan^{-1}(y/x)$ and deals with $x, y = 0$ appropriately

pow

- `pow(x,y)` $= x^y$
- `pow` has four forms:
  - float pow(float, float)
  - double pow(double, double)
  - long double pow(long double, long double)
  - ResultType pow(Arithmetic1 base, Arithmetic2 exp)
- Note that integral powers are not covered in this.
- `pow(2,3)` will convert `2` to double-precision floating-point and then calculate `2.0*2.0*2.0`
- The third option mentions `long double` - extended precision. May or may not give better accuracy.
- The fourth option allows all combinations of arithmetic types not covered by the first three. The return type is always `double` or `long double`.

Outline

Branching

We often want to change what our code does depending on input.
We can use `if` statements which evaluate a set of statements only if a given condition holds:

```
int a;
std::cin >> a;
if( a == 0 ){
    std::cout << "a is equal to 0";
}
```

Note the equality test operator `==` which is different from assignment operator `=`.

Else

The extended construct of **if** is:

```
if( condition )
    execute-if-true;
else
    execute-if-false;
```

```
if( a < 0 ){
    std::cout << "a is negative" << std::endl;
}
else{
    std::cout << "a is non-negative" << std::endl;
}
```

Else-if chains

```
if( a < 0 ){
    std::cout << "a is negative" << std::endl;
}
else   if(a > 0){
    std::cout << "a is positive" << std::endl;
}
else{
    std::cout << "a is zero" << std::endl;
}
```

The final `else` is associated with the immediately preceeding `if`. This highlights the importance of grouping statements with braces.

Else-if chains

```
if( a < 0 ){
    std::cout << "a is negative" << std::endl;
}
else { if(a > 0){
    std::cout << "a is positive" << std::endl;
}
else{
    std::cout << "a is zero" << std::endl;
}
}
```

The final **else** is associated with the immediately preceeding **if**. This highlights the importance of grouping statements with braces.

Relational operators

The following operators compare two values and result in a boolean:

- `< >` Less/Greater than
- `<= >=` Less/Greater than or equal to
- `==` Equal
- `!=` Not equal

```
if( a != 1 ){
    std::cout << "a is not equal to 1" << std::endl;
}
```

The result of one of these can also be assigned to a boolean variable:

```
bool aIsPositive = (a > 0);
if( aIsPositive ){
    std::cout << "Variable is +ve" << std::endl;
}
```

Warning

- The equality test operator `==` is not the same as the assignment operator `=`
- So, `a=3` *always* sets a to be equal to 3 and the expression returns the new value of `a`.

```
if( a = 3 ){
    std::cout << "a is 3" << std::endl;
}
```

  will set `a=3` and always print the given statement (because 3 converts to `true`).
- Here, `a==3` should be used instead.
- If you convert a boolean value to an integer, `true` is 1 and `false` is 0.

Boolean/integer equivalence

- If you allocate a non-boolean number to a boolean, some conversion must occur.
- Any non-zero number converts to `true`
- Zero converts to `false`

So, in the following:

```
if( 0 ){
    std::cout << "Never get here!" << std::endl;
}
if( -1 ){
    std::cout << "Always get here!" << std::endl;
}
```

only the second message is printed.
Although this has valid uses, you should not usually use this style.

Outline

Logical operations

You can combine the results of comparisons as follows:

- `&&` Logical AND
- `||` Logical OR

so that

```
if( a==0 || b==0 ){
    std::cout << "At least one of a and b is zero." <<
   std::endl;
}
```

works as you would expect

Note that `&&` has higher precedence than `||`, so

```
if( a == 0 && b == 0 || c == 2 ){
  // More code here.
}
```

evaluates either if `c==2` OR both `a` and `b` are zero.

Parentheses are usually a good idea here.

Logical NOT

- There is also the NOT operator `!` which has higher precedence than `&&` and `||`

  ```
  bool a = false;
  if( !a ){
    std::cout << "a is false" << std::endl;
  }
  ```

  and therefore

  ```
  if( !( a && b) )
  ```

  is the same as

  ```
  if( !a || !b )
  ```

  (so long as evaluating `a` and `b` has no side-effects)

What not to do

The following is valid C++, but will not do what you want:

```
if( 2 <= a <= 5 ){
  std::cout << "a is between 2 and 5 (inclusive)" << std::endl;
}
```

will not evaluate precisely when `a` is between 2 and 5.
The compiler will see:

```
if( (2 <= a) <=5 )
```

Whatever `a` is, the result of `(2 <= a)` is either `false` or `true`, which convert to 0 or 1, so this always evaluates to `true`.

Short-cut evaluation

When computing the result of a logical expression, only as many tests as are required to determine the result are carried out, working from left to right (taking parentheses and operator precedence into account)

```
int a = 0;
if( a == 0 || b < 0 ){
//... Code here ...
}
if( a == 1 && b > 0 ){
//... Code here ...
}
```

In both cases, the second test is not performed.
This is called short-cut evaluation.
It is of more use in the following:

```
if( v.size() >= 5 && v[4] == 5 )
```

where the second test could cause a seg-fault if v were not big enough.

Ternary Operator

- As a shortcut to **`if-else`**, there is the ternary operator **`?:`**

```
n = ( (n % 2 == 1) ? 3*n+1 : n/2 );
```

which is equivalent to

```
if( n % 2 == 1 ){
  n = 3*n + 1;
}
else{
  n = n/2;
}
```

- The ternary operator should only be used to replace very simple **`if-else`** statements.
- and should usually be contained in parentheses due to its low precedence (only just above =)