C++ Programming for Engineers

Petar Jervic
Francisco Luro
Andrew Moss (course-resp)

May 2, 2012

Representations and Transformations

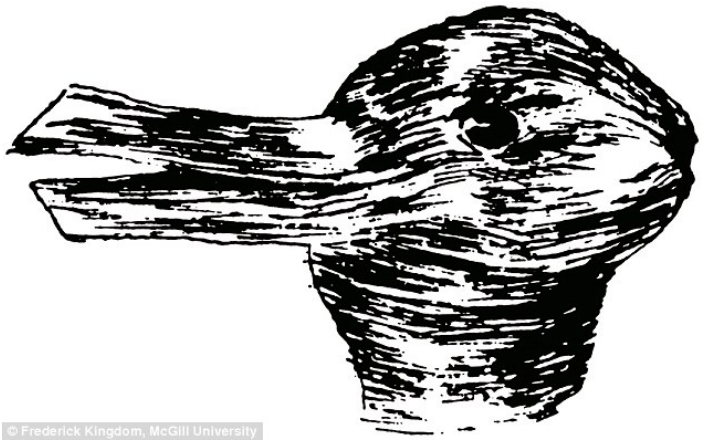
Last week covered basic variable types and expressions

- All of these topics rely implicitly on values and representations
- First hour this week we look in more depth at
  - Values
  - Representations
  - Transforming between them.

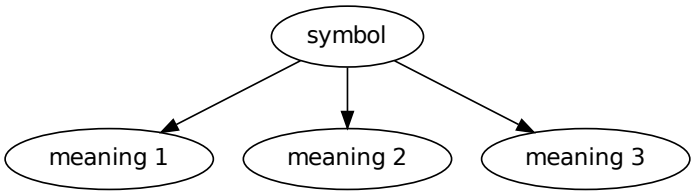
After the break

- Using simple functions

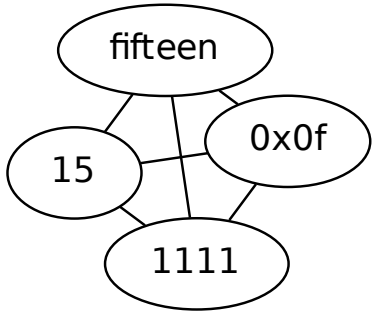
Symbols as representations



- Duck or Rabbit?
- Young or Old Lady?
- Can one image (pixels) be two pictures?
- When symbols represent something else
  - Meaning is dependent on context
  - Meaning is dependent on interpretation



Integers



The integers are normally written as 0,1,2,3,...

- It is easy to forget that these are symbols
- Because they are representations of numbers they are not unique
- We can write many different symbols for the same number

Positional number systems (binary, decimal, hexadecimal...) are defined by a *base*

- The base is the number of digit symbols
- Binary is base 2, hexadecimal is base 16...
- Each digit position is multiplied by a power of the base
  - e.g. In decimal $1234 = 4 \cdot 1 + 3 \cdot 10 + 2 \cdot 100 + 1 \cdot 1000$
  - e.g. In hexadecimal $1234 = 4 \cdot 1 + 3 \cdot 16 + 2 \cdot 256 + 1 \cdot 4096$

Bits

Simplest form of storage (memory) is a bit.

- Two possible values: 0/1, or on/off, or charge/no-charge . . .

If we arrange bits together then we can store binary representations

- e.g. | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |
- This could represent the number 55.
- It could also represent the ASCII character that is the symbol `7`.
- It could also be a bit pattern that we manipulate directly with logic.

Binary is used as a universal representation (“digital data”)

- Everything that your programs manipulate is a collection of bits.
- What those bits *mean* depends on how you use them.

Logic operations on numbers

Last week we saw logical operators on bits / binary representations

- But each of those sequences of bits represents a number
- So each of the basic logical operators also affects numbers

Let's look at masking (AND operator)

- Where we have a 1 in the mask we pass the input bit through.
- When we have a 0 in the mask we drop the bit and use 0.

x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0		x
0	0	1	1	0	1	1	1		55
0	0	x_5	x_4	0	x_2	x_1	x_0		$32x_5 + 16x_4 + 4x_2 + 2x_1 + x_0$

- Seems like a strange expression in the general case
  - Specific cases are more useful...

Masking high bits of numbers

1	1	1	0	0	0	0	1		225 (`0xE1`)
1	1	1	1	1	1	1	0		254 (`0xFE`)
1	1	1	0	0	0	0	0		224 (`0xE0`)

- Used a mask that keeps the top seven bits, removes the lowest bit
- The number changes from 225 to 224...
- Let's try more examples
  - $5 \rightarrow 4, 6 \rightarrow 6, 7 \rightarrow 6, 8 \rightarrow 8 \ldots$
  - Familiar?
- So what does `x&0xFC` do?

Masking low bits of numbers

1	1	1	0	0	0	0	1		225 (`0xE1`)
0	0	0	0	1	1	1	1		15 (`0x0F`)
0	0	0	0	0	0	0	1		1 (`0x01`)

- Used a mask that keeps the low four bits, removes the high four
- The number changes from 225 to 1 . . .
- Let's try more examples
  - $16 \rightarrow 0, 17 \rightarrow 1, 32 \rightarrow 0, 33 \rightarrow 1 \ldots$
  - Familiar?
- So what does `x&0x03` do?
- Which mask extracts whether the number is odd or even?

Modulo arithmetic (recap)

Two ways to view division

- $\frac{x}{y} = z$, or $\frac{x}{y} = q$ remainder r, where $z - q = \frac{r}{y}$
- In C++ we have two operators: `q = x/y; r = x%y;`

The floor of a number is the closest integer beneath it

- This means throwing away the fractional part
- e.g. $\lfloor \frac{10}{3} \rfloor = 3$
- Integer division `x/y` is the result $\lfloor \frac{x}{y} \rfloor$ (rounds down)

The ceiling of a number is the closest integer above it

- Still throw away the fraction, but rounding up
- e.g. $\lceil \frac{12}{5} \rceil = 3$

When x is an integer value: $\lfloor x \rfloor = \lceil x \rceil = x$

- Otherwise $\lceil x \rceil = \lfloor x \rfloor + 1$

Walkthrough of Exercise 9

Problem Definition

Input an integer and round it up to the nearest odd number.

- The description is deliberately concise
- It forces you to find ways to break it apart
- To solve the problem you need to simplify it
- Look for clues to the simplifying step
- Odd / Even — there are only two cases
  - When there are few cases try examples to see what happens

Walkthrough (cont)

Inputs		Outputs
5		5
6		7
7	$\rightarrow$	7
8		9
9		9

- Pick some numbers
- Round them up to nearest odd
- Look for patterns
  - Odd numbers remain the same
  - Even numbers are incremented
  - Output changes in steps of 2
  - Linear steps in output
- Guess simple intermediate steps
  - Linear pattern $\rightarrow$ try addition
  - What is output - 1 ?
  - What has changed / remained the same?

Inputs		Outputs - 1
5		4
6		6
7	$\rightarrow$	6
8		8
9		8

Walkthrough (cont)

Steps of two with even numbers looks similar to integer division...

- Start to guess what the expression looks like
- $\lfloor \frac{x}{2} \rfloor \cdot 2 + 1$, which is `x/2*2 + 1`
- One solution is just `cin » x; cout « x/2+1;`

Let's try another approach

- Odd numbers remain the same, even numbers are incremented.
- All odd numbers have a final bit set to 1, all even numbers have it set to 0
- If we add 1 to an even number it sets the final bit (no carry possible)
- So all we need to do is set the final bit
- Another solution is just `cin » x; cout « (x|1);`

Shifting bitstrings vs arithmetic

Last week we saw

x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0		`x`
x_6	x_5	x_4	x_3	x_2	x_1	x_0	0		`x << 1`
x_5	x_4	x_3	x_2	x_1	x_0	0	0		`x << 2`

So what this do to the number stored in `x`?

- In decimal if we add zero to the end of a number 1234 we get 12340, 123400...
- Each time we multiply by ten (the base)
- So what happens in binary if we add zeros and shift the digits up?

Some weird expressions

When looking at expressions the evaluation order is important

- Consider `x++ + --x`
- What is the result of the expression?

What is the value of `x-- - ++x` ?
Lastly, a statement that catches people out

- Mathematically we cannot say $x = x + 1$, it is inconsistent
- But in C++ we can write `x = x+1;`
- What does it do?

Break

Functions

Overview

- Extending our toolbox
- Calling a function
- Operators vs Functions
- Calling a function
  - Inputs and output datatypes
  - Type conversion (coercion)
- Standard library functions
- Stateless vs stateful functions
  - Stateless functions
  - Stateful functions

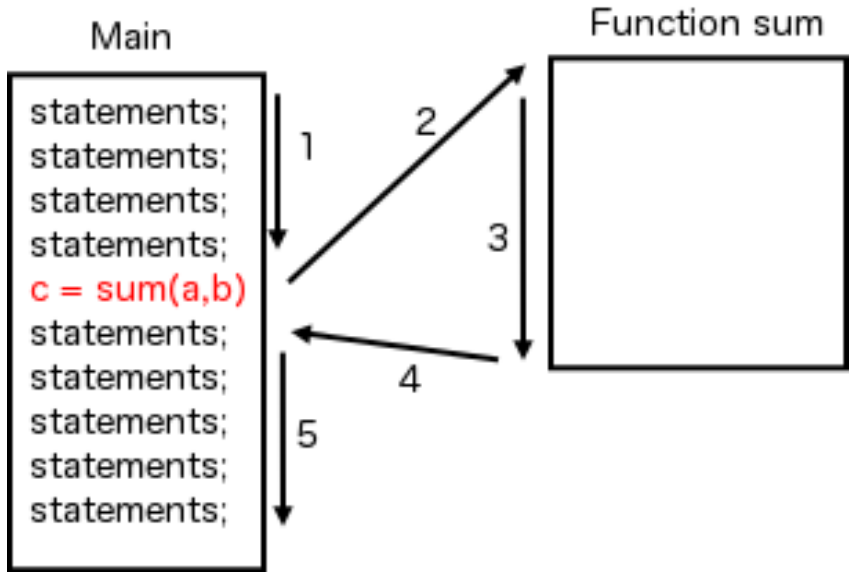
Extending our toolbox

- Functions are blocks of code, written to solve a very specific problem.
- Functions are defined outside the main program.
- Favour code reuse and top-down problem solving approach.
- How to call a function ?
- What is a prototype ?

```
datatype funName(datatype arg1, datatype arg2, ...);
```

Calling a function

- What happens when we call a function ?
- Passing arguments to a function by value.



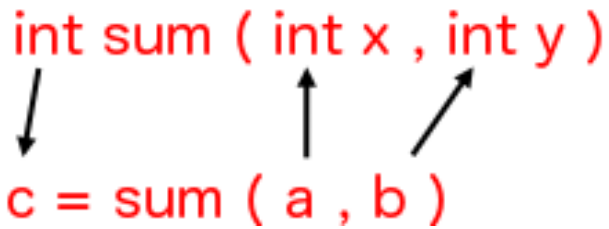

```
int a = 1; b = 2;
int c = sum(a,b);
int d = sum(1,2);
```

Operator vs Functions

- You have been introduced to *operators*.
  - Mostly basic mathematical functions
  - The operator '+' has arity of 2 (binary operator)
- Let's assume the following function
  - prototype: int sum(int a, int b);
  - usage: total = sum(10,20);
  - more usage: total = sum(x+1,sum(y,z));
- What is the difference then, between the following:
  - a = (x+y)+z;
  - a = sum(sum(x,y),z);
  - a = sum(x+y,z);

Inputs and output datatypes

- The call of a function has to match the datatypes specified in the prototype.
- For example, for the *sum* function presented earlier, the mapping is the following:



- What happens if *c* is a *double* ?
- What happens if *a* is a *double* ?

Type conversion (coercion)

- The C/C++ compiler will do an implicit type conversion (called coercion) to match the datatypes before calling the function.
- If unnoticed, this behavior could be the root of many problems.

```
...
int i = 2;
float f = 2.5;
cout « "output: " « i « ", " « f « ", " « sum(i,f);
...
```

Type conversion (coercion)

- The C/C++ compiler will do an implicit type conversion (called coercion) to match the datatypes before calling the function.
- If unnoticed, this behavior could be the root of many problems.

```
...
int i = 2;
float f = 2.5;
cout « "output: " « i « ", " « f « ", " « sum(i,f);
...
output: 2, 2.5, 4
```

Standard library functions

- There are many standard functions in C/C++ ready to use.
- You cannot remember all of them.
- For Unix/Linux/Mac, manpages are a good place to start
  - apropos command example
  - man example
- For Windows, one example
  - http://www.cplusplus.com/reference/clibrary
- What are you looking for in the docs?
- Lots of pages of errors ?

Stateless vs stateful functions

- How is the result of a function produced ?
- When we do need internal state ?
- *pow(a,b) function*
- *rand()* function: we cannot forecast the result
- *time()* function ?
  - The system current time in seconds since 1st of January of 1970.

Stateless functions

- Many functions in the standard library are stateless, their output is ONLY determined by the inputs.
- There are C functions for most of the mathematical functions such as sine, cosine, power, abs, and so on.
- For Unix/Mac/Linux, you can query the man pages
  - man 3 math
- For Windows, check out
  - http://www.cplusplus.com/reference/clibrary/cmath

Example: Calculate the Euclidian distance between 2 points

```
cin » x1 » y1 » x2 » y2;
cout « "The distance is: ";
cout « sqrt(pow(x2-x1,2) + pow(y2-y1,2));
```

Stateful functions

- Stateful functions rely on some form of internal or external state which will allow them to produce different outputs each time they are used.
- The following code, will print out the current year, and 4 random values between 0 and 99

```
int seconds_year = 31104000;
cout « "Year: " « 1970 + (time(NULL)/seconds_year);
srand(time(NULL)); // about repeatability.
cout « "Random values: " « rand()
cout « rand()
cout « rand()
cout « rand()


Year: 2012
Random values: 20, 32, 12, 81
```