MAT 685: C++ for Mathematicians

Numbers

John Perry

University of Southern Msississippi

Spring 2017

Outline

Outline

Variable?

Name representing data in computer's memory

- first character?
  - upper-/lower- case letter
  - underscore
  - avoid upper-case, underscore
- next characters?
  - upper-/lower-case letter
  - underscore
  - digit
- cannot be a keyword (word w/special meaning to C++)

Examples

`center_x`, `center_y`, `_my_data`

Type?

- Specifies data's characteristics: what *kind*

Type?

- Specifies data's characteristics: what *kind*
- Machine types
  - boolean
  - numerical
  - character
  - pointer

Type?

- Specifies data's characteristics: what *kind*
- Machine types
  - boolean
  - numerical
  - character
  - pointer
- Structured types
  - array
  - enumeration
  - structured
    - record
    - union
    - class

Type systems

weak type system variable's type ill-defined, changeable

- introduce variables without specifying type
- type can change
- flexible, interactive
- BASIC, Python, Sage

Type systems

weak type system variable's type ill-defined, changeable

- introduce variables without specifying type
- type can change
- flexible, interactive
- BASIC, Python, Sage

strong type system variable's type carefully checked

- well-defined before use
- type cannot change
- typically fast
- C++, Eiffel, Fortran
- can abuse via "cast" or conversion

Outline

Outline

Depends on "bit length"

ℓ: "bit length"

`short` $\ell \geq 16$

`int` $\ell \geq$ `short` ≥ 16

`long` $\ell \geq$ `int` ≥ 32

`long long` $\ell \geq$ `long` ≥ 64

Depends on "bit length"

ℓ: "bit length"

`short`	$\ell \geq 16$
`int`	$\ell \geq$ `short` ≥ 16
`long`	$\ell \geq$ `int` ≥ 32
`long long`	$\ell \geq$ `long` ≥ 64
T	smallest range is $\left[-2^{\ell}, 2^{\ell} - 1\right)$ ("signed")
`unsigned` T	change range to $\left[0, 2^{\ell+1} - 1\right)$

Example 1

```
int a;
unsigned long b;
```

Questions

- What values can `a` contain?
- What values can `b` contain?

Example 1

```
int a;
unsigned long b;
```

Questions

- What values can a contain? $-2^{16} \leq \mathtt{a} \leq 2^{16}$
- What values can b contain?

Example 1

```
int a;
unsigned long b;
```

Questions

- What values can a contain? $-2^{16} \leq \texttt{a} \leq 2^{16}$
- What values can b contain? $0 \leq \texttt{a} \leq 2^{33} - 1$

Example 2

```
#include <iostream>
using std::cout; using std::endl;

int main() {
  long x, y;

  x = 3;
  y = 4;

  cout << x << " + " << y << " = ";
  cout << x + y << endl;
  return 0;
}
```

Type matters!

$$2^{16} = 65536 > 1000 = 10^3$$

- if you multiply two “small” integers, you can get a “larger” one
- product must fit in type of destination!

Overflow

Mathematical operation w/larger result than allowed by type

Example of overflow

```
#include <iostream>
using std::cout; using std::endl;

int main() {
  short thousand = 1000;
  short million = thousand * thousand;

  cout << "According to this computer, ";
  cout << thousand << " squared is\n";
  cout << "\t" << million << endl;
}
```

Result on my home computer

```
$ ./a.out
According to this computer, 1000 squared is
       16960
```

Outline

Floating-point numbers

- no exact represention of real
- approximation by **floating point**
  - $a \times 10^b$
- slower, inexact, but well-specified operations
- no overflow, but division by small numbers problematic

Example

`1e+06` = 1×10^6

“Real” type names

float	machine-dependent
double	no less precise than float
long double	no less precise than double

Example of non-overflow

```
#include <iostream>
using std::cout; using std::endl;

int main() {
  float thousand = 1000;
  float million = thousand * thousand;

  cout << "According to this computer, ";
  cout << thousand << " squared is\n";
  cout << "\t" << million << endl;
}
```

Result on my home computer

```
$ ./a.out
According to this computer, 1000 squared is
        1e+06
```

"Complex" type names

```
#include <complex>
using std::complex;

complex <T> varname;
```

...where T is another numerical type

“Complex” type names

```
#include <complex>
using std::complex;

complex <T> varname;
```

...where *T* is another numerical type

`complex<double>` yer usual complex type

`complex<long>` “Gaussian” integers

“Complex” type names

```
#include <complex>
using std::complex;

complex <T> varname;
```

...where *T* is another numerical type

`complex<double>` yer usual complex type

`complex<long>` “Gaussian” integers

(**templated** type, discussed later)

Too long? `typedef` it

Typing `complex<`*T*`>` repeatedly is tiresome!

`typedef` *T* *N*`;`

Defines *N* as a shortcut for *T*

Too long? `typedef` it

Typing `complex<`T`>` repeatedly is tiresome!

`typedef` T N`;`

Defines N as a shortcut for T

Place *outside* program block, preferably immediately after `#include`'s.

Example (p. 1/2)

Program 2.7 (pp. 23–24, slightly modified)

Listing 1: `complex_demo.cpp`

```
#include <complex>
using std::complex;
#include <iostream>
using std::cout; using std::endl;

typedef complex<double> CC;

int main() {
  CC x(3,4);        // define x = 3+4i
  CC z;             // define z to be complex
  z = CC(2,7);      // assign z = 2+7i
  CC i(0,1);        // define i = sqrt(-1)
```

Example (p. 2/2)

Program 2.7 (pp. 23–24, slightly modified)

```
  cout << "z = " << z << endl;
  cout << "x = " << x << endl;
  cout << "z + x = " << z + x << endl;
  cout << "z*x = " << z*x << endl;
  cout << "z/x = " << z/x << endl;
  z = 5. - 4.*i;

  cout << "Now z = " << z << endl;

  cout << "The real part of z is " << z.real()
       << "\nand the imaginary part is "
       << z.imag() << endl;
  return 0;
}
```

Example, run on my computer

```
$ ./a.out
z = (2,7)
x = (3,4)
z + x = (5,11)
z*x = (-22,29)
z/x = (1.36,0.52)
Now z = (5,-4)
The real part of z is 5
and the imaginary part is -4
```

Outline

`bool` and `char`

`bool` value can be `true` or `false`

- old style: 1 (true) or 0 (false)
- output displayed in old style

`bool` and `char`

`bool` value can be `true` or `false`

- old style: 1 (true) or 0 (false)
- output displayed in old style

`char` character

- enclosed in *single* quotes
- 256 possibilities, defined by ASCII standard
- many the usual ones: `a`, `Y`, `1`, `_`
- includes “escape” codes: `'\n'`, `'\t'`, others

`bool` and `char`

`bool` value can be `true` or `false`
- old style: 1 (true) or 0 (false)
- output displayed in old style

`char` character
- enclosed in *single* quotes
- 256 possibilities, defined by ASCII standard
- many the usual ones: `a`, `Y`, `1`, `_`
- includes "escape" codes: `'\n'`, `'\t'`, others

`string` sequence of `char`
- enclosed in *double* quotes

Example

```
#include <iostream>
using std::cout; using std::endl;
#include <string>
using std::string;

int main() {
  bool truth = 1;
  bool same_truth = true;
  const string message = "Is the truth the same truth? ";

  cout << message << (truth == same_truth) << endl;
}
```

Result on my home computer

```
$ ./a.out
Is the truth the same truth? 1
```

Points to ponder

- ```
  const <type> <variable_name>
  ```
  `const` tells the compiler that a variable will not change
- ```
  (truth == same_truth)
  ```
  `==` discussed below; parentheses needed for order of operations
- displays `1`, not `true` or even `t` or `T`

Outline

Outline

Numerical operations

operation	usage	notes
addition	`a + b`	watch for overflow
subtraction	`a - b`	watch for overflow
multiplication	`a * b`	watch for overflow
division	`a / b`	integers? quotient only
modular division	`a % b`	remainder can be negative

Example

```
#include <iostream>
using std::cin; using std::cout;
using std::endl;

int main() {
  int a, b;

  cout << "Enter the first number --> ";
  cin >> a;
  cout << "Enter the second number --> ";
  cin >> b;
  cout << a << " % " << b << " = ";
  cout << a % b << endl;

  return 0;
}
```

Result on my home computer

```
$ ./a.out
Enter the first number --> 5
Enter the second number --> -3
5 % -3 = 2
$ ./a.out
Enter the first number --> -5
Enter the second number --> 3
-5 % 3 = -2
$ ./a.out
Enter the first number --> -5
Enter the second number --> -3
-5 % -3 = -2
```

Operate and assign

operation	**usage**	**notes**
increment by 1	`++a` or `a++`	pre- or postincrement
decrement by 1	`--a` or `a--`	pre- or postdecrement
increment by `b`	`a += b`	result in `a`; watch for overflow
decrement by `b`	`a -= b`	result in `a`; watch for overflow
dilate by `b`	`a *= b`	result in `a`; watch for overflow
contract by `b`	`a /= b`	result in `a`; integers? quotient only
modular division	`a %= b`	result in `a`; remainder can be negative

Pre- vs. Post- in/decrement?

- `++a` increments `a` *before* using it
- `a++` increments `a` *after* using it

Pre- vs. Post- in/decrement?

- `++a` increments `a` *before* using it
- `a++` increments `a` *after* using it

```
#include <iostream>
using std::cout; using std::endl;

int main() {
  int a;
  a = 10; cout << ++a << endl;
  a = 10; cout << a++ << endl;
  return 0;
}
```

Pre- vs. Post- in/decrement?

- `++a` increments `a` *before* using it
- `a++` increments `a` *after* using it

```
#include <iostream>
using std::cout; using std::endl;

int main() {
  int a;
  a = 10; cout << ++a << endl;
  a = 10; cout << a++ << endl;
  return 0;
}
```

```
$ ./a.out
11
10
```

Exponentiation?

Not a *basic* operator. Use library functions:

function	**usage**	**notes**
e^b	`exp(b)`	best to use `double` for result
a^b	`pow(a,b)`	best to use `double` for result

Exponentiation?

Not a *basic* operator. Use library functions:

function	usage	notes
e^b	`exp(b)`	best to use `double` for result
a^b	`pow(a,b)`	best to use `double` for result

```
#include <iostream>
using std::cout; using std::endl;
#include <cmath>
using std::pow;

int main() {
  double e = exp(1.);
  double pi = M_PI;

  cout << "e to the pi is " << exp(pi) << endl;
  cout << "pi to the e is " << pow(pi, e) << endl;
}
```

Exponentiation?

Not a *basic* operator. Use library functions:

function	usage	notes
e^b	`exp(b)`	best to use `double` for result
a^b	`pow(a,b)`	best to use `double` for result

```
#include <iostream>
using std::cout; using std::endl;
#include <cmath>
using std::pow;

int main() {
  double e = exp(1.);
  double pi = M_PI;

  cout << "e to the pi is " << exp(pi) << endl;
  cout << "pi to the e is " << pow(pi, e) << endl;
}
```

```
$ ./a.out
e to the pi is 23.1407
pi to the e is 22.4592
```

Numerical comparisons

Return `true` or `false` depending on values of a and b

comparison	**usage**	**notes**
equal?	`a == b`	*two* equals signs; forgetting can be catastrophic!
different?	`a != b`	what we call $a \neq b$
smaller?	`a <= b`	what we call $a \leq b$
strictly smaller?	`a < b`	
strictly larger?	`a > b`	
larger?	`a >= b`	what we call $a \geq b$

Outline

Old style (book)

Return true or false depending on values of a and b

comparison	**usage**	**notes**
equal?	`a == b`	*two* equals signs
different?	`a != b`	
logical negation?	`!a`	what we call $\neg a$ or $\sim a$
logical and?	`a && b`	true iff both true
logical or?	`a || b`	true iff at least one true
logical xor?	`a ^ b`	true iff exactly one true

New style (clearer)

Return true or false depending on values of a and b

comparison	**usage**	**notes**
equal?	`a == b`	*two* equals signs
different?	`a != b`	
logical negation?	`not a`	what we call $\neg a$ or $\sim a$
logical and?	`a and b`	true iff both true
logical or?	`a or b`	true iff at least one true
logical xor?	`a xor b`	true iff exactly one true

Example

```
#include <iostream>
using std::cout; using std::endl;

int main() {
  bool yes = true;
  bool no = false;

  cout << "yes = " << yes
       << "; no = " << no << endl;
  cout << "not yes? " << not yes << endl;
  cout << "not no? " << not no << endl;
  cout << "yes and no? " << (yes and no)
       << endl;
  cout << "yes or no? " << (yes or no)
       << endl;
  cout << "yes xor no? " << (yes xor no)
       << endl;

  return 0;
}
```

Result on my home computer

```
$ ./a.out
yes = 1; no = 0
not yes? 0
not no? 1
yes and no? 0
yes or no? 1
yes xor no? 1
```

Outline

Summary

- C++ strongly typed
- basic types: numerical, boolean, character, pointer
- numerical types allow for exact or approximate arithmetic
- many basic operations available
    - some common operations require math library

Homework

pp. 28–29 #2.1–2.8