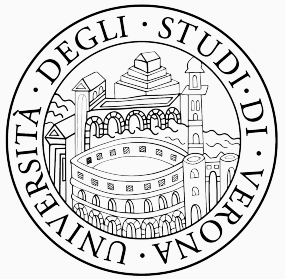
Modern C++ Programming

4. Utilities

Federico Busato

University of Verona, Dept. of Computer Science

2018, v1.0



Agenda

- **Math Functions**
  - CMath library
  - Numerical limits
  - Integer division
- **Algorithm Library**
- **String**
  - Methods
  - Operators
  - Conversion
- **Random Numbers**
  - Period and quality
  - Engines
  - Distributions
- **Time Measuring**
  - Wall-clock time
  - User time
  - System time

Math Functions

```
#include <cmath>
using namespace std;
```

- `fabs(x)` computes absolute value, $|x|$, C++11
- `exp(x)` returns e raised to the given power, e^x
- `exp2(x)` returns 2 raised to the given power, 2^x, C++11
- `log(x)` computes natural (base e) logarithm, $\log_e(x)$
- `log10(x)` computes base 10 logarithm, $\log_{10}(x)$
- `log2(x)` computes base 2 logarithm, $\log_2(x)$, C++11
- `pow(x, y)` raises a number to the given power, x^y
- `sqrt(x)` computes square root, $\sqrt{x}$

- `cqrt(x)` computes cubic root, $\sqrt[3]{x}$, C++11
- `sin(x)` computes sine, $\sin(x)$
- `cos(x)` computes cosine, $\cos(x)$
- `tan(x)` computes tangent, $\tan(x)$
- `ceil(x)` nearest integer not less than the given value, $\lceil x \rceil$
- `floor(x)` nearest integer not greater than the given value, $\lfloor x \rfloor$
- `round|lround|llround(x)` nearest integer, $\lfloor x + \frac{1}{2} \rfloor$
(return type: floating point, `long`, `long long` respectively)

Math functions in C++11 can be applied directly to integral types without implicit/explicit casting (return type: floating point).

Full list: `en.cppreference.com/w/cpp/numeric/math`

Numerical Limits

Get numeric limits of a given type: C++11

```
#include <limits>
using namespace std;
```

```
T numeric_limits<T>:: max()  // returns the maximum finite value
                             // representable by the numeric type T

T numeric_limits<T>:: min()  // returns the minimum finite value
                             // representable by the numeric type T

T numeric_limits<T>:: lowest()  // returns the lowest finite value
                                // representable by the numeric type T
```

Integer Division

Integer ceiling division and rounded division:

- **Ceiling Division:** $\left\lceil \frac{\texttt{value}}{\texttt{div}} \right\rceil$

```
unsigned ceil_div(unsigned value, unsigned div) {
    return (value + div - 1) / div;
} // note: may overflow
```

- **Rounded Division:** $\left\lfloor \frac{\texttt{value}}{\texttt{div}} + \frac{1}{2} \right\rfloor$

```
unsigned round_div(unsigned value, unsigned div) {
    return (value + div / 2) / div;
} // note: may overflow
```

Algorithm Library

`std` algorithms can be applied to **any objects** (see next lectures). In these slides, we focus on primitives types and array only

```
#include <algorithm>
```

- `swap(value1, value2)` Swaps the values of two objects
- `min(x, y)` Finds the minimum value between `x` and `y`
- `max(x, y)` Finds the maximum value between `x` and `y`
- `min_element(begin, end)` (returns a pointer)
  Finds the minimum element in the range [`begin, end`)
- `max_element(begin, end)` (returns a pointer)
  Finds the maximum element in the range [`begin, end`)
- `minmax_element(begin, end)` C++11 (returns pointers <min,max>)
  Finds the minimum and the maximum element in the range [`begin, end`)

full list: en.cppreference.com/w/cpp/algorithm

- `equal(begin1, end1, begin2)`
  Determines if two sets of elements are the same in
  [begin1, end1), [begin2, begin2 + end1 - begin1)
- `mismatch(begin1, end1, begin2)` (returns pointers <pos1,pos2>)
  Finds the first position where two ranges differ in
  [begin1, end1), [begin2, begin2 + end1 - begin1)
- `find(begin, end, value)` (returns a pointer)
  Finds the first element in the range [begin, end) equal to value
- `count(begin, end, value)`
  Counts the number of elements in the range [begin, end) equal to value

- `sort(begin, end)` (in-place)
  Sorts the elements in the range [`begin`, `end`) in ascending order
- `merge(begin1, end1, begin2, end2, output)`
  Merges two sorted ranges [`begin1`, `end1`), [`begin2`, `end2`), and store the results in [`output`, `output` + `end1` - `start1`)
- `unique(begin, end)` (in-place)
  Removes consecutive duplicate elements in the range [`begin`, `end`)
- `binary search(begin, end, value)`
  Determines if an element `value` exists in the (sorted) range [`begin`, `end`)
- `accumulate(begin, end, value)`
  Sums up the range [`begin`, `end`) of elements with initial `value` (common case equal to zero)
- `partial_sum(begin, end)` (in-place)
  Computes the inclusive prefix-sum of the range [`begin`, `end`)

- `fill(begin, end, value)`
  Fills a range of elements [`begin`, `end`) with `value`
- `iota(begin, end, value)` C++11
  Fills the range [`begin`, `end`) with successive increments of the starting `value`
- `copy(begin1, end1, begin2)`
  Copies the range of elements [`begin1`, `end1`) to the new location [`begin2`, `begin2 + end1 - begin1`)
- `swap_ranges(begin1, end1, begin2)`
  Swaps two ranges of elements
  [`begin1`, `end1`), [`begin2`, `begin2 + end1 - begin1`)
- `remove(begin, end, value)` (in-place)
  Removes elements equal to `value` in the range [`begin`, `end`)

- `includes(begin1, end1, begin2, end2)`
  Checks if the (sorted) set [begin1, end1) is a subset of [begin2, end2)
- `set_difference(begin1, end1, begin2, end2, output)`
  Computes the difference between two (sorted) sets
- `set_intersection(begin1, end1, begin2, end2, output)`
  Computes the intersection of two (sorted) sets
- `set_symmetric_difference(begin1, end1, begin2, end2, output)`
  Computes the symmetric difference between two (sorted) sets
- `set_union(begin1, end1, begin2, end2, output)`
  Computes the union of two (sorted) sets
- `make_heap(begin, end)` Creates a max heap out of the range of elements
- `push_heap(begin, end)` Adds an element to a max heap
- `pop_heap(begin, end)` Remove an element (top) to a max heap

```cpp
#include <algorithm>
using namespace std;

int a = max(2, 5); // a = 5
int array1[] = {7, 6, -1, 6, 3};
int array2[] = {8, 2, 0, 3, 7};

int  b = *max_element(array1, array1 + 5);   // b = 7
auto c = minmax_element(array1, array1 + 5);
//c.first = -1, c.second = 7
bool d = equal(array1, array1 + 5, array2); // d = false

sort(array1, array1 + 5);          // [-1, 3, 6, 6, 7]
unique(array1, array1 + 5);        // [-1, 3, 6, 7]
int e = accumulate(array1, array1 + 5, 0); // 15
partial_sum(array1, array1 + 5); // [-1, 2, 8, 15]
iota(array1, array1 + 5, 2);       // [2, 3, 4, 5, 6]
make_heap(array2, array2 + 5);   // [8, 7, 0, 3, 2]
```

String

Definition (String)

C++ Strings are wrappers of character sequences

More flexible and safer than raw char array but can be slower

```
#include <string>
using namespace std;

int main() {
    string a;               // empty string
    string b("first");

    using namespace std::string_literals;  // C++14
    string c = "second"s;                  // C++14
}
```

- `empty()` returns **`true`** if the string is empty, **`false`** otherwise
- `size()` returns the number of characters in the string
- `find(string)` returns the position of the first substring equal to the given character sequence or **`npos`** if no substring is found
- `rfind(string)` returns the position of the last substring equal to the given character sequence or **`npos`** if no substring is found
- `find_first_of(char_seq)` returns the position of the first character equal to one of the characters in the given character sequence or **`npos`** if no characters is found
- `find_last_of(char_seq)` returns the position of the last character equal to one of the characters in the given character sequence or **`npos`** if no characters is found

`npos` special value returned by string methods

- `new_string substr(start_pos)`
  returns a substring [start_pos, end]
  `new_string substr(start_pos, count)`
  returns a substring [start_pos, start_pos + count)
- `clear()` removes all characters from the string
- `erase(pos)` removes the character at position
  `erase(start_pos, count)`
  removes the characters at positions [start_pos, start_pos + count)
- `replace(start_pos, count, new_string)`
  replaces the part of the string indicated by [start_pos, start_pos + count) with `new_string`
- `c_str()`
  returns a pointer to the raw char sequence

- **access specified character** `string1[i]`
- **string copy** `string1 = string2`
- **string compare** `string1 == string2`
  works also with `!=`, $<$, $\leq$, $>$, $\geq$
- **concatenate two strings**
  `string_concat = string1 + string2`
- **append characters to the end** `string1 += string2`

Converts a string to a numeric value C++11:

- `stoi(string)` string to signed integer
- `stol(string)` string to long signed integer
- `stoul(string)` string to long unsigned integer
- `stoull(string)` string to long long unsigned integer
- `stof(string)` string to floating point value (float)
- `stod(string)` string to floating point value (double)
- `stold(string)` string to floating point value (long double)

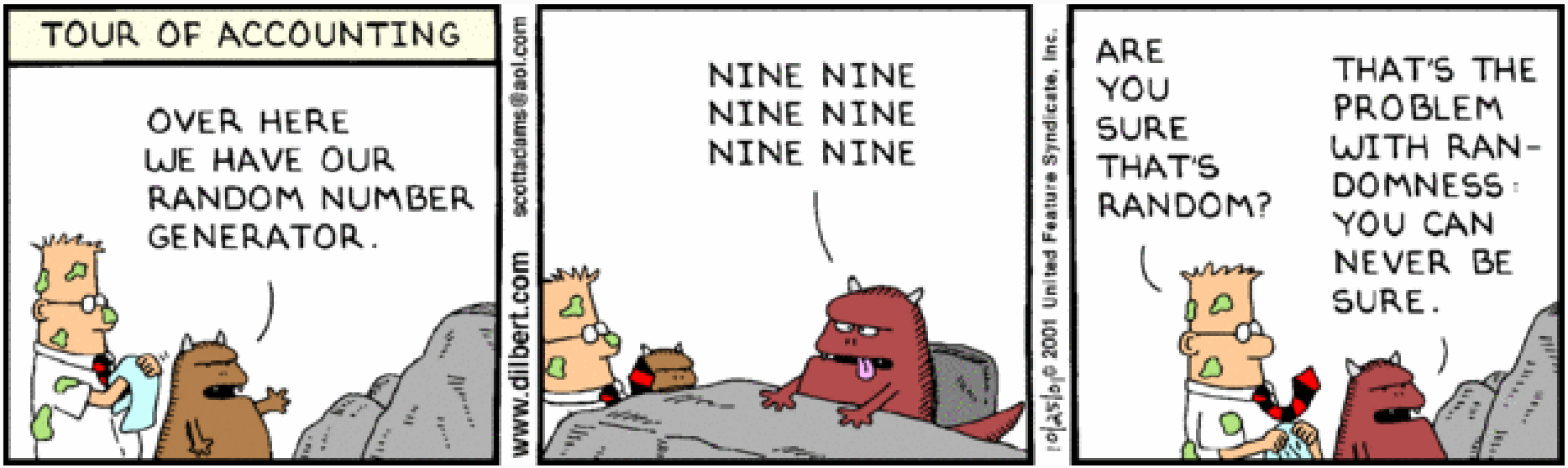
Converts a numeric value to a string C++11:

- `to_string(numeric_value)` numeric value to string

```
string str("si vis pacem para bellum");

cout << str.size();       // print 24
cout << str.find("vis"); // print 3
cout << str.find_last_of("bla"); // print 21, `l' found

cout << str.substr(7, 5);// print "pacem", pos=7 and count=5
cout << str[1];           // print `i'
cout << (str == "vis");  // print false
cout << (str < "z");      // print true
const char* raw_str = str.c_str();

cout << string("a") + "b";      // print "ab"
cout << string("ab").erase(0); // print `b'

char* str2 = "34";
int a = stoi(str2);          // a = 34;
string str3 = to_string(a); // str3 = "34"
```

- Conversion from integer to char letter (e.g. 3 → 'C'):
  `static_cast<char>('A' + value)`
  `value` ∈ [0, 25] (English alphabet)
- Conversion from char to integer (e.g. 'C' → 3):
  `value - 'A'`
  `value` ∈ [0, 25]
- Conversion from digit to char number (e.g. 3 → '3'):
  `static_cast<char>('0' + value)`
  `value` ∈ [0, 9]
- char to string `std::string(1, char_value)`

Random Number


TOUR OF ACCOUNTING
OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.
www.dilbert.com scottadams@aol.com
NINE NINE NINE NINE NINE NINE
© 2001 United Feature Syndicate, Inc.
ARE YOU SURE THAT'S RANDOM?
THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.

The problem:

C rand() function produces poor quality random numbers

- C++14 discourage the use of `rand()` and `srand()`

C++11 introduces pseudo random number generation (PRNG) facilities to produce random numbers by using combinations of generators and distributions

A random generator requires four steps:

(1) **Select the seed**

(2) **Define the random engine**

```
<type_of_random_engine> generator(seed)
```

(3) **Define the distribution**

```
<type_of_distribution> distribution(range_start, range_end)
```

(4) **Produce the random number**

```
distribution(generator)
```

Simplest example:

```
#include <iostream>
#include <random>
using namespace std;

int main() {
    unsigned seed = ...;
    default_random_engine generator(seed);
    uniform_int_distribution<int> distribution(0, 9);
    cout << distribution(generator);  // first random number
    cout << distribution(generator);  // second random number
}
```

It generates two random integer numbers in the range [0, 9] by using the default random engine

Given a **seed**, the generator produces always the **same sequence**

The seed should be selected randomly by using the actual time:

```
#include <random>
#include <chrono>
using namespace std;

int main() {
    unsigned seed = chrono::system_clock::now()
                    .time_since_epoch().count();
    default_random_engine generator(seed);
}
```

`chrono::system_clock::now()` return an object representing the current point in time

`.time_since_epoch().count()` returns the count of ticks that have elapsed since January 1, 1970 (midnight UTC/GMT)

Pseudorandom Number Generator (PRNG)

Definition (PRNG Period)

The period (or cycle length) of a PRNG is the length of the sequence of numbers that the PRNG generates before repeating.

Definition (PRNG Quality)

(informal) If it's hard to distinguish a generator's output from truly random sequences we call it a high quality generator. If it's easy, we call it a low quality generator.

Generator	Quality	Period	Performance
Linear congruential	Poor	10^9	fast
Mersenne Twister	High	10^{6000}	good
Subtract-with-carry	Highest	10^{171}	slow

Random Engines

- **Default random engine** Implementation defined
- **Linear congruential** The simplest generator engine. It implements the following transition algorithm:

$$x_{i+1} = (\alpha x_i + c) mod\ m$$

where α, c, m are implementation defined
The generator has a period of m, where m is $2^{31} - 1$

- **Mersenne Twister** *(M. Matsumoto and T. Nishimura, 1997)*
Fast generation of high-quality pseudorandom number. It relies on Mersenne prime number. (used as default random generator in linux)
The generator `mt19937, mt19937_64` has a period of $2^{(n-1)*w} - 1$, where w is 32 and n is 624, $\approx 10^{6000}$

- **Subtract-with-carry** *(G. Marsaglia and A. Zaman, 1991)*
Pseudo-random generation based on Lagged Fibonacci algorithm (used for example by physicists at CERN)
The generator `ranlux24_base/ranlux48_base` have a period of 10^{171}

Distribution

Common distributions:

- **Uniform random**

  `uniform_int_distribution<T>(range_start, range_end)`

  where T is integral type

  `uniform_real_distribution<T>(range_start, range_end)`

  where T is floating point type

- **Normal distribution** $P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$

  `normal_distribution<T>(mean, std_dev)`

  where T is floating point type

- **Exponential distribution** $P(x, \lambda) = \lambda e^{-\lambda x}$

  `exponential_distribution<T>(lambda)`

  where T is floating point type

Examples

```
unsigned seed = chrono::system_clock::now()
                .time_since_epoch().count();

minstd_rand0  lc1_generator(seed);  // original linear congruential
minstd_rand   lc2_generator(seed);  // linear congruential (better tuning)
mt19937       mt_generator(seed);   // standard mersenne twister (32-bit)
mt19937_64    mt64_generator(seed); // standard mersenne twister (64-bit)
ranlux24_base swc24_generator(seed);// subtract with carry (24-bit)
ranlux48_base swc48_generator(seed);// subtract with carry (48-bit)

uniform_int_distribution<int>    int_distribution(0, 10);
uniform_real_distribution<float> real_distribution(-3.0f, 4.0f);
exponential_distribution<float>  exp_distribution(3.5f);
normal_distribution<double>      norm_distribution(5.0, 2.0);

lc1_generator.discart(10);   // advances the internal state by 10 times
// i.e. the sequence start point is equal to apply distribution() 10 times
```

Time Measuring

Definition (Wall-Clock/Real time)

It is the human perception of the passage of time from the start to the completion of a task.

Definition (User/CPU time)

The amount of time spent by the CPU to compute in user code.

Definition (System time)

The amount of time spent by the CPU to compute system calls (including I/O calls) executed into kernel code.

Note: if the system workload (except the current program) is very low and the program uses only one thread then

`Wall-clock time = User time + System time`

`::gettimeofday()` (linux)

```
#include <time.h>     //struct timeval
#include <sys/time.h> //gettimeofday()
#include <iostream>
using namespace std;

int main() {
    struct timeval start, end; // timeval {second, microseconds}
    ::gettimeofday(&start, NULL);
    ...   // code
    ::gettimeofday(&end, NULL);

    long start_time = start.tv_sec * 1000000 + start.tv_usec;
    long end_time   = end.tv_sec * 1000000 + end.tv_usec;
    cout << "Elapsed: " << end_time - start_time; // in microsec
}
```

Problems: not portable, the time is not monotonic increasing (timezone)

`std::chrono` C++11

```cpp
#include <iostream>
#include <chrono>
using namespace std;

int main() {
    auto start_time = chrono::system_clock::now();
    ... // code
    auto end_time = chrono::system_clock::now();

    chrono::duration<double> diff = end_time - start_time;
    cout << "Elapsed: " << diff.count(); // in seconds
    cout << chrono::duration_cast<milli>(diff).count(); // in ms
}
```

Problems: The time is not monotonic increasing (timezone)

An alternative of `system_clock` is `steady_clock` which ensures monotonic increasing time

`std::clock` C++11

```cpp
#include <iostream>
#include <chrono>
using namespace std;

int main() {
    clock_t start_time = clock();
    ... // code
    clock_t end_time = clock();

    float diff = static_cast<float>(end_time - start_time)
                 / CLOCKS_PER_SEC;
    cout << "Elapsed: " << diff; // in seconds
}
```

`::times` (linux)

```cpp
#include <iostream>
#include <sys/times.h>
using namespace std;

int main() {
    struct ::tms start_time, end_time;
    ::times(&start_time);
    ... // code
    ::times(&end_time);

    auto user_diff = end_time.tmus_utime - start_time.tms_utime;
    auto sys_diff  = end_time.tms_stime - start_time.tms_stime;
    float user = static_cast<float>(user_diff) / ::sysconf(_SC_CLK_TCK);
    float sys  = static_cast<float>(sys_diff) / ::sysconf(_SC_CLK_TCK);
    cout << "user time: " << user;  // in seconds
    cout << "system time: " << sys; // in seconds
}
```