Modern C++ Programming

19. Utilities

Federico Busato

2026-01-06

Table of Contents

Table of Contents

Table of Contents

Views Introduction and `std::span`

Views

A **non-owning view** provides access to resources without taking ownership or responsibility for managing their lifecycle, such as allocation and deallocation

C++ views have *trivially-copyable* semantic (enforced in C++23) and should be always passed *by-value*

C++ provides the following views:

- `std::string_view` string view (sequence of characters)
- `std::span` 1D contiguous data view
- `std::mdspan` N-dimensional data view

`std::span` - Overview

C++20 `std::span` is a *non-owning view* of a contiguous sequence of elements

`std::span` provides a **uniform interface** for contiguous data, such as `std::vector`, `std::array`, and `raw pointers`

- `std::span` is widely used as function parameter
- `std::span` works with all Standard Library algorithms that operate on ranges or iterators

In addition, `std::span` ensures **interoperability with legacy code**, allowing to replace the common pattern *pointer + size* with a safer alternative

- Associate the buffer size and the pointer only during the construction (sub-span is also allowed)
- Bounds-checked access in standard environment (C++26 `at()`) or in debug mode (see `Debugging and Testing - Hardening Techniques`)

`std::span` - Basic Properties

`std::span` supports any data type `T`. It supports both *modifiable* (`std::span<T>`) and *read-only* views (`std::span<const T>`), allowing to enforce const-correctness

`std::span` can either have a *static* extent, in which case the number of elements in the sequence is known at compile-time, or a *dynamic* extent, determined at run-time

```
template<
    typename T,
    size_t   Extent = std::dynamic_extent>
class span;
```

std::span - Constructions

```cpp
#include <span>
#include <array>
#include <vector>

int array1[] = {1, 2, 3};
std::span s1{array1};     // static extent

std::array array2 = {1, 2, 3};
std::span s2{array2};     // static extent

auto array3 = new int[3];
std::span s3{array3, 3}; // dynamic extent

std::vector<int> v{1, 2, 3};
std::span s4{v};          // dynamic extent
```

std::span - Function Parameter

```cpp
#include <algorithm>
#include <span>

void f(std::span<int> span) { // mutable function parameter
    for (auto x : span)       // range-based loop (safe)
        cout << x;
    std::fill(span.begin(), span.end(), 3); // std algorithms support
}

void g(std::span<const int> span) {} // read-only function parameter

int array1[] = {1, 2, 3};
f(array1);
g(array1);

auto array2 = new int[3];
f({array2, 3});
```

`std::span` - Methods

Access:

`front()`, `back()`	access the first/last element
`operator[]`, `at(i)`	access specified element, with bounds checking `at(i)`
`data()`	returns a pointer to the underlying data

Size:

`size()`	number of elements
`size_bytes()`	size in bytes
`empty()`	checks if the span is empty

Subspan:

`first(N)`, `first<N>()`	subspan of the first N elements
`last(N)`, `last<N>()`	subspan of the last N elements
`subspan(offset, count)`	generic dynamic subspan
`subspan<offset, count>()`	generic static subspan

std::span - Function Parameter

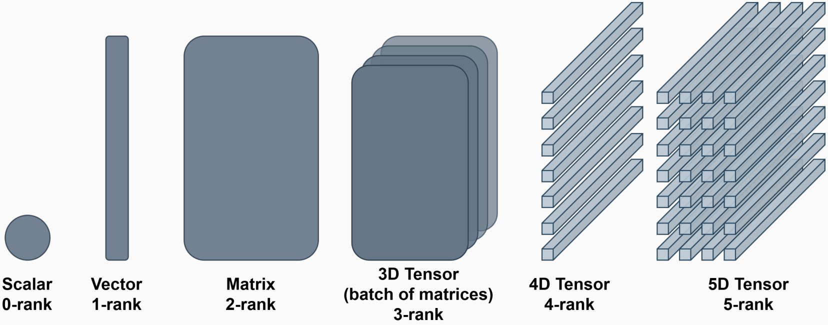
```cpp
#include <span>

int array[] = {1, 2, 3, 4, 5};
std::span span(array);

span.front(); // first element
span.bask();  // last element
span.data();  // pointer equal to 'array'

span.size();        // 5 elements
span.size_bytes();  // 20 bytes
span.empty();       // false

span.first(3);       // [1, 2, 3] (dynamic)
span.last(2);        // [4, 5] (dynamic)
span.subspan(1, 3); // [2, 3, 4] (dynamic)
```

std::mdspan

`std::mdspan`

C++23 `std::mdspan` is a non-owning view that allows to represent *multidimensional data*, from scalars to N-dimensional tensors, in a simple and flexible way

`std::mdspan` is particularly suited for scientific computing, graphics, robotics, or more in general in linear algebra applications to *abstract* the underlying data organization and provide a *uniform interface*



`std::mdspan` class

```
template<
    typename T,
    typename Extents,
    typename Layout,
    typename Accessor>
class mdspan;
```

`T`	Type of the elements of the underlying data
`Extents`	Number of dimensions (ranks) and their sizes
`Layout`	Describes how to map multidimensional indices (*domain*) to a single linear index (*codomain*), $\mathbb{N} \times \mathbb{N} \times \ldots \times \mathbb{N} \longrightarrow \mathbb{N}$
`Accessor`	Describes how to access the element pointed out by the linear index

std::mdspan - Example 1

```
#include <mdspan>    compiler explorer

// A: M x K
// B: K x N
// C: M x N
void matrix_mul(float* matrixA, float* matrixB, float* matrixC,
                int M, int N, int K) {
    std::mdspan mdA{matrixA, M, K};
    std::mdspan mdB{matrixB, K, N};
    std::mdspan mdC{matrixC, M, N};
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < K; k++)
                mdC[i, j] += mdA[i, k] + mdB[k, j];
        }
    }
}
```

Basic Methods

`rank()` Returns the number of dimensions

`extent(r)` Returns the *extent*, namely the size, at the rank index `r`

`stride(r)` Returns the *stride* at the rank index `r`, namely the distance across consecutive elements of the same dimensions

`size()` Returns the size of the multidimensional index space, namely the product of all extents. It can be different from the underlying data size, depending on the layout and strides

`data_handle()` Returns the underlying data handle, e.g. a pointer

```
std::mdspan md{matrix_ptr, M, N};
md.rank();     // 2
md.extent(0); // N
md.extent(1); // N
```

```
md.size();          // M * N
md.stride(0);       // N
md.stride(1);       // 1
md.data_handle(); // matrix_ptr
```

```
template<typename  IndexType,
         size_t... Extents>
class extents;
```

`IndexType` Type of the index, e.g. `int`, `size_t`, etc.

`Extents` List of sizes, one for each dimension

- An arbitrary number to specify the size at *compile-time*
- `std::dynamic_extent` if the dimension is determined at *run-time*

```
std::extents<int>        ext1;          // Scalar size
std::extents<int, 4, 5> ext2;          // Matrix size: 4 x 5
std::extents<int, 4, std::dynamic_extent> ext3{4, n}; // Tensor size: 4 × n
```

The standard library also provides convenient alias templates for all-dynamic extents

```
template<typename IndexType, size_t Rank>
using dextents = extents<IndexType, dynamic_extent, ... /*Rank-times*/>

template<size_t Rank, class IndexType = size_t>
using dims = dextents<IndexType, Rank>; // C++26
```

```
std::dextents<int, 3> ext4{a, b, c}; // Tensor dynamic size: a × b × c
std::dims<3>          ext5{a, b, c}; // Tensor dynamic size: a × b × c, C++26
```

std::mdspan - Example 2

```
#include <mdspan> compiler explorer

template<typename T, typename E, typename L, typename A>
void print(std::mdspan<T, E, L, A> md) {
    for (int i = 0; i < md.extent(0); i++) {
        for (int j = 0; j < md.extent(1); j++)
            cout << md[i, j] << ", ";
        cout << "\n";
    }
    cout << "\n";
}

float         array[] = {1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f, 7.0f, 8.0f};
std::dims<2> extents{2, 4};
std::mdspan  md{array, extents};
print(md); // 1, 2, 3, 4,
           // 5, 6, 7, 8,
```

Layout

The mdspan **layout** describes how the elements of the underlying data are arranged in memory.
The main functionality of the **layout** is to map a list of indices to a single offset with the method `operator()(Indices...)`

The standard library provides the following predefined *layouts*:

`layout_right`	Row-major, right-index varies fastest, C/C++
`layout_left`	Column-major, left-index varies fastest, Fortran
`layout_stride`	Arbitrary *dynamic* strides for each dimension
`layout_right_padded`	typically 2-D, row-major with a single customizable stride across rows (second dimension) to exploit alignment, C++26
`layout_left_padded`	typically 2-D, column-major with a single customizable stride across columns (first dimension) to exploit alignment, C++26

`std::layout_right`

mdspan sizes: $e_1 \times e_2 \times \ldots \times e_n$

$$\texttt{layout\_right}(i_1, i_2, \ldots, i_{n-1}, i_n) \to i_1 \cdot \sum_{j=2}^{n} e_j + i_2 \cdot \sum_{j=3}^{n} e_j + \ldots + i_{n-1} \cdot e_n + i_n$$

---

mdspan: $2 \times 4 \times 3$

$$\texttt{layout\_right}(1, 1, 2) \to 1 \cdot 4 \cdot 3 + 1 \cdot 3 + 2 = 16$$

0	1	2
3	4	5
6	7	8
9	10	11

12	13	14
15	**16**	17
18	19	20
21	22	23

`std::layout_left`

mdspan sizes: $e_1 \times e_2 \times \ldots \times e_n$

$$\texttt{layout\_left}(i_1, i_2, \ldots, i_{n-1}, i_n) \to i_1 + i_2 \cdot e_1 + i_3 \cdot \sum_{j=n-2}^{n} e_j + \ldots + i_n \cdot \sum_{j=2}^{n} e_j$$

---

mdspan: $4 \times 3 \times 2$

$\texttt{layout\_left}(3, 0, 1) \to 3 + 0 \cdot 4 + 1 \cdot 4 \cdot 3 = 15$

0	4	8
1	5	9
2	6	10
3	7	11

12	16	20
13	17	21
14	18	22
15	19	23

`std::layout_stride`

mdspan sizes: $e_1 \times e_2 \times \ldots \times e_n$

strides: $s_1 \times s_2 \times \ldots \times s_n$

`layout_stride`$(i_1, i_2, \ldots, i_{n-1}, i_n) \rightarrow i_1 \cdot s_1 + i_2 \cdot s_2 + \ldots + i_n \cdot s_n$

---

mdspan: 3×2 (submatrix), strides: $2, 6$

`layout_stride`$(1, 2) \rightarrow 1 \cdot 2 + 2 \cdot 6 = 14, \quad 14 + 8 = 22$

0	1	2	3	4	5
6	7	**8**	9	**10**	11
12	13	**14**	15	**16**	17
18	19	**20**	21	<u>**22**</u>	23

`std::layout_right_padded`

mdspan sizes: $e_1 \times e_2$

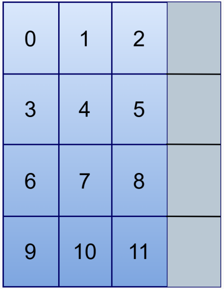
stride: S, generally $S > e_2$

`layout_right_padded`$(i_1, i_2) \rightarrow i_1 \cdot S + i_2$

---

mdspan: 4×3, stride: 4

Implication: if the value type is `float`, all rows are 16-byte aligned



`std::layout_left_padded`

mdspan sizes: $e_1 \times e_2$

stride: S, generally $S > e_1$

`layout_left_padded`$(i_1, i_2) \rightarrow i_1 + i_2 \cdot S$

---

mdspan: 3×3, stride: 4

Implication: if the value type is `float`, all columns are 16-byte aligned

0	3	7
1	4	8
2	5	9

Layout - Example 1

```
#include <mdspan> compiler explorer
#include <array>

float           array[] = {1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f, 7.0f, 8.0f};
std::dims<2>    extents{2, 4};
std::mdspan     md_right(array, std::layout_right::mapping{extents});
std::mdspan     md_left(array, std::layout_left::mapping{extents});
print(md_right); // 1, 2, 3, 4,
                 // 5, 6, 7, 8,
print(md_left);  // 1, 3, 5, 7,
                 // 2, 4, 6, 8,

std::dims<2>     extents1{2, 2};
std::array       strides{4, 2};
std::mdspan      md_stride(array, std::layout_stride::mapping{extents1, strides});
print(md_stride); // 1, 3,
                  // 5, 7,
```

Layout - Example 2

```
std::dims   extents1{2, 3};
using       layoutA = std::layout_right_padded<std::dynamic_extent>;
std::mdspan md_right_padded(array, layoutA::mapping{extents, 4});
print(md_right_padded); // 1, 3, 5,
                        // 5, 6, 7,

std::dims      extents1{1, 4};
using          layoutB = std::layout_left_padded<std::dynamic_extent>;
std::mdspan    md_left_padded(array, layoutB::mapping{extents, 4});
print(md_right_padded); // 1, 3, 5, 7,
```

Layout-related Methods

`stride(i)`	The stride of the dimension *i*, namely the number of steps in the linear index-space to move to the next element in the *ith* dimension-space
`required_span_size()`	The maximum value of the linear index (*codomain*) among all possible values of the multidimensional indices
`is_unique()`	All multidimensional indices produce a distinct linear index (*codomain*)
`is_exhaustive()`	If all *possible* values of the *codomain* can be reached, e.g. a matrix with padding is not exhaustive
`is_strided()`	If there is a fixed stride between elements of the same dimension, for all dimensions

Layout - Example 3

```
float              array[] = {1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f, 7.0f, 8.0f};
std::dims<2>       extentsA{2, 3}; // compiler explorer
using              layoutA = std::layout_right_padded<std::dynamic_extent>;
layoutA::mapping mappingA{extents, 4};
std::mdspan        mdA(array, mappingA);
cout << mdA.stride(0) << ", " << mdA.stride(1) << "\n"              // 4, 1
     << mdA.size() << ", " << mappingA.required_span_size()  << "\n" // 6, 4 * 2 = 8
     << mdA.is_exhaustive() << " "                                   // false
     << mdA.is_unique() << ", " << mdA.is_strided() << "\n";         // true, true

std::dims<2>                extentsB{2, 3};
std::layout_left::mapping mappingB{extentsB};
std::mdspan                 mdB(array, mappingB);
cout << mdB.stride(0) << ", " << mdB.stride(1) << "\n"              // 1, 3
     << mdB.size() << ", " << mappingB.required_span_size()  << "\n" // 6, 2 * 3 = 6
     << mdB.is_exhaustive() << " "                                   // true
     << mdB.is_unique() << ", " << mdB.is_strided() << "\n";         // true, true
```

The main functionality of an **accessor** is to provide a reference to an element pointed out by a 1-D linear index

```
reference access(data_handle_type p, size_t i);
```

An accessor describes how to *interpret* `mdspan` values. Some examples:

- Scaling, C++26 `std::linalg::scaled_accessor` cppref, P1673
- Conjugate, C++26 `std::linalg::conjugated_accessor` cppref, P1673
- Pointer alignment, C++26 `std::aligned_accessor` cppref, P2897
- Describe *implicit* values, avoiding explicit storage
- Prevent pointer aliasing assumptions by the compiler (`restrict`)
- Atomic operations (`std::atomic_ref`)
- and even move a *robot arm*, `CppNow'23, Spanny`

```cpp
#include <mdspan>
#include <linalg>

float array[] = {1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f};
std::dims<2>                  extents{2, 3};
std::layout_right::mapping mapping{extents};
std::mdspan                   mdA{array, mapping, std::linalg::scaled_accessor{2.0f}};
print(mdA); // 2,  4,  6,
            // 8, 10, 12,
print(std::linalg::scaled(mdA, 4.0f)); //  4,  8, 12,
                                       // 16, 20, 24,

std::mdspan mdB{array, mapping, std::aligned_accessor<16>{}};
// mdB underlying data is 16-byte aligned
```

```cpp
template<typename ElementType> // potentially other template parameters
struct simple_accessor {
    using offset_policy    = std::default_accessor<ElementType>;
    using element_type     = ElementType;
    using reference        = ElementType&;
    using data_handle_type = ElementType*;
    // constructors, conversions, etc.

    constexpr reference access(data_handle_type p, size_t i) const noexcept {
        return p[i]; // main customization point
    }
    constexpr data_handle_type
    offset(data_handle_type p, size_t i) const noexcept {
        return p + i;
    }
};
```

I/O Stream

I/O Stream

`<iostream>` input/output library refers to a family of classes and supporting functions in the C++ Standard Library that implement stream-based input/output capabilities

There are four predefined iostreams:

- `cin` standard input (`stdin`)
- `cout` standard output (`stdout`) [buffered]
- `cerr` standard error (`stderr`) [unbuffered]
- `clog` standard error (`stderr`) [buffered]

buffered: the content of the buffer is not written to disk / console until some events occur

Basic I/O Stream manipulator:

- `flush` flushes the output stream `cout << flush;`
- `endl` shortcut for `cout << "\n" << flush;`
  `cout << endl`
- `flush` and `endl` force the program to synchronize with the terminal → very slow operation!

- **Set integral representation**: default: dec
  `cout << dec << 0xF;` prints 16
  `cout << hex << 16;` prints 0xF
  `cout << oct << 8;` prints 10

- Print the underlying **bit representation** of a value:

```cpp
#include <bitset>
std::cout << std::bitset<32>(3.45f); // (32: num. of bits)
// print 01000000010111001100110011001101
```

- **Print true/false text**:
  `cout << boolalpha << 1;` prints true
  `cout << boolalpha << 0;` prints false

`<iomanip>`

- **Set decimal precision**: default: 6

  `cout << setprecision(2) << 3.538;` → 3.54

- **Set float representation**: default: `std::defaultfloat`

  `cout << setprecision(2) << fixed << 32.5;` → 32.50

  `cout << setprecision(2) << scientific << 32.5;` → 3.25e+01

- **Set alignment**: default: right

  `cout << right << setw(7) << "abc" << "##";` → `␣␣␣␣abc##`

  `cout << left << setw(7) << "abc" << "##";` → `abc␣␣␣␣##`

  (better than using tab \t)

I/O Stream - `std::cin`

`std::cin` is an example of *input* stream. Data coming from a source is read by the program. In this example `cin` is the standard input

```cpp
#include <iostream>

int main() {
    int a;
    std::cout << "Please enter an integer value:" << endl;
    std::cin >> a;

    int b;
    float c;
    std::cout << "Please enter an integer value "
              << "followed by a float value:" << endl;
    std::cin >> b >> c; // read an integer and store into "b",
}                       // then read a float value, and store
                        // into "c"
```

`ifstream`, `ofstream` are output and input stream too

`<fstream>`

- **Open a file for reading**
  Open a file in input mode: `ifstream my_file("example.txt")`
- **Open a file for writing**
  Open a file in output mode: `ofstream my_file("example.txt")`
  Open a file in append mode:
  `ofstream my_file("example.txt", ios::out | ios::app)`
- **Read a line** `getline(my_file, string)`
- **Close a file** `my_file.close()`
- **Check the stream integrity** `my_file.good()`

- **Peek the next character**

```
char current_char = my_file.peek()
```

- **Get the next character (and advance)**

```
char current_char = my_file.get()
```

- **Get the position of the current character in the input stream**

```
int byte_offset = my_file.tellg()
```

- **Set the char position in the input sequence**

  `my_file.seekg(byte_offset)` (absolute position)

  `my_file.seekg(byte_offset, position)` (relative position)

  where position can be: `ios::beg` (the begin), `ios::end` (the end), `ios::cur` (current position)

- **Ignore characters until the delimiter is found**
  `my_file.ignore(max_stream_size, <delim>)`
  e.g. skip until end of line `\n`

- **Get a pointer to the stream buffer object currently associated with the stream**
  `my_file.rdbuf()`
  can be used to redirect file stream

I/O Stream - Example 1

Open a file and print line by line:

```cpp
#include <iostream>
#include <fstream>

int main() {
  std::ifstream fin("example.txt");
  std::string str;
  while (std::getline(fin, str))
     std::cout << str << "\n";
  fin.close();
}
```

An alternative version with redirection:

```cpp
#include <iostream>
#include <fstream>

int main() {
  std::ifstream fin("example.txt");
  std::cout << fin.rdbuf();
  fin.close();
}
```

Reading files line by line in C++ using ifstream

I/O Stream - Example 2

example.txt:

```
23␣70␣␣␣44\n
\t57\t89
```

The input stream is independent from the type of space (multiple space, tab, newline `\n`, `\r\n`, etc.)

Another example:

```cpp
#include <iostream>
#include <fstream>

int main() {
    std::ifstream fin("example.txt");
    char c = fin.peek(); // c = '2'
    while (fin.good()) {
        int var;
        fin >> var;
        std::cout << var;
    }         // print 2370445789
    fin.seekg(4);
    c = fin.peek();  // c = '0'
    fin.close();
}
```

I/O Stream -Check the End of a File

- Check the current character

```
while (fin.peek() != std::char_traits<char>::eof()) // C: EOF
    fin >> var;
```

- Check if the read operation fails

```
while (fin >> var)
    ...
```

- Check if the stream past the end of the file

```
while (true) {
    fin >> var
    if (fin.eof())
        break;
}
```

I/O Stream (checkRegularType)

Check if a file is a **regular file** and can be read/written

```
#include <sys/types.h>
#include <sys/stat.h>
bool checkRegularFile(const char* file_path) {
    struct stat info;
    if (::stat( file_path, &info ) != 0)
        return false;         // unable to access
    if (info.st_mode & S_IFDIR)
        return false;                // is a directory
    std::ifstream fin(file_path); // additional checking
    if (!fin.is_open() || !fin.good())
        return false;
    try {                        // try to read
        char c; fin >> c;
    } catch (std::ios_base::failure&) {
        return false;
    }
    return true;
}
```

I/O Stream - File size

Get the **file size** in bytes in a **portable** way:

```
long long int fileSize(const char* file_path) {
    std::ifstream fin(file_path);     // open the file
    fin.seekg(0, ios::beg);           // move to the first byte
    std::istream::pos_type start_pos = fin.tellg();
                                      // get the start offset
    fin.seekg(0, ios::end);           // move to the last byte
    std::istream::pos_type end_pos = fin.tellg();
                                      // get the end offset
    return end_pos - start_pos;       // position difference
}
```

see C++17 file system utilities

Strings and `std::print`

`std::string` is a wrapper of character sequences

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

```
#include <string>

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

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

`std::string` supports `constexpr` in C++20

- `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 new substring `[start_pos, end]`
  `new_string substr(start_pos, count)`
  returns a new 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`

Conversion from/to Numeric Values

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)
- C++17 `std::from_chars(start, end, result, base)` fast string conversion (no allocation, no exception)

Converts a numeric value to a string:

- C++11 `to_string(numeric_value)` numeric value to string

Examples

```cpp
std::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    = std::stoi(str2);   // a = 34;
std::string str3 = std::to_string(a); // str3 = "34"
```

Tips

- Conversion from integer to char letter (e.g. 3 $\rightarrow$ 'C'):
  `static_cast<char>('A'+ value)`
  `value` $\in [0, 26]$ (English alphabet)

- Conversion from char to integer (e.g. 'C' $\rightarrow$ 3): `value - 'A'`
  `value` $\in [0, 26]$

- Conversion from digit to char number (e.g. 3 $\rightarrow$ '3'):
  `static_cast<char>('0'+ value)`
  `value` $\in [0, 9]$

- char to string `std::string(1, char_value)`

C++17 `std::string_view` describes a minimum common interface to interact with string data:

- `const std::string&`
- `const char*`

The purpose of `std::string_view` is to avoid copying data which is already owned by the original object

```cpp
#include <string>
#include <string_view>

std::string str  = "abc";  // new memory allocation + copy
std::string_view = "abc";  // only the reference
```

`std::string_view` provides similar functionalities of `std::string`

```cpp
#include <iostream>
#include <string>
#include <string_view>

void string_op1(const std::string& str) {}
void string_op2(std::string_view   str) {}


string_op1("abcdef"); // allocation + copy
string_op2("abcdef"); // reference

const char* str1 = "abcdef";
std::string str2(str1);          // allocation + copy
std::cout << str2.substr(0, 3); // print "abc"

std::string_view str3(str1);     // reference
std::cout << str3.substr(0, 3); // print "abc"
```

`std::string_view` supports `constexpr` constructor and methods

```
constexpr std::string_view str1("abc");
constexpr std::string_view str2 = "abc";

constexpr char c = str1[0];         // 'a'
constexpr bool b = (str1 == str2); // 'true'

constexpr int size = str1.size();                  // '3'
constexpr std::string_view str3 = str1.substr(0, 2); // "ab"

constexpr int pos = str1.find("bc");               // '1'
```

`printf` *functions*: no automatic type deduction, error prone, not extensible

`stream` *objects*: very verbose, hard to optimize

C++20 `std::format` provides python style formatting:

- Type-safe
- Support positional arguments
- Extensible (support user-defined types)
- Return a `std::string`

`std::format` - Example

Integer formatting

```
std::format("{}", 3);   // "3"
std::format("{:b}", 3); // "101"
```

Floating point formatting

```
std::format("{:.1f}", 3.273); // "3.1"
```

Alignment

```
std::format("{:>6}", 3.27); // "  3.27"
std::format("{:<6}", 3.27); // "3.27  "
```

Argument reordering

```
std::format("{1} - {0}", 1, 3); // "3 - 1"
```

`std::print`

C++23 introduces `std::print()` `std::println()`

```
std::print("Hello, {}!\n", name);

std::println("Hello, {}!", name);  // prints a newline
```

Math Libraries

`<cmath>`

- `abs(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 non-decimal value not less than the given value, $\lceil x \rceil$
- `floor(x)` nearest non-decimal value not greater than the given value, $\lfloor x \rfloor$
- `round(x)` rounding to the nearest non-decimal value halfway cases away from zero

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

<limits> Numerical Limits

Get numeric limits of a given type:

`<limits>` C++11

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

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

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

`<numbers>` Mathematical Constants

`<numbers>` C++20

The header provides numeric constants

- `e` Euler number e
- `pi` π
- `phi` Golden ratio $\frac{1+\sqrt{5}}{2}$
- `sqrt2` $\sqrt{2}$

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
```

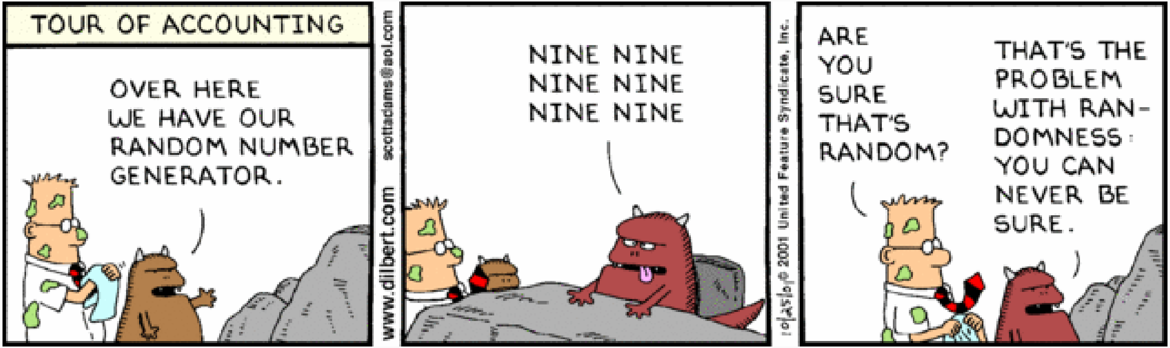
- **Rounded Division:** $\left\lfloor \frac{\texttt{value}}{\texttt{div}} + \frac{1}{2} \right\rfloor$

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

Note: do not use floating-point conversion (see `Basic Concept I`)

Random Number

Random Number



"Random numbers should not be generated with a method chosen at random"
— Donald E. Knuth

Applications: cryptography, simulations (e.g. Monte Carlo), etc.

Random Number

see Lavarand

Basic Concepts

- A **pseudorandom (PRNG)** *sequence of numbers* satisfies most of the statistical properties of a truly random sequence but is generated by a *deterministic* algorithm (deterministic finite-state machine)
- A **quasirandom** *sequence of n-dimensional points* is generated by a *deterministic* algorithm designed to fill an n-dimensional space evenly
- The **state** of a PRNG describes the status of the generator (the values of its variables), namely where the system is after a certain amount of transitions
- The **seed** is a value that initializes the *starting state* of a PRNG. The same seed always produces the same sequence of results
- The **offset** of a sequence is used to skip ahead in the sequence
- PRNGs produce **uniformly distributed** values. PRNGs can also generate values according to a probability function (binomial, normal, etc.)

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** (optional)

```
<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>

int main() {
    std::random_device                 rd;
    std::default_random_engine         generator{rd{}};
    std::uniform_int_distribution<int> distribution{0, 9};

    std::cout << distribution(generator);  // first random number
    std::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 could be selected randomly by using the current time:

```
#include <random>
#include <chrono>

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

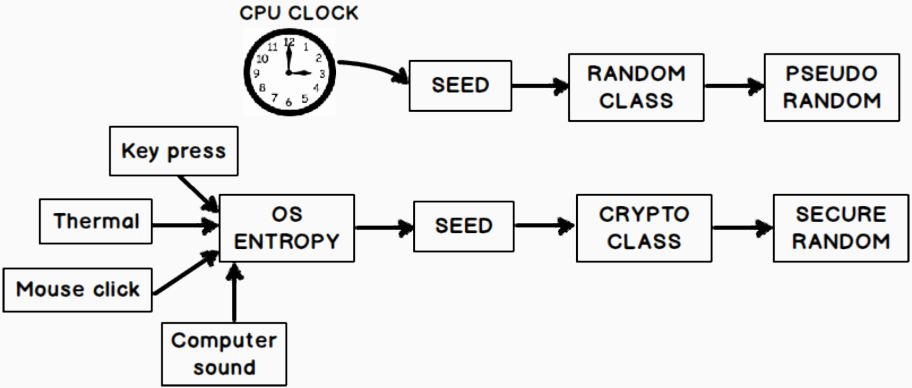
`chrono::system_clock::now()` returns an object representing the current point in time

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

Problem: Consecutive calls return *very similar* seeds

Pseuso seed: easy to guess, e.g. single source of randomness

Secure seed: hard to guess, e.g. multiple sources of randomness



How do I generate a random integer in C#?

A **random device** `std::random_device` is a uniformly distributed integer generator that produces non-deterministic random numbers, e.g. from a hardware device such as `/dev/urandom`

```
#include <random>

std::random_device                    rnd_device;
std::default_random_engine generator{rnd_device()};
```

Note: Not all OSs provide a random device `std::seed_seq` consumes a sequence of integer-valued data and produces a number of unsigned integer values in the range $[0, 2^{32} - 1]$. The produced values are distributed over the entire 32-bit range even if the consumed values are close

```
#include <random>
#include <chrono>

unsigned seed1 = std::chrono::system_clock::now()
                .time_since_epoch().count();
unsigned seed2 = seed1 + 1000;

std::seed_seq              seq1{seed1, seed2};
std::default_random_engine generator1{seq};
```

PRNG Period and Quality

PRNG Period

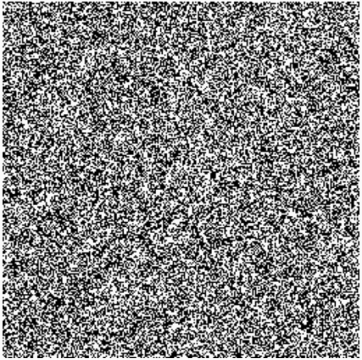
The **period** (or **cycle length**) of a PRNG is the length of the sequence of numbers that the PRNG generates before repeating

PRNG Quality

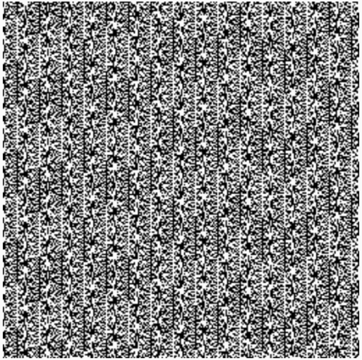
(*informal*) If it is hard to distinguish a generator output from *truly* random sequences, we call it a **high quality** generator. Otherwise, we call it **low quality** generator

Generator	Quality	Period	Randomness
Linear Congruential	Poor	$2^{31} \approx 10^9$	Statistical tests
Mersenne Twister 32/64-bit	High	10^{6000}	Statistical tests
Subtract-with-carry 24/48-bit	Highest	10^{171}	Mathematically proven

Randomness Quality



RANDOM.ORG



PHP rand() on Microsoft Windows

- `On C++ Random Number Generator Quality`
- `It is high time we let go of the Mersenne Twister`

Random Engines

- **Linear congruential (LF)**
  The simplest generator engine. Modulo-based algorithm:
  $x_{i+1} = (\alpha x_i + c) mod\ m$ where α, c, m are implementation defined
  C++ Generators: `std::minstd_rand`, `std::minstd_rand0`, `std::knuth_b`

- **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)
  C++ Generators: `std::mt19937`, `std::mt19937_64`

- **Subtract-with-carry (LF)** *(G. Marsaglia and A. Zaman, 1991)*
  Pseudo-random generation based on Lagged Fibonacci algorithm (used for example by physicists at CERN)
  C++ Generators: `std::ranlux24_base`, `std::ranlux48_base`, `std::ranlux24`, `std::ranlux48`

Statistical Tests

The table shows after how many iterations the generator fails the statistical tests

Generator	256M	512M	1G	2G	4G	8G	16G	32G	64G	128G	256G	512G	1T
`ranlux24_base`	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
`ranlux48_base`	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
`minstd_rand`	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
`minstd_rand0`	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
`knuth_b`	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
`mt19937`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗
`mt19937_64`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗
`ranlux24`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`ranlux48`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Space and Performance

Generator	Predictability	State	Performance
Linear Congruential	Trivial	`4-8 B`	**Fast**
Knuth	Trivial	`1 KB`	**Fast**
Mersenne Twister	Trivial	`2 KB`	**Good**
randlux_base	Trivial	`8-16 B`	**Slow**
randlux	Unknown?	$\sim$`120 B`	**Super slow**

Distribution

- **Uniform distribution**

  `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 = ...

// Original linear congruential
minstd_rand0  lc1_generator(seed);
// Linear congruential (better tuning)
minstd_rand   lc2_generator(seed);
// Standard mersenne twister (64-bit)
mt19937_64    mt64_generator(seed);
// Subtract-with-carry (48-bit)
ranlux48_base swc48_generator(seed);

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);
```

Recent Algorithms and Performance

Recent algorithms:

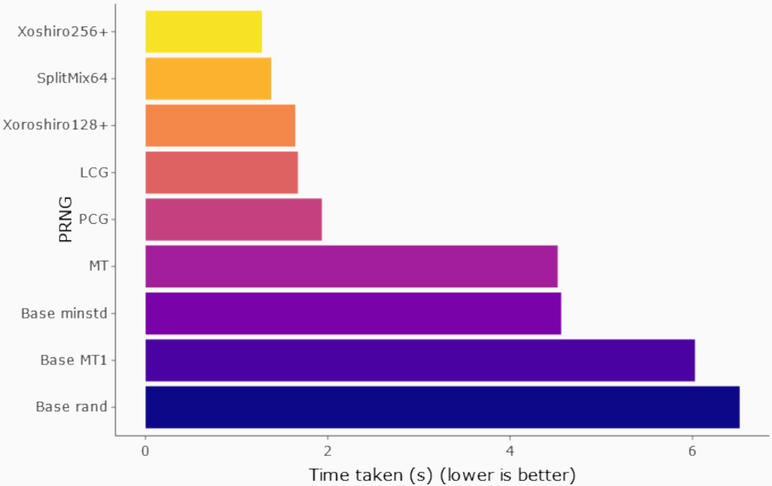
- `PCG, A Family of Better Random Number Generators`
- `Xoshiro / Xoroshiro generators and the PRNG shootout`
- `The Xorshift128+ random number generator fails BigCrush`
- `RapidHash`
- `DualMix128: A Fast and Simple C PRNG`

Parallel algorithms:

- `Squares: A Fast Counter-Based RNG`
- `Parallel Random Numbers: As Easy as 1, 2, 3` (Philox)
- `OpenRNG: ARM Random Number Generator Library`

If strong random number quality properties are not needed, it is possible to generate a random permutation of integer values (with period of 2^{32}) in a very efficient way by using hashing functions, see `Hash Function Prospector`

Performance Comparison

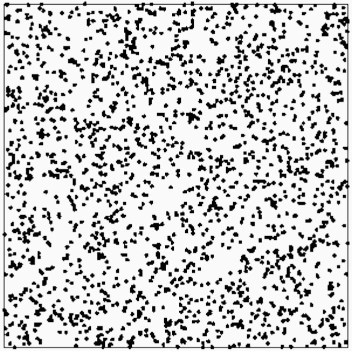


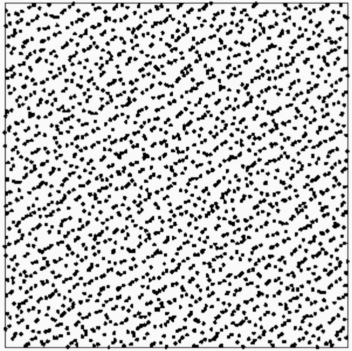
Random number generators for C++ performance tested

The **quasi-random** numbers have the low-discrepancy property that is a measure of *uniformity for the distribution* of the point for the multi-dimensional case

- Quasi-random sequence, in comparison to pseudo-random sequence, distributes evenly, namely this leads to spread the number over the entire region
- The concept of low-discrepancy is associated with the property that the successive numbers are added in a position as away as possible from the other numbers that is, avoiding *clustering* (grouping of numbers close to each other)

Pseudo-random vs. Quasi random





True Random Number Generator (TRNG)

Most modern CPUs provide **True Random Number Generator (TRNG)**. These generators use physical processes, such as electrical or thermal noise, to produce *truly non-deterministic* random numbers.

The hardware provides two different functionalities, one to produce the *seed* for a pseudorandom number generator, and another to cryptographically (securely) generate *random number sequences*. Hardware `TRNG` are typically slower than software `PRNGs`.

TRNG support:

- Query: Instructions `rdseed` and `rdrand` on Intel/AMD and `RNDR` on Arm in `/proc/cpuinfo`.
- Linux provides such functionalities via `/dev/random` and `/dev/urandom`.
- `_rdrand*()` functions in `immintrin.h`, `Intel Intrinsics Guide`

Time Measuring

Wall-Clock/Real time

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

User/CPU time

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

System time

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

Time Measuring

The *Wall-clock time* measured on a concurrent process platform may include the time elapsed for other tasks

The *User/CPU time* of a multi-thread program is the sum of the execution time of all threads

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()` : time resolution $1\mu s$

```
#include <time.h>      //struct timeval
#include <sys/time.h> //gettimeofday()

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: Linux only (not portable), the time is not monotonic increasing (timezone), time resolution is big `std::chrono` C++11

```
#include <chrono>
auto start_time = std::chrono::system_clock::now();
... // code
auto end_time   = std::chrono::system_clock::now();

std::chrono::duration<double> diff = end_time - start_time;
cout << "Elapsed: " << diff.count(); // in seconds
cout << std::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.

`steady_clock` is implemented over `clock_gettime` on POSIX system and has 1*ns* time resolution

```
#include <chrono>
auto start_time = std::chrono::steady_clock::now();
... // code
auto end_time   = std::chrono::steady_clock::now();
```

However, the overhead of C++ API is not always negligible, e.g.
`Linux libstdc++` → 20ns, `Mac libc++` → 41ns

Time Measuring - User Time

`std::clock`, implemented over `clock_gettime` on POSIX system and has 1*ns* time resolution

```
#include <chrono>

clock_t start_time = std::clock();
... // code
clock_t end_time   = std::clock();

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

Time Measuring - User/System Time

```cpp
#include <sys/times.h>

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
```

Other Standard Library Classes

`std::pair` (`<utility>`) class couples together a pair of values, which may be of different types

Construct a `std::pair`

- `std::pair pair1(value1, value2)`, C++17 CTAD
- `std::pair<T1, T2> pair2(value1, value2)`
- `std::pair<T1, T2> pair3 = {value1, value2}`
- `auto pair4 = std::make_pair(value1, value2)`

Data members:

- `first` access first field
- `second` access second field

Methods:

- comparison `==`, `<`, `>`, $\geq$, $\leq$
- swap `std::swap`

```cpp
#include <utility>

std::pair<int, std::string> pair1(3, "abc");
std::pair<int, std::string> pair2 = { 4, "zzz" };
auto                        pair3 = std::make_pair(3, "hgt");

cout << pair1.first;  // print 3
cout << pair1.second; // print "abc"

std::swap(pair1, pair2);
cout << pair2.first;  // print "zzz"
cout << pair2.second; // print 4

cout << (pair1 > pair2); // print 1
```

Note: `std::pair` is not trivially copyable `std::tuple` (`<tuple>`) is a fixed-size collection of heterogeneous values. It is a generalization of `std::pair`. It allows any number of values

Construct a `std::tuple` of size 3

```
#include <tuple>
std::tuple             tuple1(value1, value2, value3); // C++17 CTAD
std::tuple<T1, T2, T3> tuple2(value1, value2, value3);
std::tuple<T1, T2, T3> tuple3 = {value1, value2, value3};
auto                   tuple4 = std::make_tuple(value1, value2, value3);
```

Get data members

```
std::get<I>(tuple);    // returns the I-th value of the tuple
std::get<type>(tuple); // returns the tuple element with given type
                       // (compiles only if that type is unique)
```

Other methods: comparison `==, <, >,` $\geq$, $\leq$, swap `std::swap`

`std::tuple` 2/3

- `auto t3 = std::tuple_cat(t1, t2)`
  concatenate two tuples

- `const int size = std::tuple_size<TupleT>::value`
  returns the number of elements in a tuple at compile-time

- `using T = typename std::tuple_element<I, TupleT>::type` obtains the type of the specified element

- `std::tie(value1, value2, value3) = tuple`
  creates a tuple of references to its arguments

- `std::ignore`
  an object of unspecified type such that any value can be assigned to it with no effect

```cpp
#include <tuple>
std::tuple<int, float, char> f() { return {7, 0.1f, 'a'}; }

std::tuple<int, char, float> tuple1(3, 'c', 2.2f);
auto tuple2 = std::make_tuple(2, 'd', 1.5f);

cout << std::get<0>(tuple1); // print 3
cout << std::get<1>(tuple1); // print 'c'
cout << std::get<2>(tuple1); // print 2.2f
cout << (tuple1 > tuple2);   // print true

auto concat = std::tuple_cat(tuple1, tuple2);
cout << std::tuple_size<decltype(concat)>::value; // print 6

using T = std::tuple_element<4, decltype(concat)>::type; // T is int
int value1; float value2;
std::tie(value1, value2, std::ignore) = f();
```

`<variant>` C++17

`std::variant` represents a **type-safe union** as the corresponding objects know which type is currently being held

It can be indexed by:

- `std::get<index>(variant)` an integer
- `std::get<type>(variant)` a type

```
#include <variant>

std::variant<int, float, bool> v(3.3f);
auto  x = std::get<1>(v);     // return 3.3f
auto y = std::get<float>(v); // return 3.3f
// std::get<0>(v);            // member 0 is not active, run-time exception!!
```

Another useful method is `index()` which returns the position of the type currently held by the variant

```
#include <variant>

std::variant<int, float, bool> v(3.3f);

cout << v.index(); // return 1

v = true;          // not 'v' holds a bool
cout << v.index(); // return 2
```

It is also possible to query the index at run-time depending on the type currently being held by providing a **visitor**

```cpp
#include <variant>

struct Visitor {
    void operator()(int& value)   { value *= 2; }

    void operator()(float& value) { value += 3.0f; } // <--

    void operator()(bool& value)  { value = true; }
};

std::variant<int, float, bool> v(3.3f);

std::visit(v, Visitor{});

cout << std::get<float>(v); // 6.3f
```

`<optional>` C++17

`std::optional` provides facilities to represent potential "no value" states

As an example, it can be used for representing the state when an element is not found in a set

```
#include <optional>

std::optional<int> find(const std::vector<int>& vector, int value_to_search) {
    for (int i = 0; i < vector.size(); i++) {
        if (vector[i] == value_to_search)
            return i;
    }
    return {}; // std::nullopt;
}
```

std::optional 2/2

```
#include <optional>

char set[] = "sdfslgfsdg";
auto x     = find(set, 'a'); // 'a' is not present
if (!x)
    cout << "not found";
if (!x.has_value())
    cout << "not found";

auto y = find(set, 'l');
cout << *y << " " << y.value(); // print '4' '4'

x.value_or('A'); // returns 'A'
y.value_or('A'); // returns 'A'
```

`<any>` C++17

`std::any` holds arbitrary values and provides **type-safety**

```
#include <any>

std::any var = 1;           // int
cout << var.type().name(); // print 'i'

cout << std::any_cast<int>(var);
// cout << std::any_cast<float>(var); // exception!!

var = 3.14; // double
cout << std::any_cast<double>(var);

var.reset();
cout << var.has_value(); // print 'false'
```

Filesystem Library

Filesystem Library

C++17 introduces abstractions and facilities for performing operations on file systems and their components, such as **paths**, **files**, and **directories**

- Follow the Boost filesystem library
- Based on POSIX
- Fully-supported from `clang 7`, `gcc 8`, etc.
- Work on Windows, Linux, Android, etc.

Basic concepts

- **file**: a file system object that holds data
  - **directory** a container of directory entries
  - **hard link** associates a name with an existing file
  - **symbolic link** associates a name with a path
  - **regular file** a file that is not one of the other file types
- **file name**: a string of characters that names a file. Names `.` (dot) and `..` (dot-dot) have special meaning at library level
- **path**: sequence of elements that identifies a file
  - **absolute path**: a path that unambiguously identifies the location of a file
  - **canonical path**: an absolute path that includes no symlinks, `.` or `..` elements
  - **relative path**: a path that identifies a file relative to some location on the file system

path Object

A `path` object stores the pathname in native form

```
#include <filesystem> // required
namespace fs = std::filesystem;

fs::path p1 = "/usr/lib/sendmail.cf"; // portable format
fs::path p2 = "C:\\users\\abcdef\\";  // native format on Windows

cout << "p1: " << p1;           // /usr/lib/sendmail.cf
cout << "p2: " << p2;           // C:\users\abcdef\

out << "p3: " << p2 / "xyz\\";  // C:\users\abcdef\xyz\
```

`path` Methods

Decomposition (member) methods:

- **Return root-name of the path**
  `root_name()`
- **Return path relative to the root path**
  `relative_path()`
- **Return the path of the parent path**
  `parent_path()`
- **Return the filename path component**
  `filename()`
- **Return the file extension path component**
  `extension()`

Filesystem Methods - Query

- Check if a file or path exists
  `exists(path)`
- Return the file size
  `file_size(path)`
- Check if a file is a directory
  `is_directory(path)`
- Check if a file (or directory) is empty
  `is_empty(path)`
- Check if a file is a regular file
  `is_regular_file(path)`
- Returns the current path
  `current_path()`

Directory Iterators

Iterate over files of a directory (recursively/non-recursively)

```
#include <filesystem>

namespace fs = std::filesystem;

for(auto& path : fs::directory_iterator("/usr/tmp/"))
    cout << path << '\n';

for(auto& path : fs::recursive_directory_iterator("/usr/tmp/"))
    cout << path << '\n';
```

Filesystem Methods - Modify

- Copy files or directories
  `copy(path1, path2)`

- Copy files
  `copy_file(src_path, src_path, [fs::copy_options::recursive])`

- Create new directory
  `create_directory(path)`

- Remove a file or empty directory
  `remove(path)`

- Remove a file or directory and all its contents, recursively
  `remove_all(path)`

- Rename a file or directory
  `rename(old_path, new_path)`

Examples

```cpp
#include <filesystem> // required
namespace fs = std::filesystem;
fs::path p1 = "/usr/tmp/my_file.txt";

cout << fs::exists(p1);            // true
cout << p1.parent_path(p1);        // "/usr/tmp/"
cout << p1.filename();             // "my_file.txt"
cout << p1.stem();                 // "my_file"
cout << p1.extension();            // "txt"
cout << fs::is_directory(p1);      // false
cout << fs::is_regular_file(p1); // true

fs::create_directory("/my_dir/");
fs::copy(p1.parent_path(), "/my_dir/", fs::copy_options::recursive);
fs::copy_file(p1, "/my_dir/my_file2.txt");
fs::remove(p1);
fs::remove_all(p1.parent_path());
```