Cheatography

Modern C++ 17 Standard Library Features Cheat Sheet

by NexWebSites.com (NexWebSites) via cheatography.com/57483/cs/24614/

std::variant

```
std::variant<int, double> v
{12};
std::get<int>(v); // == 12
std::get<0>(v); // == 12
v = 12.0;
std::get<double>(v); // == 12.0
std::get<1>(v); // == 12.0
```

The class template std::variant represents a type-safe union. An instance of std::variant at any given time holds a value of one of its alternative types (it's also possible for it to be valueless).

std::optional

```
std::optional<std::string>
create(bool b) {
  if (b) {
    return "Wonder Woman";
  } else {
    return {};
  }
}
create(false).value_or("empty");
// == "empty"
create(true).value(); // == "-
Wonder Woman"
// optional-returning factory
functions are usable as
conditions of while and if
if (auto str = create(true)) {
  // ...
}
```

The class template std::optional manages an optional contained value, i.e. a value that may or may not be present. A common use case for optional is the return value of a function that may fail.

std::any

```
std::any x {5};
x.has_value() // == true
std::any_cast<int>(x) // == 5
std::any_cast<int&>(x) = 10;
std::any_cast<int>(x) // == 10
```

A type-safe container for single values of any type.

std::string_view

```
// Regular strings.
std::string_view cppstr {"foo"};
// Wide strings.
std::wstring_view wcstr_v {L"b-
az"};
// Character arrays.
char array[3] = {'b', 'a', 'r'};
std::string_view array_v(array,
std::size(array));
std::string str {" trim me"};
std::string_view v {str};
v.remove_prefix(std::min(v.fin-
d_first_not_of(" "), v.size()));
str; // == " trim me"
v; // == "trim me"
```

A non-owning reference to a string. Useful for providing an abstraction on top of strings (e.g. for parsing).

Parallel algorithms

```
std::vector<int> longVector;
// Find element using parallel
execution policy
auto result1 = std::find(std::-
execution::par, std::begin(long-
Vector), std::end(longVector),
2);
```

Parallel algorithms (cont)

```
// Sort elements using
sequential execution policy
auto result2 = std::sort(std::-
execution::seq, std::begin(long-
Vector), std::end(longVector));
```

Many of the STL algorithms, such as the copy, find and sort methods, started to support the parallel execution policies: seq, par and par_unseq which translate to "sequentially", "parallel" and "parallel unsequenced".

std::invoke

```
template <typename Callable>
class Proxy {
  Callable c;
public:
  Proxy(Callable c): c(c) {}
  template <class... Args>
  decltype(auto) operator()(A-
rgs&&... args) {
    // ...
    return std::invoke(c,
std::forward<Args>(args)...);
  }
};
auto add = [](int x, int y) {
  return x + y;
};
Proxy<decltype(add)> p {add};
p(1, 2); // == 3
```

Invoke a Callable object with parameters. Examples of Callable objects are std::function or std::bind where an object can be called similarly to a regular function.

Modern C++ 17 Standard Library Features Cheat Sheet

by NexWebSites.com (NexWebSites) via cheatography.com/57483/cs/24614/

std::apply

```
auto add = [](int x, int y) {
  return x + y;
};
std::apply(add, std::make_tuple(1, 2)); // == 3
```

Invoke a Callable object with a tuple of arguments.

std::filesystem

```
const auto bigFilePath {"bigFileToCopy"};
if (std::filesystem::exists(bigFilePath)) {
  const auto bigFileSize {std::filesystem::file_size(bigFilePath)};
  std::filesystem::path tmpPath {"/tmp"};
  if (std::filesystem::space(tmpPath).available > bigFileSize) {
    std::filesystem::create_directory(tmpPath.append("example"));
    std::filesystem::copy_file(bigFilePath, tmpPath.append("newFile"));
  }
}
```

The new std::filesystem library provides a standard way to manipulate files, directories, and paths in a filesystem.

Here, a big file is copied to a temporary path if there is available space.

std::variant

```
std::variant<int, double> v {12};
std::get<int>(v); // == 12
std::get<0>(v); // == 12
v = 12.0;
```

std::variant (cont)

```
std::get<double>(v); // == 12.0
std::get<1>(v); // == 12.0
```

The class template std::variant represents a type-safe union. An instance of std::variant at any given time holds a value of one of its alternative types (it's also possible for it to be valueless).

std::byte

```
std::byte a {0};
std::byte b {0xFF};
int i = std::to_integer<int>(b);
// 0xFF
std::byte c = a & b;
int j = std::to_integer<int>(c);
// 0
```

The new std::byte type provides a standard way of representing data as a byte. Benefits of using std::byte over char or unsigned char is that it is not a character type, and is also not an arithmetic type; while the only operator overloads available are bitwise operations.

Note that std::byte is simply an enum, and braced initialization of enums become possible thanks to direct-list-initialization of enums.

Splicing for maps and sets

```
// Moving elements from one map to another:
std::map<int, string> src {{1, "one"}, {2, "two"}, {3, "buckle my shoe"}};
std::map<int, string> dst {{3, "three"}};
dst.insert(src.extract(src.find(1))); // Cheap remove and insert of { 1, "one" } from src to dst.
```

Splicing for maps and sets (cont)

```
dst.insert(src.extract(2)); // Cheap remove and insert of { 2, "two" } from src to dst.
// dst == { { 1, "one" }, { 2, "two" }, { 3, "three" } };
// Inserting an entire set:
std::set<int> src {1, 3, 5};
std::set<int> dst {2, 4, 5};
dst.merge(src);
// src == { 5 }
// dst == { 1, 2, 3, 4, 5 }
// Inserting elements which outlive the container:
auto elementFactory() {
  std::set<...> s;
  s.emplace(...);
  return s.extract(s.begin());
}
s2.insert(elementFactory());
// Changing the key of a map element:
std::map<int, string> m {{1, "one"}, {2, "two"}, {3, "three"}};
auto e = m.extract(2);
e.key() = 4;
m.insert(std::move(e));
// m == { { 1, "one" }, { 3, "three" }, { 4, "two" } }
```

Moving nodes and merging containers without the overhead of expensive copies, moves, or heap allocations/deallocations.

By **NexWebSites.com** (NexWebSites)

cheatography.com/nexwebsites/
nexwebsites.com

Published 3rd October, 2020.
Last updated 3rd October, 2020.