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"baz"};
// Character arrays.
char array[3] = {'b', 'a', 'r'};
std::string_view array_v(array, std::size(ar-
ray));
std::string str {" trim me"};
std::string_view v {str};
v.remove_prefix(std::min(v.find_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(longVector), std::end(longVector),
2);
// Sort elements using sequential execution
policy
auto result2 = std::sort(std::execution::seq,
std::begin(longVector), 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 <ty
class Proxy
  Callable c
public:
  Proxy(Call
  template <
  decltype(a
&&... args)
    // ...
    return s
orward<Args>
  }
};
auto add =
  return x +
};
Proxy<declty
p(1, 2); //
```

Invoke a Callab
Examples of Ca
or std::bind whe
similarly to a reg

std::apply

```
auto add =
  return x +
};
std::apply(a
2)); // == 3
```

Invoke a Callab
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::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 an

```
// Moving element
another:
std::map<int, str
{2, "two"}, {3, '
std::map<int, str
ee"}};
dst.insert(src.ex
dst.insert(src.ex
// dst == { { 1,
{ 3, "three" } };
// Inserting an e
std::set<int> src
std::set<int> dst
dst.merge(src);
// src == { 5 }
// dst == { 1, 2,
// Inserting elem
container:
auto elementFacto
  std::set<...> s
  s.emplace(...);
  return s.extrac
}
s2.insert(element
// Changing the k
std::map<int, str
"two"}, {3, "thre
auto e = m.extrac
e.key() = 5;
m.insert(std::mov
// m == { { 1, "c
{ 5, "two" } }
```

Moving nodes and mer
overhead of expensive
tions/deallocations.