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::g et< int >(v); // == 12
std::g et< 0>(v); // == 12
v = 12.0;
std::g et< dou ble >(v); // ==
12.0
std::g et< 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 (fa lse ).v alu e_o -
r("e mpt y"); // == " emp ty"
create (tr ue).va lue(); // ==
" Wonder Woman"
// option al- ret urning 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::a ny_ cas t<i nt>(x) // ==
5
std::a ny_ cas t<i nt&>(x) = 10;
std::a ny_ cas t<i nt>(x) // ==
10
```

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

std::string_view

```
// Regular strings.
std::s tri ng_view cppstr {"fo -
o"};
// Wide strings.
std::w str ing _view wcstr_v
{L"b az"};
// Character arrays.
char array[3] = {'b', 'a', 'r'};
std::s tri ng_view array_ v(a -
rray, std::s ize (ar ray));
std::s tring str {" trim me"};
std::s tri ng_view v {str};
v.remo ve_ pre fix (st d:: min -
(v.f in d_f irs t_n ot_ 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::f ind (st -
d:: exe cut ion ::par, std::b -
egi n(l ong Vec tor), std::e -
nd( lon gVe ctor), 2);
// Sort elements using
sequential execution policy
auto result2 = std::s ort (st -
d:: exe cut ion ::seq, std::b -
egi n(l ong Vec tor), std::e -
nd( lon gVe ctor));
```

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 {
    Cal lable c;
public:
    Pro xy( Cal lable c): c(c)
{}
    tem plate <cl ass... Args>
    dec lty pe( auto) operat -
or( )(A rgs &&... args) {
       // ...
        return std::i nvo ke(c,
std::f orw ard <Ar gs> (ar -
gs)...);
   }
};
auto add = [](int x, int y) {
    return x + y;
```


Sponsored by ApolloPad.com
Everyone has a novel in them. Finish Yours!
https://apollopad.com

std::invoke (cont)

```
> };
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.

std::apply

```
auto add = [](int x, int y) {
    return x + y;
};
std::a ppl y(add, std::m ake -
_tu ple(1, 2)); // == 3
```

Invoke a Callable object with a tuple of arguments.

std::filesystem

```
const auto bigFilePath
{"bigFileToCopy"};
if (std:: fil esy ste m:: exi -
sts (bi gFi leP ath)) {
    const auto bigFil eSize
{std:: fil esy ste m:: fil e_s -
ize (bi gFi leP ath)};
    std ::f ile sys tem ::path
tmpPath {"/t mp"};
   if (std:: fil esy ste m:: -
spa ce( tmp Pat h).a va ilable >
bigFil eSize) {
        std ::f ile sys -
tem ::c rea te_ dir ect ory -
(tm pPa th.a pp end ("ex amp -
le"));
        std ::f ile sys -
tem ::c opy _fi le( big Fil -
ePath, tmpPat h.a ppe nd( " -
new Fil e"));
```

std::filesystem (cont)

```
> }
}
```

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::t o_i nte ger <in -
t>(b); // 0xFF
std::byte c = a & b;
int j = std::t o_i nte ger <in -
t>(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::m ap<int, string> src {{1,
" one "}, {2, " two "}, {3, " -
Thr ee"}};
std::m ap<int, string> dst {{3,
" thr ee"}};
```

Splicing for maps and sets (cont)

```
> dst.insert(src.extract(src.find(1)));
dst.insert(src.extract(2));
// dst == { { 1, "one" }, { 2, "two" }, { 3, "thr-
ee" } };
// 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() = 5;
m.insert(std::move(e));
// m == { { 1, "one" }, { 3, "three" }, { 5, "-
two" } }
```

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

Sponsored by **ApolloPad.com**
Everyone has a novel in them. Finish Yours!
https://apollopad.com