Other C++11/14/17 features

- C++11

  - Auto, decltype

  - Range for

  - Enum class

  - Initializer list

  - Default and delete functions

  - User defined literals

- C++17

  - Structured binding

  - Nested namespaces

  - Compile-time static if

  - Attributes

Auto, decltype

```cpp
void f(T&& t)
{
    auto a = t;                // type deduction
    const auto& car = t;       // type deduction + modifiers
    auto&& pf = t;             // universal reference
    auto           i = 1;      // int
    decltype(auto) j = i;      // int      decltype(i)
    decltype(auto) k = (i);    // int&     decltype((i))
}
void g()
{
    auto i = 1 , d = 3.14;     // error, deduced types must match
    auto i = 1 , *p = &i;      // ok, deduced type is int
}
template <typename F, typename... Args> // perfect forwarding requires decltype(auto)
decltype(auto) forwarding( F fun, Args&&... args)
{
    return fun(std::forward<Args>(args)... );
}
template <auto n>                  // since C++17
auto f() -> std::pair<decltype(n), decltype(n)>  // auto can't deduce from { init-list }
{
    return { n, n};
}
```

Range for

```cpp
void f(const vector<double>& v)
{
  for (auto x : v) cout << x << '\n';
  for (auto& x : v) ++x;   // using reference allows us to change value
}

// You can read that as "for all x in v" going through starting with
// v.begin() and iterating to v.end(). Another example:

for (const auto x : { 1,2,3,5,8,13,21,34 }) cout << x << '\n';

// The begin() (and end()) can be a member to be called v.begin()
// or a free-standing function to be called begin(v).
```

Enum classes (C++11)

```cpp
enum Alert { green, yellow, election, red }; // traditional enum

enum class Color { red, blue };    // scoped and strongly typed enum
                                   // no export of enumerator names into enclosing scope
                                   // no implicit conversion to int
enum class TrafficLight { red, yellow, green };

Alert a = 7;                  // error (as ever in C++)
Alert a1 = static_cast<Alert>(i); // ok, if i in 0..3
Color c = 7;                  // error: no int->Color conversion
Color c1 = Color::blue;    // ok
int i2 = red;                 // ok: Alert->int conversion
int i3 = Alert::red;       // error in C++98; ok in C++0x
int i4 = blue;                // error: blue not in scope
int i5 = Color::blue;      // error: not Color->int conversion
int i6 = static_cast<int>(Color::blue)  // ok, 1

enum class Color : char { red, blue, }; // compact representation, C++11 allows extra ,

enum struct TrafficLight { red, yellow, green };// by default, the underlying type is int
enum E { E1 = 1, E2 = 2, Ebig = 0xFFFFFFF0U }; // how big is an E?

enum class EE : unsigned long { EE1 = 1, EE2 = 2, EEbig = 0xFFFFFFF0U }; // we are specific

enum class Color_code : char;       // (forward) declaration
void foobar(Color_code* p);         // use of forward declaration

enum class Color_code : char { red, yellow, green, blue }; // definition
```

Initializer list

```cpp
vector<double> v = { 1, 2, 3.456, 99.99 };

list<pair<string,string>> languages = {
                                {"Nygaard","Simula"},
                                {"Richards","BCPL"},
                                {"Ritchie","C"}
};

map<vector<string>,vector<int>> years = {
  { {"Maurice","Vincent","Wilkes"},{1913, 1945, 1951, 1967, 2000} },
  { {"Martin","Ritchards"}, {1982, 2003, 2007} },
  { {"David","John","Wheeler"}, {1927, 1947, 1951, 2004} }
};



auto x1 = 5;      // deduced type is int
auto x2(5);       // deduced type is int
auto x3{ 5 };     // deduced type is int since C++17
auto x4 = { 5 };  // deduced type is std::initializer_list
```

Default and delete functions

```
struct X
{
  // ...
  X& operator=(const X&) = delete;      // disallow copying
  X(const X&) = delete;  // delete must be at the first declaration
};

// Conversely, we can also say explicitly that
// we want to default copy behavior:
struct Y
{
  // ...
  Y& operator=(const Y&&) = default;      // default move semantics
  Y(const Y&&) = default;
};

const char *func()
{
  // static const char __fun__ = "func";
  return __fun__;
}
```

User defined literals

```
void f()
{
    int i = 12, j = 014, k = 0xC, l = 0b1100; // literals
    complex<double> cd = 2 + 3i;    // looking for operator"" I (also if, il)
    auto dur = 3h+5min+25s+567ms+765us+10ns;   // chrono duration
    "Hello"s         // basic_string, since C++14
    "Hello"sv        // string_view, since C++17
}

// identifiers not starting with underscore are reserved for std::
constexpr complex<double> operator "" i(long double d)  // imaginary literal
{
    return {0,d}; // complex is a literal type
}

long double  operator "" _w(long double);
std::string  operator "" _w(const char16_t*, size_t);
unsigned int operator "" _w(const char*);

void g()
{
  1.2_w      // operator"" _w(1.2)
  u"one"_w   // operator"" _w(u"one", 3)
  12_w       // operator"" _w("12")
}
```

Using (C++11)

- Typedef won't work well with templates

- Using introduce type alias

```
using myint = int;
template <class T> using ptr_t = T*;

void f(int) { }
// void f(myint) { }   syntax error: redeclaration of f(int)

// make mystring one parameter template
template <class CharT> using mystring =
    std::basic_string<CharT,std::char_traits<CharT>>;
```

Structured bindings (C++17)

```cpp
#include <tuple>   // C++11 tuple

auto get()          // C++14 return type deduction
{
    return std::make_tuple(“hello”,42);
}

int f1()
{
    auto t = get();    // t is a tuple
    std::cout << std::get<1>(t);  // 42
}

int f2()
{
    std::string s;
    int         i;

    std::tie(s,i) = get();        // C++11
    std::cout << s << i;
}
```

Structured bindings

```cpp
int f3()  // C++17
{
    auto [fraction, reminder] = divide_reminder(16,3);        // C++17
    std::cout << "16/3 == " << fraction << ", reminder is " << reminder;
}

int f4()  // C++17
{
    auto  t = std:make_tuple("hello", 42);
    auto& [s, i] = t;
    ++i;
    std::cout << s << i;      // "hello" 43
    std::cout << std::get<0>(t) << " " << std::get<1>(t);     // "hello" 43
}

struct C { std::string name; int nstuds; };

int f4()  // C++17
{
    C m{"Multiparadigm prog", 55};
    std::array<int, 5> arr{1,2,3,4,5};

    auto [s, n] = m;
    auto [i1,i2,i3,i4,i5] = arr;
}
```

Structured bindings

```
std::map<std::string, std::string> phone_book
{
  {"abel", "+36 30 123 4567"},
  {"bela", "+36 30 234 5678"},
  ...
}

int f5()  // C++17
{
    for ( const auto &[person, phone] : phone_book)
    {
        std::cout << "person " << person
                  << ", phone " << phone << '\n';
    }
}
```

Class template deduction (C++17)

- The compiler can deduce template parameter(s) from
    - Declaration that specifies initialization
    - New expression
    - Function-style cast expressions

```
// examples from cppreference.com
std::pair p(2,4.5)  // C++11: std::pair<int,double>(2,4.5)
std::vector v = { 1, 2, 3, 4}; // std::vetor<int>

template <class T> struct A { A(T,T); };
auto y = new A{1,2}; // A<int>{1,2}

std::mutex mtx;
auto lck = std::lock_guard(mtx); // std::lock_guard<std::mutex>(mtx)
std::copy_n(v1,3,std::back_insert_iterator(v2)); // back_inserter(v2)
```

Class template deduction (C++17)

- Automatic and User defined deduction guideline

```cpp
// example from cppreference.com
template <class T> struct Container
{
  Container(T t) {}
  template<class It> Container(It beg, It end);
};

template<class It>
Container(It beg, It end) ->
      Container<typename std::iterator_traits<It>::value_type>;

int main()
{
  Container c(7) // ok T=int, using automatic guide
  std::vector<double> vd = { 3.14, 4.14, 5.14 };
  auto c = Container(v.begin(), v.end()); // ok, T=double using guide
  Container d = {5,6}; // error
}
```

Attributes (C++11)

- Extra information helping the compiler or static analysis tools or others

  – Like OpenMP

- Earlier non standard, compiler dependent

- Compiler ignores unknown attributes (since C++17)

```
#pragma once

void fatal() __attribute__ ((noreturn));

struct S
{
  char t[3];
} __attribute__ ((aligned (8)));

#if COMPILING_DLL
  #define DLLEXPORT __declspec(dllexport)
#else
  #define DLLEXPORT __declspec(dllimport)
#endif
```

Attributes

- From C++11

- Almost everything can be annotated

  - Type

  - Function

  - Enum

  - ...

```
[[attr]]
[[namespace::attr]]

// C++11
[[noreturn]] void terminate() noexcept;
[[carries_dependency]]

// C++14
[[depricated]]
[[depricated("reason")]]
```

Attributes in C++17

```
switch(c)
{
case 'a':
         f();
         [[fallthrough]];
case 'A':
         g();
}

[[nodiscard]] int foo();
void bar()
{
  foo(); // warning, return value of [[nodiscard]] function discarded
}

[[nodiscard]] struct NoDiscard { ... };

NoDiscard f();
void bar()
{
  f(); // warning, return value of [[nodiscard]] type discarded
}

void f()
{
  [[maybe_unused]] int y = 42; // Do not warn on unused y
}
```

Filesystem (C++17)

- Path object

- Directory_entry

- Directory iterators

- Support functions

    - Info from path

    - File manipulation: copy, move, symlink, …

    - Space, filesize, last write time, ...

    - Permissions

        ...

Examples

```cpp
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main()
{
  fs::path pathToShow("/home/gsd/ftp/file/file.cpp");

  std::cout << "exists() = " << fs::exists(pathToShow) << "\n"
            << "root_name() = " << pathToShow.root_name() << "\n"
            << "root_path() = " << pathToShow.root_path() << "\n"
            << "relative_path() = " << pathToShow.relative_path() << "\n"
            << "parent_path() = " << pathToShow.parent_path() << "\n"
            << "filename() = " << pathToShow.filename() << "\n"
            << "stem() = " << pathToShow.stem() << "\n"
            << "extension() = " << pathToShow.extension() << "\n";
}

$ g++ -std=c++17
```

Examples

```cpp
#include <experimental/filesystem>
#include <iostream>

namespace fs = std::experimental::filesystem;

int main()
{
  fs::path pathToShow("/home/gsd/ftp/file/file.cpp");

  std::cout << "exists() = " << fs::exists(pathToShow) << "\n"
            << "root_name() = " << pathToShow.root_name() << "\n"
            << "root_path() = " << pathToShow.root_path() << "\n"
            << "relative_path() = " << pathToShow.relative_path() << "\n"
            << "parent_path() = " << pathToShow.parent_path() << "\n"
            << "filename() = " << pathToShow.filename() << "\n"
            << "stem() = " << pathToShow.stem() << "\n"
            << "extension() = " << pathToShow.extension() << "\n";
}

$ g++ -std=c++17 -lstdc++fs
$ ./a.out
exists() = 1
root_name() = ""
root_path() = "/"
relative_path() = "home/gsd/ftp/file/file.cpp"
parent_path() = "/home/gsd/ftp/file"
filename() = "file.cpp"
stem() = "file"
extension() = ".cpp"
```

Create filename

```cpp
#include <experimental/filesystem>
#include <iostream>

namespace fs = std::experimental::filesystem;

int main( int argc, char *argv[])
{
  fs::path pathToShow("/home/gsd/ftp/file/file.cpp");

  fs::path myPath{ argc > 1 ? argv[1] : fs::current_path() };

  myPath /= "subdir1";
  myPath /= "subdir2";
  myPath /= "file";
  myPath += ".cpp";

  std::cout << "myPath = "     << myPath << "\n";
  std::cout << "filename() = " << myPath.filename() << "\n";
}

$ g++ -std=c++17 -lstdc++fs
$ ./a.out
myPath = "/home/gsd/ftp/file/subdir1/subdir2/file.cpp"
filename() = "file.cpp"
```

Recursive directory listing (1)

```cpp
#include  <string>
#include <iostream>
#include <sstream>
#include <experimental/filesystem>

using namespace std;
namespace fs = std::experimental::filesystem;

std::uintmax_t ComputeFileSize(const fs::path& pathToCheck)
{
  if (fs::exists(pathToCheck) && fs::is_regular_file(pathToCheck))
  {
    auto err = std::error_code{};
    auto filesize = fs::file_size(pathToCheck, err);
    if (filesize != static_cast<uintmax_t>(-1))
      return filesize;
  }
  return static_cast<uintmax_t>(-1);
}

void DisplayFileInfo(const std::experimental::filesystem::v1::directory_entry & entry,
                     std::string &lead, std::experimental::filesystem::v1::path &filename)
{
  time_t cftime = chrono::system_clock::to_time_t(fs::last_write_time(entry));
  cout << lead << " " << filename << ", " << ComputeFileSize(entry)
       << ", time: " << std::asctime(std::localtime(&cftime));
}
```

Recursive directory listing (2)

```cpp
void DisplayDirTree(const fs::path& pathToShow, int level)
{
    if (fs::exists(pathToShow) && fs::is_directory(pathToShow))
    {
        auto lead = std::string(level * 3, ' ');
        for (const auto& entry : fs::directory_iterator(pathToShow))
        {
            auto filename = entry.path().filename();
            if (fs::is_directory(entry.status()))
            {
                cout << lead << "[+] " << filename << "\n";
                DisplayDirTree(entry, level + 1);
                cout << "\n";
            }
            else if (fs::is_regular_file(entry.status()))
                DisplayFileInfo(entry, lead, filename);
            else
                cout << lead << " [?]" << filename << "\n";
        }
    }
}

int main(int argc, char* argv[])
{
  const fs::path pathToShow{ argc >= 2 ? argv[1] : fs::current_path() };
  DisplayDirTree(pathToShow, 0);
}
```

Any (C++17)

- Type-safe container for a single value of any type

- Namespace function any_cast provides access

- type() returns type_info (or typeid(void))

- Small object optimization is possible

  - But only when move constructor is nothrow

Type safe access of Any

- any_cast checks the content run-time
- May throw std::bad_any_cast

```
std::any a = 3.14;
std::cout << a.type().name() << ' ' << std::any_cast<double>(a) << '\n';
a = 42;
std::cout << a.type().name() << ' ' << std::any_cast<int>(a) << '\n';
std::cout << sizeof(a) << ' ' << sizeof(int) << '\n';

try
{
  std::cout << std::any_cast<double>(a) << '\n';
}
catch( std::bad_any_cast& e)
{
  std::cerr << e.what() << '\n';
}


d: 3.14
i: 42
i: bad_any_cast
```

Works with user types

```cpp
struct MyStruct
{
  int i;
  double d;

};
std::ostream& operator<<(std::ostream& os, const MyStruct& s)
{
  os << "[ " << s.i << ", " << s.d << " ]";
  return os;
}
void f()
try
{
  MyStruct s{1, 3.14};
  a = s;
  std::cout << a.type().name() << ": " << std::any_cast<MyStruct>(a) << '\n';
  std::cout << sizeof(a) << " " << sizeof(s) << '\n';
}
catch (std::bad_any_cast& e)
{
    std::cout << e.what() << '\n';
}

8MyStruct: [ 1, 3.14 ]
16 16
```

Works with user types

```cpp
struct MyStruct
{
  int i;
  double d;
  char t[100];
};
std::ostream& operator<<(std::ostream& os, const MyStruct& s)
{
  os << "[ " << s.i << ", " << s.d << " ]";
  return os;
}
void f()
try
{
  MyStruct s{1, 3.14};
  a = s;
  std::cout << a.type().name() << ": " << std::any_cast<MyStruct>(a) << '\n';
  std::cout << sizeof(a) << " " << sizeof(s) << '\n';
}
catch (std::bad_any_cast& e)
{
    std::cout << e.what() << '\n';
}

8MyStruct: [ 1, 3.14 ]
16 120
```

any_cast has multiple specializations

```
void f()
{
  std::any a = 42;

  if ( a.has_value() )
  {
    std::cout << a.type().name() << '\n';

    int *ip = std::any_cast<int>(&a);    // not any_cast<int*> !!!
    std::cout << *ip << '\n';
  }
  a.reset();

  If ( ! a.has_value() )
    std::cout << a.type().name() << '\n';
}


1
v
```

any_cast has multiple specializations

```
void f()
{
  std::any a = 42;

  if ( a.has_value() )
  {
    std::cout << a.type().name() << '\n';

    int *ip = std::any_cast<int*>(&a);   // ERROR
    std::cout << *ip << '\n';
  }
  a.reset();

  If ( ! a.has_value() )
    std::cout << a.type().name() << '\n';
}


37: error: cannot convert int** to int* in initialization
```

Optional (C++17)

- Maybe monad implementation

- Replaces return types like std::pair<T,bool>

- Optional contains value

  - Initialized/assigned with value of T

  - Initialized/assigned with optional<T> which contains value

- Optional does not contain value

  - Default initialized or initialized with value of std::nullopt_t

  - Initialized/assigned with optional<T> which does not contain value

- If optional<T> contains a value, than it is allocated as T

  - Not a pointer based heap storage

- Convertible to bool: true if contains value

- No optional reference

std::optional

```cpp
std::optional<double> convert( const std::string& s)
try
{
    return std::stoi(s);   // C++11
}
catch (std::invalid_argument e)
{
    return {};
}
catch (std::out_of_range e)
{
    return {};
}

int main()
{
    double d = convert("-3.14e-5").value_or(0.0);
}
```

Use of optional

```cpp
void f(bool b1)
{
  std::optional<int> opt1;
  std::cout << opt1.value_or(-1) << '\n';
  try
  {
    std::cout << opt1.value() << '\n';
  }
  catch( std::bad_optional_access& e)
  {
    std::cerr << e.what() << '\n';
  }
  opt1 = b1 ? std::optional<int>(42) : std::nullopt;

  std::cout << opt1.value_or(-1) << '\n';
  if ( opt1 )
  {
    std::cout << opt1.value() << '\n';
    *opt1 = 2;    // access contained data, also ->  exists
    int i = opt1.value();
    std::cout << i << '\n';
  }
}

-1
bad optional access
42
42
2
```

Use of pointers

```cpp
void f(bool b1)
{
  std::optional<std::string> opt2;
  *opt2 = "Hello";

  std::cout << *opt2 << '\n';
  std::cout << std::boolalpha << opt2.has_value() << '\n';

  std::cout << opt2.value_or("no value") << '\n';
  std::string s = *std::move(opt2);

  std::cout << s << ", " << opt2->size() << '\n';
}
```

```
Hello
false
no value
Hello, 0
```

Variant (C++17)

- Type-safe union with automatic discriminant

- Holds one of the alternative types or no value

  - No value state is hard to achieve

- Variant is not allowed to allocate dynamic memory

  - The contained object representation is inside the variant object

- Can hold the same type more than once

  - Can hold the type with different cv qualifier

- Default initialized variant hold the first alternative (index() == 0)

  - If it is unambiguous

- Not allowed to hold a reference or an array

std::variant

```
std::variant<int, std::string, double>  v;

v = 55;

try
{
  std::cout << std::get<double>(v);
}
catch (std::bad_variant_access e)
{
  std::err << "not a double";
}


// supports non-POD types (unions does not)
```

std::variant

```cpp
void f()
{
    std::variant<int, float> v, w;
    v = 42; // v contains int
    int i = std::get<int>(v);
    std::cout << i << '\n';
    std::cout << std::get<0>(v) << '\n';
    w = std::get<0>(v); // same effect as the previous line
    w = v;              // same effect as the previous line

 // std::get<double>(v); // error: no double in [int, float]
 // std::get<3>(v);      // error: valid index values are 0 and 1

    try
    {
      std::get<float>(w); // w contains int, not float: will throw
    }
    catch (std::bad_variant_access& e)
    {
      std::cerr << e.what() << '\n';
    }
}
```

```
42
42
Unexpected index
```

std::variant

```cpp
void f()
{
    std::variant<int, float> v, w;
    v = 42; // v contains int
    int i = std::get<int>(v);
    std::cout << i << '\n';
    std::cout << std::get<0>(v) << '\n';
    w = std::get<0>(v); // same effect as the previous line
    w = v;              // same effect as the previous line

    if ( auto pval = std::get_if<int>(&v) )  // pointer to variant!
    {
      std::cout << *pval << '\n';
    }
    else
    {
      std::cerr << "Other type" << '\n';
    }
}
```

```
42
42
Unexpected index
```

std::variant

```
void f()
{
    std::variant<std::string, double> s1("Hello");      // conversion works
    std::variant<std::string const char *> s2("Hello"); // choose const char *

    s1 = "Hallo";   // conversion when non-ambigous

    std::cout << std::boolalpha
              << "variant holds double? "
              << std::holds_alternative<double>(s1) << '\n'
              << "variant holds string? "
              << std::holds_alternative<std::string>(s1) << '\n';
        //    << std::holds_alternative<int>(s1) << '\n';  // compile error
}

variant holds double? false
variant holds string? true
```

std::visit

```
template <class Visitor, class... Variants>
constexpr /*see below*/ visit(Visitor&& vis, Variants&&... vars);

// vis is a callable
// The return type is deduced from the returned expression as if by decltype


template <class R, class Visitor, class... Variants>
constexpr R visit(Visitor&&, Variants&&...);        // since C++20


Applies the vis visitor to the variants vars:

std::invoke(std::forward<Visitor>(vis), std::get<is>(std::forward<Variants>(vars))...)
```

std::visit

```cpp
using var_t = std::variant<int, long, double, std::string>; // the variant to visit
template<class T> struct always_false : std::false_type {}; // helper type for the visitor

int main()
{
    std::vector<var_t> vec = {10, 15l, 1.5, "hello"};
    for(auto& v: vec)
    {
        // 1. void visitor, only called for side-effects (here, for I/O)
        std::visit([](auto&& arg){std::cout << arg;}, v);

        // 2. value-returning visitor, demonstrates the idiom of returning another variant
        var_t w = std::visit([](auto&& arg) -> var_t {return arg + arg;}, v);

        // 3. type-matching visitor: a lambda that handles each type differently
        std::cout << ". After doubling, variant holds ";
        std::visit([](auto&& arg) {
            using T = std::decay_t<decltype(arg)>;
            if constexpr (std::is_same_v<T, int>)
                std::cout << "int with value " << arg << '\n';
            else if constexpr (std::is_same_v<T, long>)
                std::cout << "long with value " << arg << '\n';
            else if constexpr (std::is_same_v<T, double>)
                std::cout << "double with value " << arg << '\n';
            else if constexpr (std::is_same_v<T, std::string>)
                std::cout << "std::string with value " << std::quoted(arg) << '\n';
            else
                static_assert(always_false<T>::value, "non-exhaustive visitor!");
        }, w);
    }
}
```

std::visit

```cpp
using var_t = std::variant<int, long, double, std::string>; // the variant to visit

// helper type for the visitor
template<class... Ts> struct overloaded : Ts... { using Ts::operator()...; };
template<class... Ts> overloaded(Ts...) -> overloaded<Ts...>;

int main()
{
    std::vector<var_t> vec = {10, 15L, 1.5, "hello"};

    for (auto& v: vec)
    {
        // type-matching visitor: a class with 3 overloaded operator()'s
        std::visit(overloaded {
            [](auto arg) { std::cout << arg << ' '; },
            [](double arg) { std::cout << std::fixed << arg << ' '; },
            [](const std::string& arg) { std::cout << std::quoted(arg) << ' '; },
        }, v);
    }

}


10 15 1.500000 "hello"
```

string_view (C++17)

- basic_string_view<char>

- Can refer a constant contiguous char-like sequence

- Therefore iterator and const_iterator are the same types

- Typical implementation is

```
struct string_view
{
    CharT *ptr;
    size_t len;
};
```

String_view example

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

int main()
{
  std::string s =  "Hello world";
  std::string_view sv = s;
  std::string_view sv1(s.data(),5);
  std::cout << sv1 << '\n';
  sv = sv.substr(6);
  std::cout << sv << '\n';
  std::cout << sv[2] << '\n';
  // sv[2] = 'R';   // ERROR, cannot modify characters
  std::string_view sv2 = sv.substr(1);
  std::cout << sv2 << '\n';
  std::string_view sv3 = sv.substr(-1);

}


Hello
world
r
orld
terminate called after throwing an instance of 'std::out_of_range'
  what():  basic_string_view::substr: __pos (which is 18446744073709551615) > this->size()
(which is 5)
Aborted (core dumped)
```

String_view is not owner

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

int main()
{
  std::string_view sv5;
  {
    sv5 = std::string{"Hello world"};
    std::string_view sv6 = "Hello world"s;
    std::cout << sv6[3] << '\n';
  }
  std::cout << sv5[3] << '\n';
}
```

par_algorithms (C++17)

- Based on Intel's Threading Building Blocks (TBB)

- Extends STL algorithms with execution policy

  - std::execution::seq   Sequential execution

  - std::execution::par   Parallel execution

  - std::execution::par_unseq   Parallel SIMD execution

- These policies are permissions not obligations. Implementation may choose what can be parallelized

- Minimal requirement: forward iterator

- The programmer's task to ensure that element access functions will not cause dead lock or data race

- In case of parallelization and vectorization access must not use any blocking synchronization

Vectorization

```cpp
std::vector<int> v {1,2, ... };

int sum { std::accumulate(v.begin(), v.end(), 0) };


int sum = 0;
for ( size_t i = 0; i < v.size(); ++i)
{
    sum += v[i];
}



int sum = 0;
for ( size_t i = 0; i < v.size() / 4; i+=4)
{
    sum += v[i] + v[i+1] + v[i+2] + v[i+3];    // most CPU supports this
}
// handle if (v.size()/4) is not null...
```

Transform vs reduce

```cpp
#include <iostream>
#include <chrono>
#include <vector>
#include <numeric>
#include <execution>

int main()
{
    std::vector<double> v(10'000'007, 0.5);
    {
        auto t1 = std::chrono::high_resolution_clock::now();
        double result = std::accumulate(v.begin(), v.end(), 0.0);
        auto t2 = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double, std::milli> ms = t2 - t1;
        std::cout << std::fixed << "std::accumulate result " << result
                << " took " << ms.count() << " ms\n";
    }
    {
        auto t1 = std::chrono::high_resolution_clock::now();
        double result = std::reduce(std::execution::par, v.begin(), v.end());
        auto t2 = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double, std::milli> ms = t2 - t1;
        std::cout << "std::reduce result " << result
                << " took " << ms.count() << " ms\n";
    }
}
```

Transform_reduce

```cpp
// A o B == a1*b1 + a2*b2 + … + an*bn

#include <vector>
#include <functional>
#include <iostream>
#include <numeric>
#include <execution>

int main()
{
    std::vector<double> xvalues(10007, 1.0), yvalues(10007, 1.0);

    double result = std::transform_reduce(
        std::execution::par,
        xvalues.begin(), xvalues.end(),
        yvalues.begin(), 0.0
    );
    std::cout << result << '\n';
}
```