

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

```
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

```
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).

Enumerations

- Name constants of values of the underlying type
- Pre C++11 (C-like) enums
 - Underlying type is not fixed (implementation defined)
- C++11 enum class
 - Creates own “namespace”
 - Underlying type can be given

```
// underlying type is implementation defined
enum Color { yellow, green, blue, red }; // C style enum
enum class Color { yellow, green, blue, red }; // enum class

// underlying type is specified
enum class Color : char { yellow, green, blue, red }; // enum class
enum struct Color : char { yellow, green, blue, red }; // same as class

enum struct Color : char; // ok, complete type
```

Enum classes (C++11)

```
enum Alert { green, yellow=5, election, red }; // traditional enum

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

enum class TrafficLight { red, yellow=5, green };

Alert a = 7; // compile error: no int->Color conversion
Alert a1 = static_cast<Alert>(i); // ok, if i in 0..3

Color c = 7; // compile error: no int->Color conversion
Color c1 = Color::blue; // ok

int i2 = red; // ok: automatic Alert->int conversion
int i3 = Alert::red; // compile error in C++98; ok in C++0x

int i4 = blue; // compile error: blue not in scope
int i5 = Color::blue; // compile error: no Color->int conversion
int i6 = static_cast<int>(Color::blue) // ok, 1
```

Enum classes (C++11)

```
// by default, the underlying type is int
enum struct TrafficLight { red, yellow, green };

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

enum E { E1 = 1, E2 = 2, Ebig = 0xFFFFFFFF0U }; // how big is an E?

// we are specific with the underlying type
enum class EE : unsigned long { EE1 = 1, EE2 = 2, EEbig = 0xFFFFFFFF0U };

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

// definition should exist in one source file
enum class Color_code : char { red, yellow, green, blue };
```

Enum classes (C++11)

- Not named values in the underlying type are valid enum values
- List of the names can be empty
- This allows us to create new strong types

```
enum byte : unsigned char {}; // byte is a new type with size == unsigned char
```

```
byte x{0xff};           // ok, but only {} works
byte y = 0xff;         // error, no automatic conversion
byte z = byte{0xff};   // ok
byte w = byte{-1};     // error, narrowing conversion, forbidden
```

```
unsigned char uc{1};
w = uc;           // error, no automatic conversion
```

```
enum unix_handle : int { Error = -1, Ok = 0 };
```

```
unix_handle_t f(int i)
{
    if ( i < 0 )      return unix_handle_t::Error;
    else if ( i == 0 ) return unix_handle_t::Ok;
    else             return unix_handle_t{i};
}
```

Enum classes (C++11)

- Names can be exported to the encapsulating class scope

```
enum Color : unsigned char { yellow, green, blue, red };
```

```
struct traffic_t  
{
```

```
    Color s = Color::yellow;  
};
```

```
void f(traffic_t t)  
{  
    t.s = Color::yellow;  
}
```

Enum classes (C++11)

- Names can be exported to the encapsulating class scope

```
enum Color : unsigned char { yellow, green, blue, red };
```

```
struct traffic_t  
{  
    using enum Color;  
    Color s = yellow;  
    Color s = Color::yellow;  
};
```

```
void f(traffic_t t)  
{  
    t.s = Color::yellow;  
    t.s = traffic_t::yellow;  
}
```

Initializer list

- Lightweight proxy accessing a temporary `const T[]` maybe in read only memory
- Created when brace-list: list-initialize, assignment, function call, auto, range for
- `Initializer_list` applies shallow copy

```
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
    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")
}
```

Piecewise_construct

```
{
std::cout << "regular: \n";
std::pair<MyType, MyType> p { MyType{"one", 1}, MyType{"two", 2}};
}

{
std::cout << "piecewise + forward: \n";
std::pair<MyType, MyType> p2(std::piecewise_construct,
    std::forward_as_tuple("one", 1),
    std::forward_as_tuple("two", 2));
}
```

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)

```
#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

- Structured binding helps

```
#include <iostream>
#include <vector>

int main()
{
    std::vector v = { 1, 2, 3 };

    if (auto [s,it] = std::pair{ v.size(),v.begin() }; s > 0 && s < *it) {
        std::cerr << "s < c" << '\n';
    }
    else {
        std::cerr << "s == c" << '\n';
    }
    return 0;
}
```

Structured bindings

```
#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

- Binds the specified names to subobjects or members
 - Array-like
 - Tuple-like
 - Member

```
#include <iostream>
```

```
int main()  
{  
    int arr[] = { 1, 2 };  
    auto [x,y] = arr;  
  
    std::cout << x << y << arr[0] << arr[1] << '\n';  
    return 0;  
}
```

```
1212
```

Structured bindings

- Array like

```
#include <iostream>

int main()
{
    int arr[] = { 1, 2 };
    auto [x,y] = arr;
    --x;
    std::cout << x << y << arr[0] << arr[1] << '\n';
    return 0;
}

0212
```

Structured bindings

- Introduces tmp is a 'uniquely named' variable
- If no reference initialization

```
#include <iostream>
```

```
int main()  
{  
    int arr[] = { 1, 2 };  
    auto [x,y] = arr; // creates int tmp[2];int &x=tmp[1];int &y=tmp[2]  
    --x;  
    std::cout << x << y << arr[0] << arr[1] << '\n';  
    return 0;  
}
```

```
0212
```

Structured bindings

- If reference initialization happens, names bind to the initialization object

```
#include <iostream>

int main()
{
    int arr[] = { 1, 2 };
    auto& [x,y] = arr; // creates int &x=arr[1]; int &y=arr[2]
    --x;
    std::cout << x << y << arr[0] << arr[1] << '\n';
    return 0;
}
```

Structured bindings

- If reference initialization happens, names bind to the initialization object

```
#include <iostream>
```

```
int main()  
{  
    int arr[] = { 1, 2 };  
    auto& [x,y] = arr; // creates int &x=arr[1]; int &y=arr[2]  
    --x;  
    std::cout << x << y << arr[0] << arr[1] << '\n';  
    return 0;  
}
```

```
0202
```

Structured bindings

- Works with `std::array` too

```
#include <array>
#include <iostream>

int main()
{
    std::array<int,2> arr = { 1, 2 };
    auto& [x,y] = arr;
    --x;
    std::cout << x << y << arr[0] << arr[1] << '\n';
    return 0;
}

0202
```

Structured bindings

- Tuple-like
 - `Std::tuple_size<Tmp>::value` should be the number of ids

```
#include <iostream>
#include <tuple>

int main()
{
    int    a = 1;
    double b = 3.14;
    long   c = 2L;

    std::tuple<int, double&, long&&> tpl{a,b,std::move(c)};
    auto [x,y,z] = tpl;

    std::cout << x << ", " << y << ", " << z << '\n';
}
```

Structured bindings

- Tuple-like
 - `Std::tuple_size<Tmp>::value` should be the number of ids

```
#include <iostream>
#include <tuple>

int main()
{
    int    a = 1;
    double b = 3.14;
    long   c = 2L;

    std::tuple<int, double&, long&&> tpl{a,b,std::move(c)};
    auto [x,y,z] = tpl;

    std::cout << x << ", " << y << ", " << z << '\n';
}
```

error: tuple is not copyable

Structured bindings

- Tuple-like
 - `Std::tuple_size<Tmp>::value` should be the number of ids

```
#include <iostream>
#include <tuple>

int main()
{
    int    a = 1;
    double b = 3.14;
    long   c = 2L;

    std::tuple<int, double&, long&&> tpl{a,b,std::move(c)};
    auto& [x,y,z] = tpl;

    std::cout << x << ", " << y << ", " << z << '\n';
}

1, 3.14, 2
```

Structured bindings

- Tuple-like
 - `Std::tuple_size<Tmp>::value` should be the number of ids

```
#include <iostream>
#include <tuple>

int main()
{
    int    a = 1;
    double b = 3.14;
    long   c = 2L;

    std::tuple<int, double&, long&&> tpl{a,b,std::move(c)};
    auto& [x,y,z] = tpl;

    std::cout << x << ", " << y << ", " << z << '\n';
}

1, 3.14, 2
```

Structured bindings

- Members
 - May not have anonymous union
 - Number of ids should be the same as non-static members

```
#include <iostream>
```

```
struct C
```

```
{  
    int    a = 1;
```

```
    double b = 3.14;  
};
```

```
int main()
```

```
{  
    C obj;  
    auto [x,y] = obj;  
  
    std::cout << x << ", " << y << '\n';  
}
```

```
1, 3.14
```

Structured bindings

- Members
 - May not have anonymous union
 - Number of ids should be the same as non-static members

```
#include <iostream>

struct C
{
    int    a = 1;
    static const int xx = 99;    // static members are not considered
    double b = 3.14;
};
```

```
int main()
{
    C obj;
    auto [x,y] = obj;

    std::cout << x << ", " << y << '\n';
}
```

1, 3.14

Structured bindings

```
#include <utility>
#include <map>

auto divide_reminder(int a, int b)
{
    return std::pair{ a/b, a%b };
}

void f()
{
    auto [fraction, reminder] = divide_reminder(16,3);
    std::cout << "16/3 == " << fraction << ", reminder is " << reminder;

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

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

Nested namespace

```
#include <iostream>

namespace A {
    namespace B {

        struct C { void f(); };
    }
}

void A::B::C::f() { std::cerr << "f()\n"; }

int main()
{
    A::B::C c;
    c.f();
    return 0;
}
```

Nested namespace

```
#include <iostream>

namespace A::B {

    struct C { void f(); };

}

void A::B::C::f() { std::cerr << "f()\n"; }

int main()
{
    A::B::C c;
    c.f();
    return 0;
}
```

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  
[[deprecated]]  
[[deprecated("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
}
```

Selection statements with initializers

- ISO/IEC JTC1 SC22 WG21 P0305R1 (Thomas Köppe)

Selection statements with initializers

```
/* C language, before C99 */
{
    int i;
    for ( i = 0; i < 10; ++i) {
        /* use i here */
    }
    /* i still visible here */
}
```

```
/* C++ language, C since C99 */
{
    for ( int i = 0; i < 10; ++i) {
        /* use i here */
    }
    /* i is not visible here */
}
```

Selection statements with initializers

- ```
/* C++, since the beginning */
{
 if (const char *path = std::getenv("PATH")) {
 /* use path here */
 }
 else {
 /* path is also available here, nullptr */
 }
 /* path not available here */
}

{
 if (auto sp = wp.lock()) { /* shared_ptr from weak_ptr */
 /* use sp here */
 }
 /* sp is destructed here */
}
```

# Selection statements with initializers

- Not works well, when
  - it is not the declared variable we depend on
  - the success/fail is not usual int/bool/ptr != 0

```
std::set<int> s;
```

```
auto p = s.insert(42);
if (p.second) {
 std::cerr << "insert ok" << '\n';
}
else {
 std::cerr << "insert failed" << '\n';
}
```

```
std::mutex mut1, mut2, mut3;
```

```
int ret = std::try_lock(mut1, mut2, mut3); // many OS functions
if (-1 == ret) {
 std::cerr << "locks done" << '\n';
}
```

# Selection statements with initializers

- Declaration is allowed in if and switch statements
  - The scope of declared variable is not “leaking” out
  - More flexibility for the condition

```
std::set<int> s;
```

```
// auto p = s.insert(42);
if (auto p = s.insert(42); p.second) {
 std::cerr << "insert ok" << '\n';
}
else {
 std::cerr << "insert failed" << '\n';
}
```

```
std::mutex mut1, mut2, mut3;
```

```
// int ret = std::try_lock(mut1, mut2, mut3);
if (int ret = std::try_lock(mut1, mut2, mut3); -1 == ret) {
 std::cerr << "locks done" << '\n';
}
```

# Selection statements with initializers

- Declaration is allowed in if and switch statements
  - The scope of declared variable is not “leaking” out
  - More flexibility for the condition

```
std::set<int> s;
```

```
// auto p = s.insert(42);
if (auto p = s.insert(42); p.second) {
 std::cerr << "insert ok" << '\n';
}
else {
 std::cerr << "insert failed" << '\n';
}
```

```
std::mutex mut1, mut2, mut3;
```

```
// int ret = std::try_lock(mut1, mut2, mut3);
if (int ret = std::try_lock(mut1, mut2, mut3); -1 == ret) {
 std::cerr << "locks done" << '\n';
} // unlock????
```

# Selection statements with initializers

- Don't trick yourself!!!

```
std::mutex mut;
std::deque<int> data;
```

```
// producer
{
 std::lock_guard sl(mut);
 data.push_back(i);
}
```

```
// consumer
if (std::lock_guard(mut); !data.empty()) { // bad!!!
 int i = data.front();
 data.pop_front();
}
```

# Selection statements with initializers

- Use `lock_guard`, `unique_lock`, `scoped_lock`, ...

```
std::mutex mut;
std::deque<int> data;
```

```
// producer
```

```
{
 std::lock_guard sl(mut);
 data.push_back(i);
}
```

```
// consumer
```

```
if (std::lock_guard sl(mut); !data.empty()) {
 int i = data.front();
 data.pop_front();
}
```

# Selection statements with initializers

- Switch is more interesting than you think.

```
#include <iostream>

int main(int argc, char *argv[])
{
 switch (argc)
 {
 case 1: std::cout << "1" << '\n'; break;
 case 2: std::cout << "2" << '\n'; break;
 default: std::cout << "d" << '\n'; break;
 }
 return 0;
}
```

# Selection statements with initializers

- Switch is more interesting than you think.

```
#include <iostream>

int main(int argc, char *argv[])
{
 switch (argc)
 {
 int x;

 case 1: std::cout << "1" << x << '\n'; break;
 case 2: std::cout << "2" << x << '\n'; break;
 default: std::cout << "d" << x << '\n'; break;
 }
 return 0;
}
```

# Selection statements with initializers

- Switch is more interesting than you think.

```
#include <iostream>

int main(int argc, char *argv[])
{
 switch (argc)
 {
 int x;

 case 1: std::cout << "1" << x << '\n'; break; // undefined beh.
 case 2: std::cout << "2" << x << '\n'; break;
 default: std::cout << "d" << x << '\n'; break;
 }
 return 0;
}
```

# Selection statements with initializers

- Switch is more interesting than you think.

```
#include <iostream>

int main(int argc, char *argv[])
{
 switch (argc)
 {
 int x = argc;

 case 1: std::cout << "1" << x << '\n'; break;
 case 2: std::cout << "2" << x << '\n'; break;
 default: std::cout << "d" << x << '\n'; break;
 }
 return 0;
}
```

# Selection statements with initializers

- Switch is more interesting than you think.

```
#include <iostream>

int main(int argc, char *argv[])
{
 switch (argc)
 {
 int x = argc;

 case 1: std::cout << "1" << x << '\n'; break;
 case 2: std::cout << "2" << x << '\n'; break;
 default: std::cout << "d" << x << '\n'; break;
 }
 return 0;
}
```

error: jump to **case** label XXX crosses initialization of **int** x

# Selection statements with initializers

- Switch is more interesting than you think.

```
#include <iostream>

int main(int argc, char *argv[])
{
 switch (int x = argc)
 {
 // works even in "old" C++

 case 1: std::cout << "1" << x << '\n'; break;
 case 2: std::cout << "2" << x << '\n'; break;
 default: std::cout << "d" << x << '\n'; break;
 }
 return 0;
}
```

# Selection statements with initializers

- Switch is more interesting than you think.

```
#include <iostream>

int main(int argc, char *argv[])
{
 switch (int x = argc; ++x)
 {
 // works since C++17

 case 1: std::cout << "1" << x << '\n'; break;
 case 2: std::cout << "2" << x << '\n'; break;
 default: std::cout << "d" << x << '\n'; break;
 }
 return 0;
}
```

# Selection statements with initializers

- Declaration list is allowed

```
#include <iostream>
#include <vector>

int main()
{
 std::vector v = { 1, 2, 3 }; // CTAD, C++17

 if (int s = v.size(), c = v.capacity(); s < c) {
 std::cerr << "s < c" << '\n';
 }
 else {
 std::cerr << "s == c" << '\n';
 }
 return 0;
}
```

# Selection statements with initializers

- A bit more interesting case

```
#include <iostream>
#include <vector>

int main()
{
 std::vector v = { 1, 2, 3 }; // CTAD, C++17

 if (int s = v.size(), it = v.begin(); s > 0 && s < *it) {
 std::cerr << "s < c" << '\n';
 }
 else {
 std::cerr << "s == c" << '\n';
 }
 return 0;
}
```

error: v.begin() is not convertible to int

# Selection statements with initializers

- Auto deduction must be consistent

```
#include <iostream>
#include <vector>

int main()
{
 std::vector v = { 1, 2, 3 }; // CTAD, C++17

 if (auto s = v.size(), it = v.begin(); s > 0 && s < *it) {
 std::cerr << "s < c" << '\n';
 }
 else {
 std::cerr << "s == c" << '\n';
 }
 return 0;
}
```

error: inconsistent deduction for 'auto'

# Selection statements with initializers

- Structured binding helps

```
#include <iostream>
#include <vector>

int main()
{
 std::vector v = { 1, 2, 3 }; // CTAD, C++17

 if (auto [s,it] = std::pair{ v.size(),v.begin()}; s > 0 && s < *it){
 std::cerr << "s < c" << '\n';
 }
 else {
 std::cerr << "s == c" << '\n';
 }
 return 0;
}
```

works fine

# Selection statements with initializers

- Ideally, we should allow multiple statements

```
#include <iostream>
#include <vector>

int main()
{
 std::vector v = { 1, 2, 3 }; // CTAD, C++17

 if (auto s = v.size(); auto it = v.begin(); s > 0 && s < *it){
 std::cerr << "s < c" << '\n';
 }
 else {
 std::cerr << "s == c" << '\n';
 }
 return 0;
}
```

error: parse error

# Aggregates

- Array
- Class type which has
  - no user-declared or inherited constructor
  - no private or protected data members
  - no virtual or private or protected base class
  - no virtual member functions
- Aggregates are basically “pure data”
  - Old name for such classes: POD types

```
struct complex_t
{
 double re = 0.; // default member initialization is ok
 double im = 0.; // but only since C++14
};
complex_t c; // (0+0i)
struct complex_t vectors[10]; // arrays are also aggregates
```

# Aggregates

- Elements:
  - Array elements
  - Base classes followed by non-static data members
- Aggregates can be initialized with list-initialization: { 1, 2, ... 9 }
  - The length of list must not exceed the number of elements
  - The list can have less or no elements
  - Array initialization cannot “skip” element: ~~{{2}=5, [0]=1}~~ C99 only!

```
int a[5] = {1, 2, 3, 4, 5}; // ok
int a[5] = {1, 2, 3, 4, 5, 6}; // compile error
int a[5] = {1, 2, 3, 4}; // ok, 4 elements are initialized, the 5th will 0
int a[5] = {}; // ok, all elements are value initialized to 0
int a[5]; // ok, all elements are value initialized to 0
int a[5] = {[4]=1, [0]=2, 3, 4, 5}; // compile error, only in C, since C99
char s[6] = "Hello"; // for char[] only
```

# Aggregates

- Class types can use designated initializers

```
struct complex_t
{
 double re = 0.; // default member initialization is ok
 double im = 0.; // but only since C++14
};
complex_t c1{}; // ok, 0.+0.i
complex_t c2{1.}; // ok, 1.+0.i
complex_t c3{1., 3.14}; // ok, 1.+3.14i
complex_t c2{.im=3.14}; // ok, 0.+3.14i
complex_t c3{.re=1.0, .im=3.14}; // ok, 1.+3.14i
complex_t c4{.im=3.14, .re=1}; // compile error: designator order vs decl.ord

struct S : complex_t
{
 int i;
};
S s{ {}, 5}; // ((0.+0.i), 5)
S s{ {1.}, 5}; // ((1.+0.i), 5)
S s{ {1., 3.14}, 5}; // ((1.+3.14.i), 5)
S s{ {.im=3.14}, 5}; // ((0.+3.14.i), 5)
S
```

# Aggregates

- Class types can have member functions and static members

```
struct complex_t
{
 double re = 0.; // default member initialization is ok
 double im = 0.; // but only since C++14
 std::string to_string() const;
};
complex_t operator~(complex_t c)
{
 return { c.re, -c.im};
}
inline std::string complex_t::to_string() const
{
 return std::to_string(re)+"+"+std::to_string(im)+"i";
}
inline std::string complex_t::to_string() const
{
 return std::to_string(re)+"+"+std::to_string(im)+"i";
}
int main()
{
 complex_t c{1, 3.14};
 std::cout << ~c << '\n';
}
```

# Aggregates

- Class types can have member functions and static members

```
struct complex_t
{
 double re = 0.; // default member initialization is ok
 double im = 0.; // but only since C++14
 std::string to_string() const;
};
complex_t operator~(complex_t c)
{
 return { c.re, -c.im};
}
inline std::string complex_t::to_string() const
{
 return std::to_string(re)+"+"+std::to_string(im)+"i";
}
inline std::string complex_t::to_string() const
{
 return std::to_string(re)+"+"+std::to_string(im)+"i";
}
int main()
{
 complex_t c{1, 3.14};
 std::cout << ~c << '\n'; // 1.0000+-3.1400i
}
```

# Aggregates

- Class types can have member functions and static members

```
struct complex_t
{
 double re = 0.; // default member initialization is ok
 double im = 0.; // but only since C++14
 std::string to_string() const;
};
complex_t operator~(complex_t c)
{
 return { c.re, -c.im};
}
inline std::string complex_t::to_string() const
{
 return std::to_string(re)+"+"+std::to_string(im)+"i";
}
inline std::string complex_t::to_string() const
{
 return std::to_string(re)+(im<0?"":"+")+std::to_string(im)+"i";
}
int main()
{
 complex_t c{1, 3.14};
 std::cout << ~c << '\n'; // 1.0000-3.1400i
}
```

# 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

```
#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

```
#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

```
#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)

```
#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)

```
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

```
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

```
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
```

Possible implementation without RTTI (LLVM Any): [https://llvm.org/doxygen/Any\\_8h.html](https://llvm.org/doxygen/Any_8h.html)

# 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

```
std::optional<double> convert(const std::string& s)
try
{
 return std::stod(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

```
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

```
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
```

# Monadic operations (C++20)

```
std::optional<UserProfile> fetchFromCache(int userId);
std::optional<UserProfile> fetchFromServer(int userId);
std::optional<int> extractAge(const UserProfile& profile);
```

```
int main() {
 const int userId = 12345;
 std::optional<int> ageNext;

 std::optional<UserProfile> profile = fetchFromCache(userId);

 if (!profile)
 profile = fetchFromServer(userId);

 if (profile) {
 std::optional<int> age = extractAge(*profile);

 if (age)
 ageNext = *age + 1;
 }
 if (ageNext)
 cout << "Next year, the user will be " << *ageNext;
 else
 cout << "Failed to determine user's age.\n";
}
```

# Monadic operations (C++20)

```
std::optional<UserProfile> fetchFromCache(int userId);
std::optional<UserProfile> fetchFromServer(int userId);
std::optional<int> extractAge(const UserProfile& profile);

int main() {
 const int userId = 12345;

 const auto ageNext = fetchFromCache(userId)
 .or_else([&]() { return fetchFromServer(userId); })
 .and_then(extractAge)
 .transform([](int age) { return age + 1; });

 if (ageNext)
 cout << format("Next year, the user will be {} years old", *ageNext);
 else
 cout << "Failed to determine user's age.\n";
}
```

# Unions

```
#include <iostream>

union check
{
 long l;
 unsigned char t[sizeof(l)];
};

int main()
{
 check c;
 c.l = 12;

 for(int i = 0; i < sizeof(c.t)/sizeof(c.t[0]); ++i)
 std::cout << std::hex << int{c.t[i]};

 return 0;
}

$ g++ conv.cpp
$./a.out
c0000000
```

# Unions

- Union types also can have member functions

```
#include <iostream>
#include <string>

struct complex_t
{
 enum { CARTESIAN, POLAR} tag;
 union
 {
 struct
 {
 double re = 0.; // default member initialization is ok
 double im = 0.; // but only since C++14
 } c;
 struct
 {
 double r = 0.; // default member initialization is ok
 double fi = 0.; // but only since C++14
 } p;
 };
 std::string to_string() const; // member function
};
```

# Unions

```
inline std::string complex_t::to_string() const
{
 switch(tag)
 {
 case CARTESIAN: return std::to_string(c.re) + (c.im<0 ? "" : "+")
 + std::to_string(c.im) + "i"; break;
 case POLAR : return std::to_string(p.r) + "*fi("
 + std::to_string(p.fi) + ")"; break;
 default : return std::string{"IMPOSSIBLE"}; break;
 }
}
int main()
{
 complex_t c1 = { .tag=complex_t::POLAR, .p={ 4., 3.14/2 } };
 complex_t c2 = { c2.CARTESIAN, .c={1.0, 3.14} };
 complex_t c3 = { complex_t::POLAR, {1.0, 3.14} };
 std::cout << c1.to_string() << '\n';

 c1.tag = complex_t::CARTESIAN; // store that we change the union tag
 c1.c = { 1, 3.14 }; // store the value of (re,im)
 std::cout << c1.to_string() << '\n';
 return 0;
}
```

# Variant (C++17)

- Type-safe union with automatic discriminant
- Holds one of the alternative types or no value
  - No-value state is almost never happens
- Variant does not allocate dynamic memory
  - The contained object representation is inside the variant object
- Can hold non-trivially copyable types
- 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` )
- Not allowed to hold a reference or an array

# std::variant

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

```
v = 55.55;
```

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

```
v = {}; // type of index() == 0, here: int
```

```
std::cout << std::get<0>(v); // 0
```

# std::variant

```
void f()
{
 std::variant<int, float> v, w; // default constructor: type of index 0
 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

Zoltán Porkoláb: Basic C++

85

# std::variant

```
void f()
{
 std::variant<int, float> v, w; // default constructor: type of index 0
 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-ambiguous

 std::cout << s1.index() << '\n'; // 0
 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
}
0
variant holds double? false
variant holds string? true
```

# std::visit

```
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

```
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"
```

# std::variant

```
#include <iostream>
#include <string>
#include <variant>
#include <vector>

struct complex_cart_t
{
 double re = 0.; // default member initialization is ok
 double im = 0.; // but only since C++14
};
struct complex_pol_t
{
 double r = 0.; // default member initialization is ok
 double fi = 0.; // but only since C++14
};
using complex_t = std::variant<complex_cart_t, complex_pol_t>;

inline std::string to_string(complex_cart_t c)
{
 return std::to_string(c.re) + "+" + std::to_string(c.im) + "i";
}
inline std::string to_string(complex_pol_t c)
{
 return std::to_string(c.r) + "*fi(" + std::to_string(c.fi) + ")";
}
```

# std::variant

```
#include <vector>

// inline std::string to_string(complex_cart_t c);
// inline std::string to_string(complex_pol_t c);

int main()
{
 std::vector<complex_t> vec = {
 complex_cart_t{1., 3.14}, complex_pol_t{ 4., 3.14/2}
 };

 for (auto c : vec)
 {
 switch (c.index())
 {
 case 0: std::cout << to_string(std::get<complex_cart_t>(c)) << '\n'; break;
 case 1: std::cout << to_string(std::get<complex_pol_t>(c)) << '\n'; break;
 }
 }
 return 0;
}
```

# std::variant

```
#include <vector>

// inline std::string to_string(complex_cart_t c);
// inline std::string to_string(complex_pol_t c);

int main()
{
 std::vector<complex_t> vec = {
 complex_cart_t{1., 3.14}, complex_pol_t{ 4., 3.14/2}
 };

 for (auto c : vec)
 {
 std::visit([](auto cc){ std::cout << to_string(cc) << '\n'; }, c);
 }
 return 0;
}
```

# 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

```
#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

```
#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';
}
```