C++ 17 Key Features

A Perfect Move Forward

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

C++17 LANGUAGE & STL CHANGES

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

MULTIPLE NAMESPACES

before C++17 :

```
namespace a {
    namespace b {
        namespace c {
                namespace d {
                struct S;
            }
        }
    }
}
```

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

MULTIPLE NAMESPACES

```
namespace a::b::c::d{
        struct S;
}
```

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

STRUCTURED BINDINGS

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

STRUCTURED BINDINGS

In C++11 we could write a function that returns a tuple:

```
std::tuple<std::vector<int>, bool, size_t>
calculatePoints(….){
  ……………………
  return std::make_tuple(vec, isOk, numOfPoints);
}
```

To work with this function, we had to use the tuple and then std::get or unpack it:

```
std::vector<int> vec;
bool res; size_t numOfPoints;
std::tie(vec, res, numOfPoints) =
calculatePoints(some_parameters);
```

STRUCTURED BINDINGS

In C++17 we can use this function like this:

```
auto [vec, isOk, numOfPoints] =
calculatePoints(some_parameters);
```

STRUCTURED BINDINGS: **ARRAYS**

```
int a[4] = {1, 2, 3, 4};
auto [x, y, z, q] = a;

auto& [xr, yr, zr, qr] = a;
xr = 10;

std::cout << a[0]; // print 10
```

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

STRUCTURED BINDINGS:
DATA MEMBERS

```
struct S
{
    int i;
    volatile float j;
};

S s;

auto [i, j] = s;

auto& [ir, jr] = s;
```

STRUCTURED BINDINGS: **MORE EXAMPLES**

```
std::unordered_map<std::string, int> carParts{{"Engine", 1}, {"Doors", 5}, {"Bolts", 200}};
for (auto& [key, val] : carParts){
    if (val < 10){
        val += 1;
    }
}
```

Who said that C++ is not Python ☺

INIT IF AND SWITCH STATEMENTS

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

INIT IF AND SWITCH STATEMENTS

Let's look at the next Example:

```
std::unordered_map<std::string, int> carParts{{"Engine", 1}, {"Doors", 5}, {"Bolts", 200}};

auto foundDoor = carParts.find("Door");
if (foundDoor != carParts.end()){
    foundDoor->second = 10;
}

auto foundBolt = carParts.find("Bolts");
if (foundBolt != carParts.end()){
    foundBolt->second = 100;
}
```

INIT IF AND SWITCH STATEMENTS

Maybe we will be templated to do this:

```
std::unordered_map<std::string, int> carParts{{"Engine", 1}, {"Doors", 5}, {"Bolts", 200}};

{
auto foundPart = carParts.find("Door");
if (foundPart != carParts.end()){
    foundPart->second = 10;
}
}

{
auto foundPart = carParts.find("Bolts");
if (foundPart != carParts.end()){
    foundPart->second = 100;
}
}
```

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

INIT IF AND SWITCH STATEMENTS

But With C++17 We can use init list:

```
std::unordered_map<std::string, int> carParts{{"Engine", 1}, {"Doors", 5}, {"Bolts", 200}};

if (auto foundPart  = carParts.find("Door"); foundPart != carParts.end()){
    foundPart->second = 10;
}

if (auto foundPart  = carParts.find("Bolt"); foundPart != carParts.end()){
    foundPart->second = 100;
}
```

INIT IF AND SWITCH STATEMENTS

We can even do this:

```
if (auto [partIter, inserted] = carParts.emplace("Window", 4); inserted){
    partIter->second +=1;
}
```

CONSTEXPR IF

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

CONSTEXPR IF

Let's look at the next Example:

```
template <typename T>
auto getT(T t)
{
    if (std::is_pointer_v<T>)
    {
        return *t;
    }
    else
    {
        return t;
    }
}
```

Will it compile ?

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

CONSTEXPR IF

Now To fix it with C++17:

```
template <typename T>
auto getT(T t)
{
    if constexpr (std::is_pointer_v<T>)
    {
        return *t;
    }
    else
    {
         return t;
    }
}
```

INLINE MEMBERS

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

INLINE MEMBERS

- C++11 and constexpr keyword allow you to declare and define static variables in one place, but it's limited to constexpr'essionsonly .

With C++17 We can do this:

```
struct mystruct{
    inline static int xyz = 100;
};
```

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

USEFUL ATTRIBUTES

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

USEFUL ATTRIBUTES

[[maybe_unused]] – if function doesn't use a variable we can suppress the warning by this attribute

```
[[maybe_unused]] float doSomthing([[maybe_unused]] int x, float y)
{
    return y;
}
```

USEFUL ATTRIBUTES

[[nodiscard]] – enforces using return value from a function

```
[[nodiscard]] float square(float x)
{
    return x*x;
}
```

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

EVALUATION ORDER

EVALUATION ORDER

What may happen here ?

```
struct MyStruct{
    void f(int x, double y, std::string z){
    }
    int x;
    double y;
};

void foo(std::unique_ptr<MyStruct>, int z);
int add(int number) {return number+1;};

int main(){
    foo(std::unique_ptr<MyStruct>(new MyStruct), add(1));
```

EVALUATION ORDER

What the solution?

```
struct MyStruct{
    void f(int x, double y, std::string z){
    }
    int x;
    double y;
};

void foo(std::unique_ptr<MyStruct>, int z);
int add(int number) {return number+1;};

int main(){
    foo(std::make_unique<MyStruct>(), add(1));
```

make_unique is not just synthetic sugar

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

EVALUATION ORDER

With C++17 its even Better

- We can use the original code as it will finish the parameter scope

- All the rules can be found here: P0145R3

- Brief summery of rules are evaluated in order a → b → c → d:
  - a.b
  - a->b
  - a->*b
  - a(b1, b2, b3)
  - b @= a
  - a[b]
  - a << b
  - b >> b

AUTO IN TEMPLATES

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

AUTO IN TEMPLATES

- template <auto>
  indicate a non-type parameter the type
  of which is deduced at the point of instantiation

Examples:

C++11 : template <typename Type, Type value> constexpr Type constant = value; constexpr auto const IntConstant42 = constant<int, 42>

C++17: template <auto value> constexpr auto constant = value;
constexpr auto const IntConstant42 = constant<42>;

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

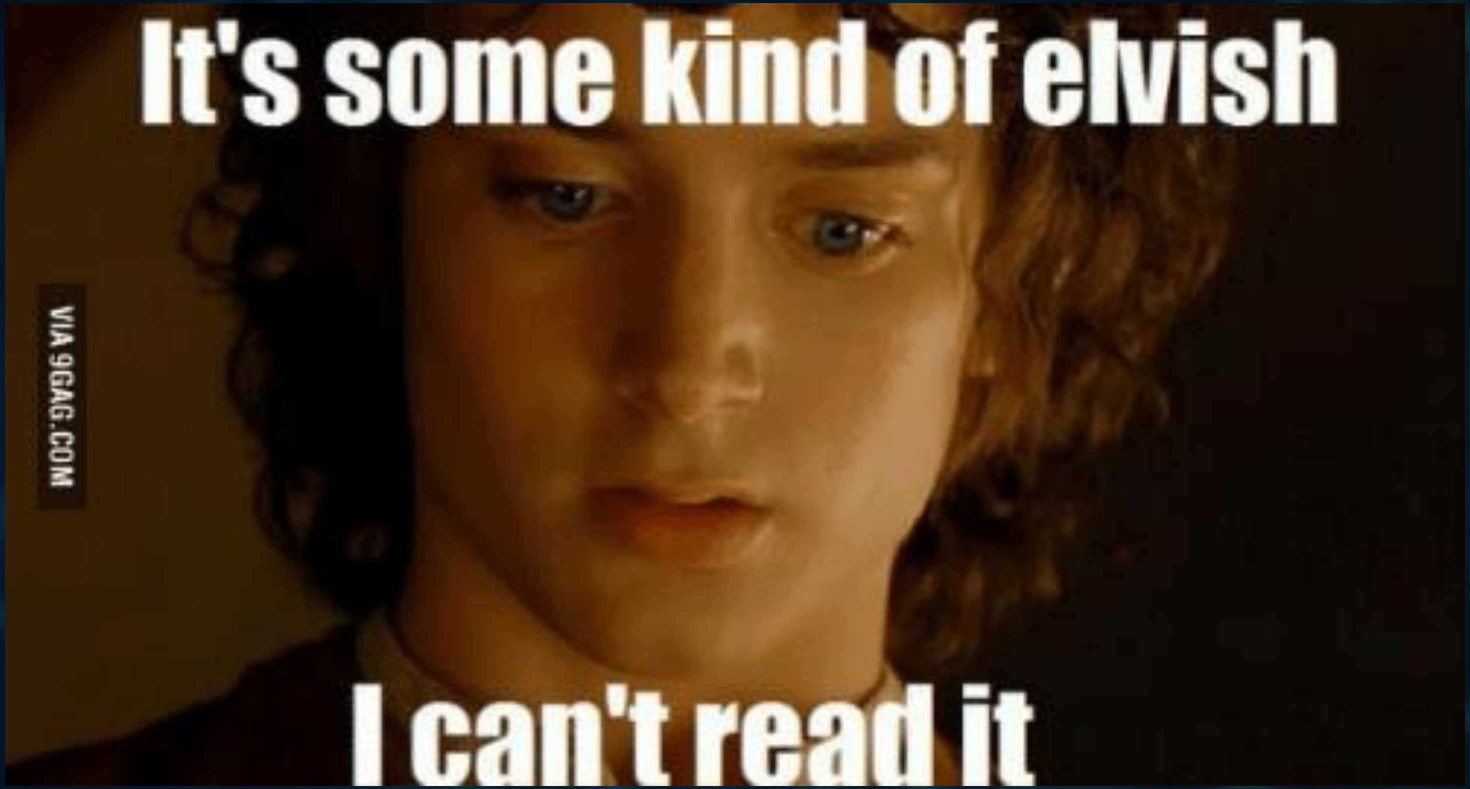
AUTO IN TEMPLATES

```
template <auto ... vs> struct HeterogenousValueList {};
using MyList1 = HeterogenousValueList<42, 'X', 13u>;

template <auto v0, decltype(v0) ... vs> struct
HomogenousValueList {}; using MyList2 =
HomogenousValueList<1, 2, 3>;
```

FOLD EXPRESSIONS

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com


It's some kind of elvish
VIA 9GAG.COM
I can't read it

FOLD EXPRESSIONS

Fold Expressions C++17:

```
template <typename... T>
auto mult(T... t)
{
    return ( t * ...);
}
```

FOLD EXPRESSIONS

Fold Expressions C++17:

```
template <typename... T>
auto avg(T... t)
{
    return (t + ...) / sizeof...(t);
}

template <typename... T>
auto somthing(T... t)
{
    const int n = 5;
    return (t + ... + n);
}
```

FOLD EXPRESSIONS

Fold Expressions C++17:

```
template <typename... Funcs>
auto sumFuncs(Funcs... f){
    return (f() + ...);
}

template <auto... numbers>
auto addAll(auto x){
    using Ret = std::common_type_t<decltype(x), decltype(numbers)...>;
    Ret sum = (numbers + ... + x);
    return sum;
}
```


YES!!!
imgflip.com
Mymemes.biz

CLASS TEMPLATE ARGUMENT TYPE DEDUCTION

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

CLASS TEMPLATE ARGUMENT TYPE DEDUCTION

- Struct Auto Deduction:

Before C++17 we had to write this

```
std::tuple<int, float, double>  t(1, 1.f, 1.);
```

Or with helpers

```
auto t2 = std::make_tuple(1, 1.f, 1.);
```

With C++17 we can just write:

```
std::tuple t3(1, 1.f, 1.)
std::tuple t4(1, 2, “Hello”)
```

CLASS TEMPLATE ARGUMENT TYPE DEDUCTION

- Will this work:

```
std::tuple t3(1, 1.f, 1.)
std::tuple t4(1, 2, "Hello")
t3 == t4 ?
```

DEDUCTION GUIDELINES

Alex Dathshkovsky | 054-7685001 | calebxyz@gmail.com

DEDUCTION GUIDELINES

Examples:

will this work?

```
Void func() {} ;

Int main(){
	std::function f(&func);
}
```

The answer is Yes.

DEDUCTION GUIDELINES

Examples:

will this work?

```
Class MyClass {
    Public:
    void f(int x, double y, std:string z) {};
} ;

Int main(){
    std::function f(&MyClass::func);
}
```

The answer is No.

DEDUCTION GUIDELINES

With C++ 17 We Can Fix That:

```
namespace std{
    template <typename Class, typename Ret, typename... Args>
    function(Ret(Class::*)(Args...)) -> function<Ret(Class&, Args...)>;

    template <typename Class, typename Ret, typename... Args>
    function(Ret(Class::*)(Args...) const) -> function<Ret(const Class&, Args...)>;
}

class MyClass{
    public:
    void f(int x, double y, std::string z){
    }
};

void f1(int i) {};
void f2(double x, int z);
```

DEDUCTION GUIDELINES

With C++ 17:

```
int main(){
    std::function func1(&f1);
    std::function func2(&f2);
    std::function func3(&MyClass::f);
    MyClass c;

    func1(1);
    func2(1., 1);
    func3(c ,1, 1.1, "c++core");
```

VARIANT

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

VARIANT

- represents a type-safe union
- at any given time either holds a value of one of its alternative types, or it holds no value

Example:

```
std::variant<int, float> v, w;
v = 12;  int i = std::get<int>(v); w = std::get<int>(v);
w = std::get<0>(v); w = v;
```

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

VARIANT

- bad_variant_access

```
std::variant<int, string> v;
v = 42;
try {
    std::get<string>(v);
} catch(std::bad_variant_access& exp) {…}
```

VARIANT

- Visit - allows to apply a visitor to a list of variants

```
std::vector<std::variant<int, double, char, float, long long>>
vect{1, 1.0, 'a', 1.f, 5};

for (auto&& v : vect)
{
    std::visit([](auto&& var){
        std::cout << var << "\n";
    }, v);
}
```

VARIANT

- Common_type
  Determines the common type among all types
  that is the type all T...
  can be implicitly converted to

```
std::common_type<char, long, float, int,
double, long long>::type res{};  //will peak double
```

VARIANT

Putting it together

```
std::vector<std::variant<int, double, char, float, long long>>
vect{1, 1.0, 'a', 1.f, 5};

//this will pick double
std::common_type<char, long, float, int, double, long long>::type res{0};

for (auto&& v : vect)
{
    std::visit([&res](auto&& var){
        res+=var;}, v);
}
```

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

OPTIONAL

OPTIONAL

- The class template std::optional manages an *optional* contained value, i.e. a value that may or may not be present

OPTIONAL

```cpp
std::optional<std::string> create(bool b)
{
    if (b)
    {
        return "Harmony";
    }

    return {};
}
```

```cpp
std::cout << create(true).value_or("Empty") << " /n"    //Harmony
<< create(false).value_or("Empty");                     //Empty
```

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

OPTIONAL: NULL OPT

```
auto create2(bool b)
{
    return b ? std::optional{"Harmony"} : std::nullopt;
}
```

```
if (auto str = create2(true))
{
    std::cout << *str; //Harmony
}
```

ANY

Alex Dathshkovsky | 054-7685001 | calebxyz@gmail.com

ANY

- The class ANY, describes a type-safe container for single values of any type

- std::any a = 1;
- a = 3.14;
- a = true;
- a = std::string(“XYZ”);

ANY

- any_cast
  Performs type-safe access to the contained object

```
std::any a = 1;
std::cout << std::any_cast<int>(a);
a = 3.14;
std::cout << std ::any_cast<double>(a);
a = true;
std::cout << std::boolalpha << std ::any_cast<bool>(a);
```

STRING_VIEW

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

STRING_VIEW

- non-owning reference to a string.
  It represents a view of a sequence of characters

- offers four type synonyms
  for the underlying character-types

STRING_VIEW

- Why do we need string_view?

- What's wrong with string ?

- There is a cost to working with strings though, and that is that they *own* the underlying buffer in which the string of characters is stored.

- Often require dynamic memory

STRING_VIEW

```
#include <iostream>

void* operator new(std::size_t n)
{
    std::cout << "[allocating " << n << " bytes]\n";
    return malloc(n);
}

bool compare(const std::string& s1, const std::string& s2)
{
    if (s1 == s2)
        return true;
    std::cout << '\"' << s1 << "\" does not match \"" << s2 << "\"\n";
    return false;
}

int main()
{
    std::string str = "turn around !!!!";

    compare(str, "every now and then i feel a bit lonely");
    compare(str, "every now and then i get little bit tired");
    compare(str, "turn around my child");

    return 0;
}
```

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

STRING_VIEW

The output

```
[allocating 41 bytes]
[allocating 63 bytes]
"turn around !!!!" does not match "every now and then i feel a bit lonely"
[allocating 66 bytes]
"turn around !!!!" does not match "every now and then i get little bit tired"
[allocating 45 bytes]
"turn around !!!!" does not match "turn around my child"
```

STRING_VIEW

The real solution is string View

```
#include <iostream>
#include <experimental/string_view>

void* operator new(std::size_t n)
{
    std::cout << "[allocating " << n << " bytes]\n";
    return malloc(n);
}

bool compare(std::experimental::string_view s1,
        const std::experimental::string_view s2)
{
    if (s1 == s2)
        return true;
    std::cout << '\"' << s1 << "\" does not match \"" << s2 << "\"\n";
    return false;
}


int main()
{
    std::string str = "turn around !!!!";

    compare(str, "every now and then i feel a bit lonely");
    compare(str, "every now and then i get little bit tired");
    compare(str, "turn around my child");

    return 0;
}
```

STRING_VIEW

The output

[allocating 41 bytes]
"turn around !!!!" does not match "every now and then i feel a bit lonely"
"turn around !!!!" does not match "every now and then i get little bit tired"
"turn around !!!!" does not match "turn around my child"

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

STRING_VIEW

Additional benefits:

creating a string_view from a substring in an existing string

```
int main()
{
    std::string str = "will this work?";

    std::experimental::string_view sv(&str.at(str.find_first_of('t')));

    compare(str, sv);

    return 0;
}
```

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

STRING_VIEW

The output

[allocating 39 bytes]
"will this work" does not match "this work"

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

EXTRACT

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

EXTRACT

- C++17 provides a method that returns a handle to the node of the container
- The handle can modify the container
- The Node is removed from a container

EXTRACT

Example: Changing a key of a map<string, int> efficiently.
Before C++17 we could write this:

```
std::map<std::string, size_t> fruitMap{{"Apple", 5}, {"Peach", 10}, {"Grapes", 12}};

auto found = fruitMap.find("Apple");
found->first = "Green Apple";
```

Will this work?

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

EXTRACT

Example: Changing a key of a map<string, int> efficiently.
Before C++17 we had to do this:

```cpp
auto found = fruitMap.find("Apple");
if (found != std::end(fruitMap))
{
    auto const value = std::move(found->second);
    fruitMap.erase(found);
    fruitMap.insert({"Green Apple", std::move(value)});
}
```

EXTRACT

Example: Changing a key of a map<string, int> efficiently.

With C++17 we can change to this and save an allocation:

```
auto found = myMap.extract("Apple");
if (!found.empty())
{
    found.key() = "Green Apple";
    myMap.insert(std::move(found));
}
```

MERGE

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

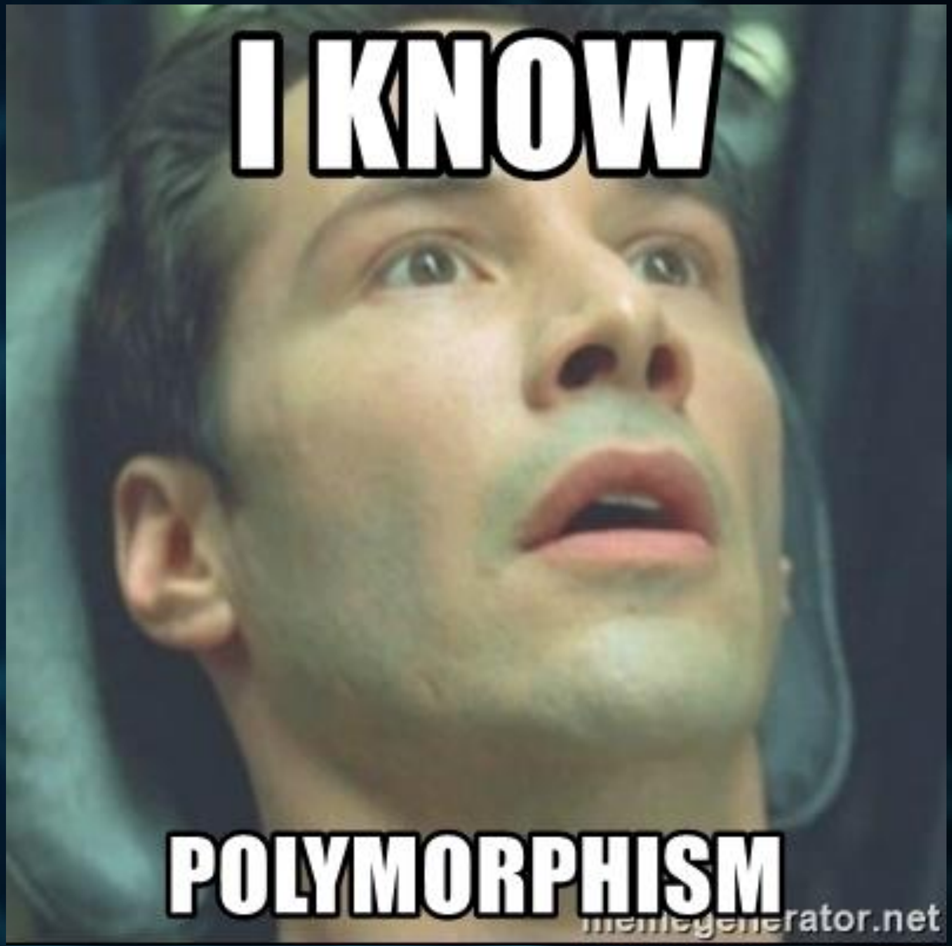
MERGE

- C++17 provides a method that merges containers (maps/sets)

```
std::map<std::string, size_t> carParts{{"Engine", 1},{"Doors", 5},{"Windows", 4}};
std::map<std::string, size_t> motorCycleParts{{"Seat", 1}, {"Helmet", 2}};
std::map<std::string, size_t> trucksParts{{"Horn", 6}, {"OptimusPrime", 1}};

carParts.merge(trucksParts);
motorCycleParts.merge(carParts);
```

PMR



PMR

- In C++ Polymorphic Memory Resource
  is a way to optimize allocators for STL Collections

- The class behaves differently upon the memory_resource call

PMR: CORE ELEMENTS

- Memory Recourse – An abstract class that defines the memory type
- Polymorphic_allocator – an implementation of a STD allocator that uses the memory resource

- Set of classes for pool resources
  - synchronized_pool_resource
  - unsynchronized_pool_resource
  - Monotonic_buffer_resource

PMR: **CORE ELEMENTS**

- Predefined STL Collections
  - pmr::vector
  - pmr::string
  - pmr::map
  - And more

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

PMR: EXAMPLE

```cpp
int main(){
    //a small buffer on the stack
    char buffer[20] = {0};
    std::fill_n(std::begin(buffer), std::size(buffer)-1, 'E');
    std::cout << buffer << "\n";
    std::pmr::monotonic_buffer_resource memory{std::data(buffer), std::size(buffer)};

    std::pmr::vector<char> vect{&memory};

    for (char c = 'a'; c <= 'e'; c++){
        vect.emplace_back(c);
    }

    std::cout << buffer;
}
```

PMR: **OUTPUT**

```
EEEEEEEEEEEEEEEEEEE
aababcdabcdeEEEEEEE
```

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

PMR: RESERVE

```
int main(){
    char buffer[20] = {0};
    std::fill_n(std::begin(buffer), std::size(buffer)-1, 'E');
    std::cout << buffer << "\n";
    std::pmr::monotonic_buffer_resource memory{std::data(buffer), std::size(buffer)};

    std::pmr::vector<char> chars{&memory};
    chars.reserve(20);

    for (char c = 'a'; c <= 'e'; c++){
        chars.emplace_back(c);
    }

    std::cout << buffer << "\n";
```

PMR: **RESERVE OUTPUT**

```
EEEEEEEEEEEEEEEEEEE
abcdeEEEEEEEEEEEEEE
```

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

PMR: USING A PMR COLLECTION OF PMR COLLECTION

Using a collection of PMR collection
is a very interesting concept, as the children
of the main collection will ask for
allocation space from the father collection.

PMR: USING A PMR COLLECTION OF PMR COLLECTION

```
void Printer(char* buffer, std::string_view title){
    std::cout << title << "\n";
    auto buff = std::string_view(buffer, 256);
    for (const auto& ch : buff){
        if (ch >= ' ' ? ch : '_'){
            std::cout << ch;
        }
    }

    std::cout << " !endof buffer! \n ";
}
```

```
int main() {
    char buffer[256] = {};
    std::fill_n(std::begin(buffer), std::size(buffer) - 1, '#');


    Printer(buffer, "Empty buffer:");

    std::pmr::monotonic_buffer_resource pool{std::data(buffer), std::size(buffer)};
    std::pmr::vector<std::pmr::string> vec{ &pool };
    vec.reserve(4);

    vec.emplace_back("Hello Darkness");
    vec.emplace_back("My Old Friend");
    Printer(buffer, "Tree string in:");

    vec.emplace_back("This is a longer string so what will happen now ?????  well it will not");
    Printer(buffer, "Long string:");


    vec.emplace_back("1234");
    Printer(buffer, "saved on buffer again");
```

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

PMR: USING A PMR COLLECTION OF PMR COLLECTION

```
Empty buffer:
########################################################################################################################################
######################################################################################################################### !endof buffer!
Two string in:
���□x���□Hello Darkness# ���□����□ My Old
Friend##################################################################################################################################
################################################ !endof buffer!
Long string:
���□x���□Hello Darkness# ���□����□ My Old Friend## ���□���□GG################################################This is a
longer string so what will happen now ????? well it will not####################### !endof buffer!
saved on buffer again
���□x���□Hello Darkness# ���□����□ My Old Friend## ���□���□GG######## ���□����□1234###########This is a longer string
so what will happen now ????? well it will not####################### !endof buffer!
```

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

PMR: USING A PMR COLLECTION OF PMR COLLECTION – USING REGULAR STRING

```
Empty buffer:
######################################################################################################################################
######################################################################################################################### !endof
buffer!
Two string in:
p�k+�Hello Darkness#��k+� My Old
Friend################################################################################################################################
################################################################# !endof buffer!
Long string:
p�k+�Hello Darkness#��k+� My Old Friend##�>
GG####################################################################################################################################
################################### !endof buffer!
saved on buffer again
p�k+�Hello Darkness#��k+� My Old Friend##�>
GG#######��k+�1234#################################################################################################################
######################### !endof buffer!
```

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

FEATURES THAT DIDN’T MAKE THE CUT

FEATURES THAT DIDN’T MAKE THE CUT

- File System
- Common_type
- Conjunction, disjoint and negation
- Lambda Inheritance
- Apply
- Invoke
- Many more

QUESTIONS

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

THANK YOU FOR LISTENING

Alex Dathskovsky

054-7685001

calebxyz@gmail.com

BONUS!

LAMBDA INHERITANCE

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

LAMBDA INHERITANCE

Examples:

what happens if we want to combine 2 lambdas with different signatures?

```
auto l1 = [](){return  4;};
auto l2 = [](int i) {return 10*i; };
```

We want to call combined(int) or combined()

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

LAMBDA INHERITANCE

Examples:

Don’t forget Lambdas are objects too!

```
template <typename L1, typename L2>
struct CombinedLambda : public L1, public L2
{
    CombinedLambda(L1 l1, L2 l2): L1(std::move(l1)), L2(std::move(l2)) {}
    using L1::operator();
    using L2::operator();
};

template <typename L1, typename L2>
auto make_combined(L1&& l1, L2&& l2)
{
    return CombinedLambda<std::decay_t<L1>, std::decay_t<L2>>
            (std::forward<L1>(l1), std::forward<L2>(l2));
}
```

LAMBDA INHERITANCE

Examples:

Don’t forget Lambdas are objects too!

```
auto l1 = [](){return  4;};
auto l2 = [](int i) {return 10*i; };

auto combined = make_combined(std::move(l1), std::move(l2));
std::cout << combined() << "\n";
std::cout << combined(4) << "\n";
```

Output:
4
40

- This was our solution in C++14

LAMBDA INHERITANCE

C++ 17 Solution

```cpp
#include <atomic>
#include <thread>
#include <random>
#include <chrono>
#include <map>
#include <utility>
#include <tuple>
#include <type_traits>
#include <memory>

template <typename... L>
struct Merged : L...
{
    template <typename... T>
    Merged(T&&... t): L(std::forward<T>(t))... {};

    using L::operator()...;
};


template <typename... T>
Merged(T...) -> Merged<std::decay_t<T>...>;

int main()
{
    auto l1 = [](){return  4;};
    auto l2 = [](int i) {return 10*i; };

    Merged merged(l1, l2, [p = std::make_unique<double>(5.)](

    return merged(1.);
}
```

A▾ 11010 .LX0: .text // \s+ Intel Demangle

```asm
main: # @main
  mov eax, 5
  ret
```

Alex Dathshkovsky | 054-7685001 | calebxyz@gmail.com

WHY SHOULD WE EVERY USE THIS MONSTROSITY

Alex Dathshkovsky | 054-7685001 | calebxyz@gmail.com

WHY SHOULD WE EVERY USE THIS MONSTROSITY

```cpp
#include <utility>
#include <array>
#include <variant>
#include <algorithm>

template <typename... L>
struct Visitor : L...
{
    template <typename... T>
    Visitor(T&&... t): L(std::forward<T>(t))...
    {}

    using L::operator()...;
};

//sadly gcc didnt implement the standard as it was suggested
//and you need very specific deduction guides
template <typename U, typename... Us>
Visitor(U u, Us... us) -> Visitor<U, Us...>;
```

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com

WHY SHOULD WE EVERY USE THIS MONSTROSITY

```cpp
int main()
{
    std::array<std::variant<bool, float, int, double>, 8>
    varArr{0.1f, 2, 3, 0.3, 1.5f, true, 0.4, false};

    int sumInts{0};
    float sumFloats{0};
    double deductDoubles{1};
    bool aggregateBools{true};

    auto visitor = Visitor{
            [&sumInts](int i) { sumInts += i; },
            [&sumFloats](float f) { sumFloats += f; },
            [&deductDoubles](double d) { deductDoubles -= d; },
            [&aggregateBools](bool b) { aggregateBools &= b; }
    };

    std::for_each(varArr.begin(), varArr.end(),
            [&visitor](const auto& var){std::visit(visitor, var);});

    return sumInts;
}
```

Alex Dathskovsky | 054-7685001 | calebxyz@gmail.com