A Perfect Move Forward

Author: Alex Dathskovsky calebxyz@gmail.com

Introduction

Few Words About Me

- Developer with experience of more than 14 years.
- Performance expert
- Linux environment expert
- Architect for large scale systems

Agenda

- File System Lib
- C++17 new features
- Perfect Forwarding
- Most important thing is feel free to ask questions during the talk.

C++17 File System

FileSystem Lib

- The new filesystem library is based on boost::filesystem. Some of its components are optional. That means not all functionality of std::filesytem is available on each implementation of the filesystem library. For example, FAT-32 does not support symbolic links.

FileSystem Lib

- The library is based on the three concepts file:
  - file name and path
  - Files can be directories, hard links, symbolic links or regular files
  - Paths can be absolute or relative

FileSystem Lib

- How we used to look at filesystems:
  - Nothing was implemented
  - Used old c functions and strings (getpwd…)
  - The only real implementation that was available is File* Filestream…

FileSystem Lib

- What can we do now ?
  - create and remove directories
  - iterate over directories
  - check properties of files
  - Traversing a path, even recursively

FileSystem Lib

- Core Objects
  - path object
  - directory_entry
  - Directory iterator

FileSystem Lib

- Supportive functions
  - Getting info about paths
  - File manipulations
  - Write time tags
  - Permissions
  - Space On Drive
  - File Size

FileSystem Lib

▶Working With Paths

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

namespace fs = std::filesystem;

fs::path pathToShow(/* ... */);
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";
```

FileSystem Lib

- Working With Paths , program output for c:/MyDir/txt.ini

exists() = 1

root_name() = C:

root_path() = C:\

relative_path() = MyDir/txt.ini

parent_path() = C:\MyDir

filename() = txt.ini

stem() = txt

extension() = .ini

FileSystem Lib

- Working With Paths , iteration over path elements

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

namespace fs = std::filesystem;

fs::path pathToShow("C:/MyDir");

int i = 0;
for (auto&& part : pathToShow)
{
    cout << "path part: " << i++ << " = " << part << "\n";
}
```

FileSystem Lib

Creating Paths

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

namespace fs = std::filesystem;

fs::path p1("C:\\temp");
p1 /= "user";
p1 /= "data";
cout << p1 << "\n";

fs::path p2("C:\\temp\\");
p2 += "user";
p2 += "data";
cout << p2 << "\n";
```

FileSystem Lib

- Creating Paths, Output
- P1- c:\temp\user\data
- P2- c:\temp\userdata

FileSystem Lib

Checking File Size

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

namespace fs = std::filesystem;

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);
}
```

FileSystem Lib

▶Last Write TT

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

namespace fs = std::filesystem;

auto timeEntry = fs::last_write_time(entry);
time_t cftime = chrono::system_clock::to_time_t(timeEntry);
cout << std::asctime(std::localtime(&cftime));
```

FileSystem Lib

Permissions

```
void printPerms(fs::perms perm)
{
  std::cout << ((perm & fs::perms::owner_read) !=
              fs::perms::none ? "r" : "-")
              << ((perm & fs::perms::owner_write) !=
              fs::perms::none ? "w" : "-")
              << ((perm & fs::perms::owner_exec) !=
              fs::perms::none ? "x" : "-")
              << ((perm & fs::perms::group_read) !=
              fs::perms::none ? "r" : "-")
              << ((perm & fs::perms::group_write) !=
              fs::perms::none ? "w" : "-")
              << ((perm & fs::perms::group_exec) !=
              fs::perms::none ? "x" : "-")
              << ((perm & fs::perms::others_read) !=
              fs::perms::none ? "r" : "-")
              << ((perm & fs::perms::others_write) !=
              fs::perms::none ? "w" : "-")
              << ((perm & fs::perms::others_exec) !=
              fs::perms::none ? "x" : "-")
              << std::endl;
}

printPerms(fs::status("someFile.txt").permissions());
```

FileSystem Lib

- Space

```
fs::space_info root = fs::space("/");
    fs::space_info usr = fs::space("/usr");

    std::cout << ".          Capacity       Free       Available\n"
              << "/     " << root.capacity << "    "
              << root.free << "    " << root.available << "\n"
              << "usr   "  << usr.capacity << "    "
              << usr.free << "    " << usr.available;
```

FileSystem Lib

- Iteration on dir

```
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";
        }
    }
}
```

FileSystem Lib

- Computing file size async way

```
std::vector<std::filesystem::path> paths;

std::filesystem::recursive_directory_iterator dirpos{ root };
std::copy(begin(dirpos), end(dirpos), std::back_inserter(paths));
```

FileSystem Lib

- Computing file size async way

```
template <typename Policy>
uintmax_t ComputeTotalFileSize(const std::vector<std::filesystem::path>& paths,
                               Policy policy)
{
    return std::transform_reduce(
        policy,
        paths.cbegin(), paths.cend(),            // range
        std::uintmax_t{ 0 },                    // initial value
        std::plus<>(),                            // accumulate ...
        [](const std::filesystem::path& p) {    //  file size if regular file
        return is_regular_file(p) ? file_size(p)
            : std::uintmax_t{ 0 };
    });
}
```

FileSystem Lib

- Computing file size async way

```
start = std::chrono::steady_clock::now();
uintmax_t FinalSize = 0;
if (executionPolicyMode)
FinalSize = ComputeTotalFileSize(paths, std::execution::par);
else
FinalSize = ComputeTotalFileSize(paths, std::execution::seq);

PrintTiming("computing the sizes", start);

std::cout << "size of all " << paths.size()
        << " regular files: " << FinalSize/1024 << " kbytes\n";
return 0;
```

FileSystem Lib

- Computing file size async way
  - On my system I got :
    - PAR : 0.623 ms
    - SEQ: 1.12564 ms

C++17 Features

String View

- 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

- std::string_view   std::basic_string_view<char>
- std::wstring_view   std::basic_string_view<wchar_t>
- std::u16string_view   std::basic_string_view<char16_t>
- std::u32string_view   std::basic_string_view<char32_t>

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

- look at the next code

```cpp
#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;
}
```

String View

- 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

- Possible solution
  - we could cr … yle strings, but then we lose the
  - we now hav … stensibly do the same thing
  - What happe … as Qt's Qstring?
  - What happe … C-style string or a Qstring?

```
bool compa ... char* s2);
bool compa ... QString& s2);
bool compa ... ring& s2);
bool compa ... 2);
bool compa ... & s2);
bool compa ... :string& s2);
bool compa ... * s2);
bool compare(const QString& s1, const QString& s2);
```

String View

- string_view is our real solution

```
#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

- 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"

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;
}
```

String View

- output

[allocating 39 bytes]

"will this work" does not match "this work"

Deduction Guides

Examples:

will this work?

```
Void func() {} ;

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

The answer is no.

Deduction Guides

But now we can teach the compiler what to do

```
namespace std
{
template<typename Ret, typename… Args> function(Ret, (*)(Args…)) -> function<Ret(Args…)>;
}
```

Deduction Guides

Before c++17

```cpp
#include <functional>

void test1(){};

void test2(int x, double y, std::string z){};

int main() {
  // i can write
  std::function x = []() { test1(); };

  // but if i want another function like test 2?
  std::function y = [](int x, double y, std::string z) { test2(x, y, z); };

  // what about other functions
}
```

Deduction Guides

```cpp
#include <functional>

namespace std
{
    template <typename Ret, typename... Arg> function(Ret(*)(Arg...)) ->
    function<Ret (Arg...)>;

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

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

class TestClass
{
    public:
    void memFun1() {};
    void memFun2() const {};
};

void test1(){};

void test2(int x, double y, std::string z) {};

```

Deduction Guides

```
int main()
{
    std::function f1(test1);
    std::function f2(test2);
    std::function f3(&TestClass::memFun1);
    std::function f4(&TestClass::memFun2);
    TestClass c;

    f1();
    f2(1, 2.1, "XYZ");
    f3(c);
    f4(c);
```

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()

Lambda Inheritance

Examples:

Don't forget Lambdas are objects too.

```cpp
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

Lambda Inheritance

- Lambda Inheritance c++17:

Examples:

```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.);
}
```

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

Fold Expressions

- Fold Expressions c++14:

  Examples:

```
template <typename... T>
auto mult(T... t)
{
    typename std::common_type<T...>::type res{1};
    (void)std::initializer_list<int>{(res *= t, 0)...};

    return res;
}
```

Fold Expressions

- Fold Expressions c++17:

  Examples:

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

```
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);
}
```

```
template <typename... T>
auto somthing2(T... t)
{

    return (t() + ...);
}
```

Template with auto

- 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>;

Template with auto

Examples:

C++17 :

```
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>;

C++14: template <typename T, T ... vs> struct Cxx14HomogenousValueList {};
using MyList3 = Cxx14HomogenousValueList<int, 1, 2, 3>; //not to bad
template <typename ... ValueTypes> struct Cxx14HeterogenousValueList {}; using
MyList4 = Cxx14HeterogenousValueList<constant<int, 42>, constant<char, 'X'> >;

        //this is much worst that in c++17
```

Invoke

- std::invoke

interface to

14

Invoke

- So what is it good for ?

Learn by example:

struct S { int j = 5;

Int doSomthing(const int i)

{return j*j+i};};

Invoke

```
S s;
cout:: << s.doSomthing(3) //print 28

auto fp = &S::doSomthing; //int (S::*(int))
(s.*fp)(4); //prints 29
```

Invoke

- So what is it good for ?

Learn by example:

```
struct S { int j = 5;
Int doSomthing(const int i)
{return j*j+i};
Int doSomthing2(const int i)
{return j+j+i};};
```

Invoke

```
Int (S::*fp2)(int) = nullptr;
If (expression == true)
    fp2 = &S::doSomthing2;
Else
    fp2 = &S::doSomthing;

expression = true
cout << (s.*fp2)(1) // prints 11
```

Invoke

Instead of this monstrosity we can use the invoke

Std::invoke(&S::doSomthing2, s, 1); //will calc 11

You can even do this

Std::invoke(&S::j, s) //returns 5

Optional

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

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

```
std::cout << "create(false) returned " <<
create(false).value_or("empty");
```

Optional

- nullopt
  - Just like null pointer but with a different type
  - Creates an empty optional value

```
auto create2(bool b) { return b ?
std::optional<std::string>{“Precision"} :
std::nullopt; }
```

```
if (auto str = create2(true)) {
std::cout << “what do we do" << *str;}
```

Optional

- More examples

```
std::optional<std::string>
opt1(std::in_place, "C++17");
std::optional<std::string>
opt2(std::in_place,5, 'C');
std::optional<std::string>
opt3(std::in_place,
{'C', '+', '+', '1', '7'});
std::optional<std::string> opt4(opt3);
```

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);
```

Any

Checking the type of any

```
std::any a = 1;
const std::type_info &ti = a.type();
cout << ti.name();
```

Variant

- 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

```
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;
```

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<char, long, float, int, double, long long>> vecVariant =
{5, '2', 5.4, 100ll, 2011l, 3.5f, 2017};

for (auto& v: vecVariant){ std::visit([](auto&& arg){std::cout << arg << " ";}, v);}
```

Variant

- Common_type
  - Determines the common type among all types T

    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<char, long, float, int,
   double, long long>> vecVariant =
{5, '2', 5.4, 100ll, 2011l, 3.5f, 2017};
std::common_type<char, long, float, int, double,
   long long>::type res{};

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

Const Expressions and If

- Will This code Compile ?
  - template <typename T>
    auto calcSomthing(const T& t){
    if (std::is_integral<T>::value)
    return t++;
    else
    return t+0.5;}

  Answer : No

Const Expressions and If

```
template <typename T>
    auto calcSomthing(const T& t){
     if constexpr (std::is_integral<T>::value)
     return t++;
     else
     return t+0.5;}

Cout << calcSomthing(1) ; //prints 2
Cout << calcSomthing(0.1); //prints 0.6
```

Multiple Namespaces

before c++17

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

Multiple Namespaces

Multiple name spaces

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

Perfect Forwarding

Forwarding

- Motivation by Example
  - The next code segments show an example of a make function that takes an argument and passes it.
    - Example 1 doesn't work on non-copyables and adds another copy for regular objects ☹

```
template <typename T, typename Arg> my_smart_obj<T>
make_smart_obj(Arg arg)
{
    return my_smart_obj<T>(new T(arg));
}
```

- Example 2 doesn't work for rvalues ☹ ☹

```
template <typename T, typename Arg> my_smart_obj<T>
make_smart_obj(Arg& arg)
{
    return my_smart_obj<T>(new T(arg));
}
```

Forwarding

- The solution can be adding two overloads but..
- what if we have two arguments?

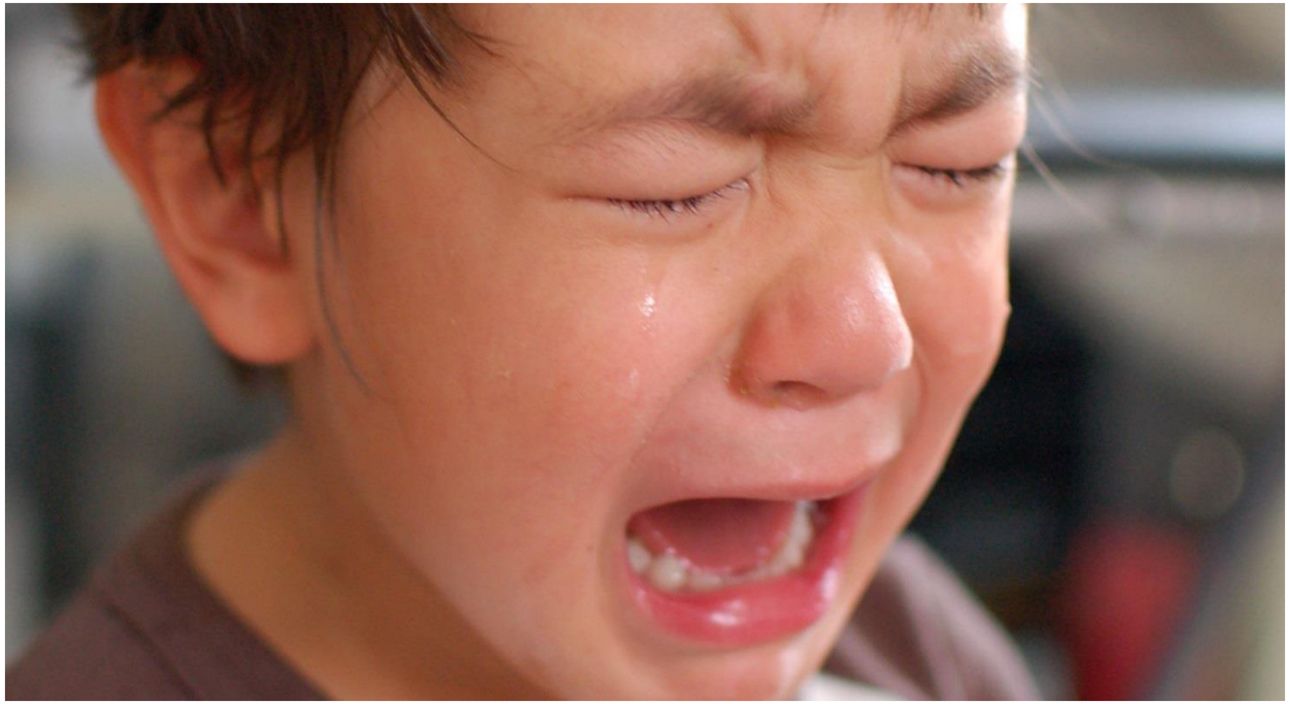
```
Void func(T t, U u);
Void func(T& t, U u);
Void func(T t, U& u);
Void func(T& t, U& u);
```

Forwarding

```
Void func(T t, U u, W w);
Void func(T& t, U u, W w);
Void func(T t, U& u, W w);
Void func(T t, U  u, W& w);
Void func(T& t, U&  u, W w);
Void func(T& t, U  u, W& w);
Void func(T t, U&  u, W& w);
Void func(T& t, U&  u, W& w);
```

Forwarding

- And if it is 10 arguments?



Forwarding

- Help me Universal Reference
  - If we are using templates the && receives a different role - it is called **universal reference.**
  - Reference collapsing rules:
    - X& & becomes X&
    - X& && becomes X&
    - X&& & becomes X&
    - X&& && becomes X&&
  - Example:

```
template <typename T> void Foo(T&& t);
X x;
Foo(x);   // resolves to Foo<X&>(x); x is X&
Foo(X()); // resolves to Foo<X>(x); x is X&&
Foo(std::move(x)); // resolves to Foo<X>(x); x is X&&
```

Forwarding

- UR Pitfalls
  - Will this work?

```
template<typename T>
class MyClass
{
public:
    void func(T&& t) { //do something with t };
};

int x = 25;
MyClass<int> myClass;
myClass.func(x);
```

Forwarding

- Meet std::forward

  For rvalues, T will be plain type , t is T&&, T&& is returned

```
template <typename T>
T&& forward(typename std::remove_reference<T>::type&& t)        {
   return static_cast<T&&>(t);
}
```

  For lvalues, T will be a reference, t is T&, T&& &, T& is returned

```
template <typename T>
T&& forward(typename std::remove_reference<T>::type& t)
{
   return static_cast<T&&>(t);
}
```

Forwarding

- Perfect Forwarding
  - Now our make_smart_obj can forward the argument.
  - No need for crazy overloading ☺☺

```
template <typename T, typename Arg> my_smart_obj<T>
make_smart(Arg&& arg) {
  return my_smart_ptr<T>(new T(std::forward<Arg>(arg)));
}
```

  - Important: the std::forward function doesn’t perform auto deduction!

Forwarding

- Universal reference in other context
  - auto&& is also a way to define a universal reference

    auto&& rv = std::move(val); //rvalue reference

    auto&& lv = val; //lvalue reference

    for (auto&& elem : container) //depends on the iterator.
  - extra helpful for generic lambdas.

    auto lambda = [](auto&& fun, auto&&... args) {

    //do something before invoking fun

    forward<decltype(fun)>(fun)(forward<decltype(args)>(args
)...);

    //do something after invoking fun };

Forwarding

- Parameter Passing Guidance
  - If function needs to modify the parameter use X&.
  - If the function only observes the parameter use X const &
  - If the function will always copy the parameter
    - If X non-copyable take X&&
    - If X non-moveable take X const&
    - If X is copyable and moveable take X or provide 2 overloads
    - The preferred way is universal reference

Forwarding

- Multiple Parameters
  - Overloading on T const& and T&& is unfeasible
  - Solution: use urefs and static asserts

```
template <typename T1, typename T2>
void create(T1&& t1, T2&& t2)
{
   static_assert(std::is_constructible<someClass1,T1>::value,...);
   static_assert(std::is_constructible<someClass2,T2>::value,...);
   auto x = someClass1(std::forward<T1>(t1));
   auto y = someClass2(std::forward<T2>(t2));
}
```

Forwarding

- Multiple Parameters
  - Alternative approach

```
template <typename T1, typename T2>
std::enable_if<
      std::is_constructible<someClass1, T1>::value &&
      std::is_constructible<someClass2, T2>::value >
create(T1&& t1, T2&& t2)
{ …}
```

Forwarding

- More bad news!
  - Universal reference shadow everything.

```
class Example {
    std::string m_name;
public:
    template <typename T> explicit Example(T&& name) :
        m_name(std::forward<T>(name)){};
    Example(const Example& other);
};
```

  - So what will happen here ?

```
Example e1(“Alex”s);
Example e2(e1);
```

Forwarding

- Solution
  - Use enable_if to avoid clashes with other methods.

```
class Example {
   std::string m_name;
public:
   template <typename T ,
   typename = std::enable_if<
              !std::is_base_of<someclass,
               std::decay<T>::value>>>
   explicit Example(T&& name) :
      m_name(std::forward<T>(name)){};
};
```

Forwarding

- More confusing usage from C++ but its helpful
  - Very helpful for overloading.

char* dup_chars() & // this obj is lvalue we can duplicate

char* dup_chars() && // this is a rvalue just move

Thank You