C++ workshop 2018

2 Oct 2018

Arun Bharadwaj

Plan for the day

- 9:30 start
- 12:00 : lunch break (45 mins?)
- 12:45 : continue
- 15:00 : break (20 mins)
- 15:20 : continue
- 16:30 : close

Topics

- Day 1
  - Refresh some fundamentals
  - Move semantics
  - Type deduction
  - Function templates
  - Universal reference
  - Forward semantics
  - ~~Reference collapsing~~
  - Lambda expression
- Day 2
  - Copy-swap idiom
  - Function template specialization/overloading
  - ~~Constness~~
  - Type-traits
  - Refactor some code in IDES2

Some fundamentals

Value categories

- lvalue : if you can take its address, its lvalue
- rvalue: if (above condition) not, its rvalue

  Example –

```
int foo(){return 42;}

{
 int a = 10; // OK a -> lvalue
 int b = foo(); // OK b -> lvalue, foo() is an rvalue.
 int c = a + 2; // OK c -> lvalue, expression a+2 is rvalue. Cannot take the address of (a+2)
 foo() = a; // error! Visual studio -> cannot assign to rvalue
                // foo() is rvalue
}
```

Value categories contd…

- All parameters to a function are lvalues.

```
int foo(T1 a, T2 b, T3 c){}

a, b and c are lvalues
T1, T2 and T3 are types
```

References

- lvalue reference

```
int a = 10;
int& ra = a; // a is lvalue
```

- rvalue reference

```
int&& rb = a + 2; // a+2 is rvalue
```

- What about this?

```
int& rc = a + 2; // error! You cannot bind rvalue to lvalue reference
int&& rd = a; // error! You cannot bind lvalue to rvalue reference
```

Rule - lvalue binds to lvalue reference, and rvalue binds to rvalue reference

Related rule

Error case –

```
int& rc = a + 2; // error! You cannot bind rvalue to lvalue reference
int&& rd = a; // error! You cannot bind lvalue to rvalue reference
```

But -

```
const int& b = a + 2; // OK! But why?
```

Related rule –

binding a temporary object (rvalue) to a lvalue reference *to const* lengthens the lifetime of the temporary to the lifetime of the reference itself

Consequently, you can do something like this –

```
const int& b = foo(); // lifetime of temporary object returned by foo now extends to lifetime of b.
                      // therefore avoiding having to make a copy
```

Code example

```
void foo(int& a)
{
   ...
}
void foo(int&& a)
{
   ...
}
void foo(const int& a)
{
   ...
}

int bar(){ return 42;}
const int bar2(){ return 44;}

{
    int x = 10;
    foo(x); // foo(int&)
    foo(bar()); // foo(int&&)
    foo(4); // foo(int&&)
    const int y = 33;
    foo(y); // foo(const int&)
    std::getchar();
}
```

Move semantics

need for
speed

Move semantics : std::move + move constructors

syntax

- Copy constructor

```
foo(const foo& other){}
```

- Move constructor

```
foo(foo&& other){}
```

- Copy assignment operator

```
foo& operator=(const foo& other){}
```

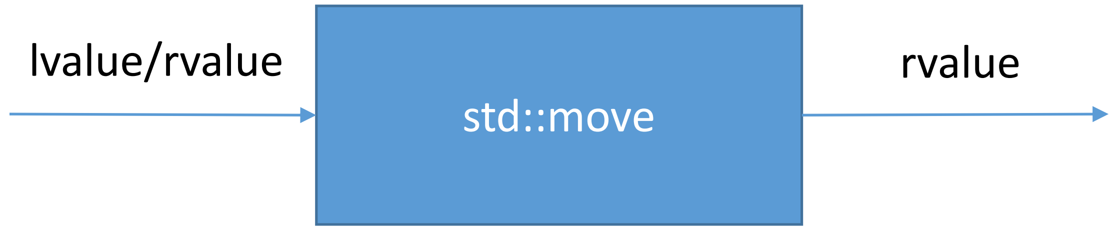
- Move assignment operator

```
foo& operator=(foo&& other){}
```

{ Code example }

What does std::move do?

- Takes an argument, casts it to be an rvalue of type rvalue reference



Code example

Previous Code example

Exercise time

Exercise 1

Step 1. Implement a class foo that has one data member of type int. Provide constructor, copy constructor, move constructor, copy assignment operator and move assignment operator. Make sure to print out the type of constructor within each of them.

Step 2. Implement a main function. In the main function, declare a vector<foo> and reserve the size to be 10.

Step 3. In a loop of 10 iterations, do the following –

```
foo f(i);
vec_foo.push_back(f);
```

Step 4. Run the program. What do you see?

Step 5. Now replace of vec_foo.push_back(f) with vec_foo.push_back(std::move(f)). What do you see?

Step 6. Now get rid of local variable f and replace vec_foo.push_back(std::move(f)) with vec_foo.push_back(foo(i)). What do you see?

{ Code example }

Exercise 2

Continuing from where we left off…

Step 1. Add another loop of 10 iterations, and run the same push_back statement. vec_foo.push_back(foo(i))

Step 2. Run it. What do you see?

{ Code example }

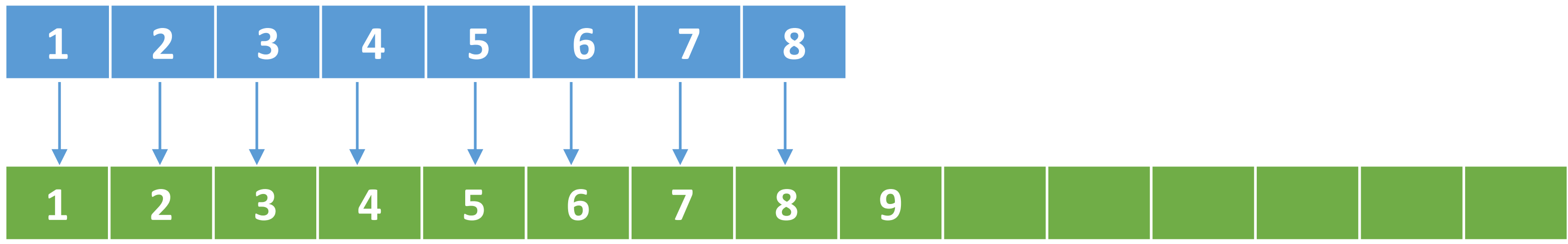
Exception Safety - briefly

- Nothrow (or nofail) exception guarantee -- the function never throws exceptions.
- Strong exception guarantee -- If the function throws an exception, the state of the program is rolled back to the state just before the function call.
- Basic exception guarantee -- If the function throws an exception, the program is in a valid state. It may require cleanup, but all invariants are intact.
- No exception guarantee -- If the function throws an exception, the program may not be in a valid state: resource leaks, memory corruption, or other invariant-destroying errors may have occurred.

Source - https://en.cppreference.com/w/cpp/language/exceptions

vector.push_back() – exception safety 1

- C++ 98 : push_back() offered strong exception safety gurantee
  - vec<int> :
  - push_back(9):
    - Step 1 – allocate bigger chunk of memory
    - Step 2 – COPY each element
    - Step 3 – push_back(9)



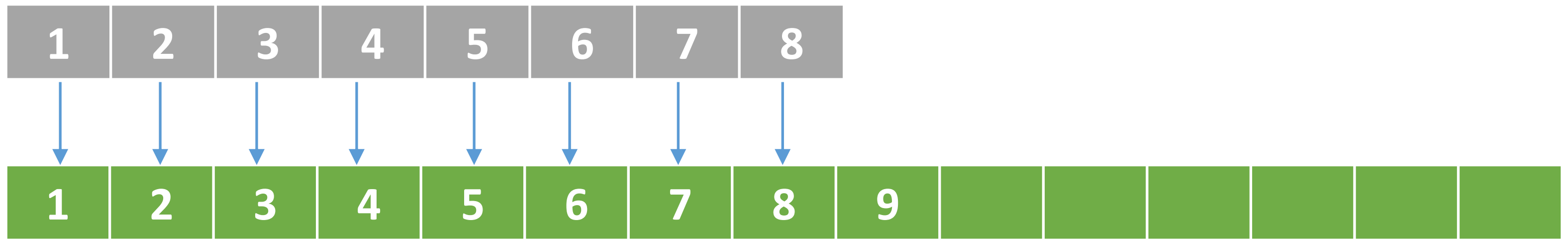
vector.push_back() – exception safety 2

- If an exception is thrown (which can be due to Allocator::allocate() or element copy/move constructor/assignment), this function has no effect (strong exception guarantee).

  NOTE - If T's move constructor is not noexcept and T is not CopyInsertable into *this, vector will use the throwing move constructor. If it throws, the guarantee is waived and the effects are unspecified.

  Source - https://en.cppreference.com/w/cpp/container/vector/push_back

- vec<int>:
- push_back(9):



  - Step 1 – allocate bigger chunk of memory
  - Step 2 – MOVE iff move constructor is noexcept, else COPY each element
  - Step 3 – push_back(9)

Exercise 3

Continuing from where we left off

Step 1. Add “noexcept” exception specification to your move constructor…so your move constructor should now look like this –

```
foo(foo&& other) noexcept
```

Step 2. Run the code. What do you see?

{ Code example }

Wrong move

- When is using std::move the wrong thing to do!

{ Code example }

- So beware of copy elision and return value optimization.

Right move

- So when is using std::move the right thing to do!

{ Code example }

- When the returning object itself is an rvalue…use std::move!
- When the parameter was rvalue reference, and you want to operate on that parameter, use std::move

Function Template

- Function template defines a family of functions.
- Compiler generates code ONLY when the function template is instantiated (right form is invoked)

```
int mul(int a, int b) { return a * b;}
float mul(float a, float b) { return a * b;}
float mul(int a , float b) { return a * b;}
```

OR

```
template<typename TA, typename TB>
auto mul(TA a, TB b) -> decltype(a*b)
{
    return a * b;
}
```

{ Really Cool Tool }

Benefits of templates

- Use compiler to generate code for all **required** data types instead of overloading your functions/classes manually
- One API for all data types
- No wasted APIs – **Generate** what you need…not **WRITE** what you MAY need
- No additional runtime cost
- No additional size cost.

Type deduction

- Template
- auto
- decltype
- decltype(auto) (this is an extension of auto)

Examples where type deduction happen - 1

- Templates

```
template <typename T>
void foo(ParamType param){}
```

- auto

  - Function return types

```
auto foo() {return 1 + 2;} // return value type deduction
```

  - auto variable initializations

```
auto var = foo(); // var type deduction
auto var = 1 + 2; // var type deduction
auto var = {1,2,3,4}; // var type deduction
```

Examples where type deduction happens - 2

- Lambda

```
    auto glambda = [x = x] (auto a, auto b) { return a < b; };
```

- decltype

```
template<typename TA, typename TB>
auto mul(TA a, TB b) -> decltype(a*b)
{
   return a * b;
}
```

Type Deduction - template

```
template<typename T>
void foo(ParamType param);

foo(expr); // instantiate the template
```

- Two **separate** type deduction –
  - T
  - ParamType – generally different from T, for example const T&
- Both these type deductions depend on the type of **expr**

Type Deduction - auto

```
template<typename T>
void foo(ParamType param);
```

```
auto⌠A⌡ bar = expr; // example auto& bar = expr
```

- auto : T
- auto⌠A⌡ : ParamType

Rules for type deduction

template<typename T> foo(T param);
auto v = input;

Input	T auto	ParamType auto∫Aʅ
int	int	int
const int	int	int
const int&	int	int

template<typename T> foo(T& param);
auto& v = input;

Input	T auto	ParamType auto∫Aʅ
int	int	int&
const int	int	const int&
const int&	int	const int&

template<typename T> foo(const T& param);
const auto& v = input;

Input	T auto	ParamType auto∫Aʅ
int	int	const int&
const int	int	const int&
const int&	int	const int&

* Reference of a reference is not a valid code (only compilers can do that)

my_typeid and really cool website

For template use this really cool website – https://godbolt.org/

For auto use this helper function →

```
#include <typeinfo>

template <class T>
std::string type_name() {
    using TR = typename std::remove_reference<T>::type;

    std::string r = typeid(TR).name();
    if (std::is_const<TR>::value)
        r += " const";
    if (std::is_volatile<TR>::value)
        r += " volatile";
    if (std::is_lvalue_reference<T>::value)
        r += "&";
    else if (std::is_rvalue_reference<T>::value)
        r += "&&";
    return r;
}

#define my_typeid(var) type_name<decltype(var)>()

-------------------- usage -----------------------
auto a = 1 + 2;
std::cout<<my_typeid(a)<<std::endl;
```

Universal reference

Recap

int a = 20; // a -> lvalue

int& b = a; // lvalue reference

int&& c = 30; // rvalue reference

```
int&& d = a; // error. Only rvalues can bind to rvalue reference

foo(int&& f);
foo(a); // error again. Only rvalues can bind to rvalue reference.
```

How about this?

```
auto&& d = a; // a is lvalue

template<typename T>
foo(T&& f);
foo(a);
```

Universal reference - 2

And this?

```
const int a = 10;
auto&& d = a; // a is const lvalue

template<typename T>
foo(T&& f);
foo(a);
```

And this?

```
int& b = a;
auto&& d = b; // b is lvalue reference

template<typename T>
foo(T&& f);
foo(d);
```

Universal reference - 3

And this?

```
const int& b = a;
auto&& d = b; // b is lvalue reference to const int

template<typename T>
foo(T&& f);
foo(d);
```

And this?

```
int&& b = 10;
auto&& d = b; // b is rvalue reference. Try auto&& e = 10

template<typename T>
foo(T&& f);
foo(d);
```

Rules for type deduction - complete

template<typename T> foo(T param);
auto v = input;

Input	T auto	ParamType autoʃAʅ
int	int	int
const int	int	int
const int&	int	int

template<typename T> foo(T& param);
auto& v = input;

Input	T auto	ParamType autoʃAʅ
int	int	int&
const int	int	const int&
const int&	int	const int&

template<typename T> foo(const T& param);
const auto& v = input;

Input	T auto	ParamType autoʃAʅ
int	int	const int&
const int	int	const int&
const int&	int	const int&

template<typename T> foo(T&& param);
auto&& v = input;

Input	T auto	ParamType autoʃAʅ
int	int&	int&
const int	const int&	const int&
const int&	const int&	const int&
temp(int) (rvalue)	int	int&&

* Reference of a reference is not a valid code (only compilers can do that)

References in one slide

```
int x = 42; // lvalue
```

```
int& y = x; // lvalue reference
```

```
int&& z = 7 * 6; // rvalue reference
```

```
auto&& u = x and y and z and 6 * 7; // universal reference

template<typename T>
RetType foo(T&& f){} // universal reference
```

Until now

- Refresh some fundamentals
- Move semantics
- Type deduction
- Function templates
- Universal reference

Move – again!

{ Code Example }

template<typename T> foo(T&& param);
auto&& v = input;

Input	T auto	ParamType auto∫A⌉
int	int&	int&
const int	const int&	const int&
const int&	const int&	const int&
temp(int) (rvalue)	int	int&&

There is a solution – perfect forwarding.

std::move -> unconditional returns value reference
std::forward -> conditional returns either rvalue reference or lvalue reference

To Move or To Forward?

`Rvalue reference? Use std::move`

`Universal reference? Use std::forward<T>`

`Example`

```
// rvalue reference
foo(int&& f) { std::move(f); }

// universal reference
template<typename T>
void foo(T&& x) { std::forward<T>(x); }
```

Lambda - syntax

```
[captures](params) -> ret_type // explicit return type
{
    body
}
```

```
[captures](params)  // deduced return type
{
    body
}
```

```
[captures] // no params needed
{
    body
}
```

Capture modes

- Default by-value : `[=](){} // get all the variables in the scope by value`
- Default by-reference : `[&](){} // get all the variables in scope by reference`
- Explicit by-value : `[x](){} // get x by value`
- Explicit by-reference: `[&x](){} // take the reference of x into the closure`
- Init capture: C++14: `[x=x](){} // creates a separate variable within lambda`

```
[x=std::move(x)](){} // move x into closure
```

* Prefer Init capture

Type Deduction in Lambda Capture

- Capture by reference – Type deduction same as that for templates

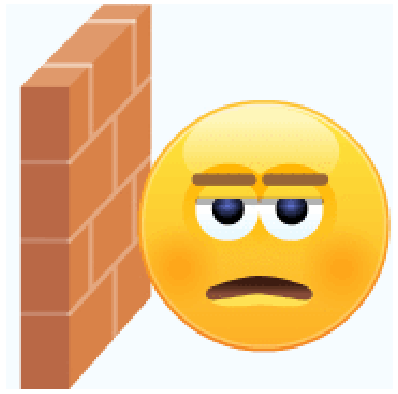
- Init capture – Type deduction same as that for auto (which is almost same as templates but one of two minor exceptions – not too important for now)

- By Value – Type deduction same as that for template except for one major difference: and a bit un-intuitive

Type deduction quirk in Lambda – by value

```
int main()
{
    int x = 10;
    auto foo = [=] ()
    {
        x = 20;
    };

    foo();
}
```



```
error C3491: 'x': a by copy capture cannot be modified in a non-mutable lambda
```

Type deduction quirk in Lambda – by value 2

```
int x = 10;
auto foo = [=] ()
{
    x = 20;
    std::cout << x << std::endl;
};
```

```
class compiler_generated_name
{
public:
    void operator()() const { cx = 20; }
private:
    int cx;
};
```

const member function cannot modify data member

Type deduction quirk in Lambda – by value 3

```
int main()
{
    int x = 10;
    auto foo = [=] () mutable
    {
        x = 20;
    };

    foo();
}
```

Type deduction quirk in Lambda – by value 4

```
int main()
{
    const int x = 10;
    auto foo = [=] () mutable
    {
        x = 20;
    };

    foo();
}
```

```
error C2166: l-value specifies const object
```

Type deduction quirk in Lambda – by value 5

```
const int x = 10;
auto foo = [=] () const
{
    x = 20;
};
```

```
class compiler_generated_name
{
public:
    void operator()() { cx = 10; }
private:
    const int cx;
};
```

const data member…cannot be modified

Type Deduction in Lambda Capture contd…

- Capture by reference – Type deduction same as that for templates
- Init capture – Type deduction same as that for auto (which is almost same as templates but one of two minor exceptions – not too important for now)
- By Value – Type deduction same as that for template except for one major difference: and a bit un-intuitive: Maintains the CV-qualifiers

Fun things

Template meta programming

{ Code Example }

Template meta programming

{ Code Example }

Return of the Move

{ need for speed }

Remember – short string optimization and return value optimization

End of Day 1

Welcome back – Day 2

copy-swap idiom

copy-swap 1

```
class foo
{
    int size_;
    int* data_;

public:
    explicit foo(const int size)
        : size_(size), data_(size_ ? new
int[size_]():nullptr){}

    foo(const foo& other)
        : size_(other.size_), data_(size_ ? new
int[size_]():nullptr)
    {
        std::copy(other.data_, other.data_ +
other.size_, data_);
    }

    foo& operator=(const foo& other)
    {
       ...
    }

    ~foo()
    {
        if(data_) {
            delete [] data_;
        }
}
};
```

Copy assignment operator

```
 foo& operator=(const foo& other)
    {
        if (this == &other)
            return *this;

        delete[] data_;
        data_ = nullptr;

        size_ = other.size_;
        data_ = size_ ? new int[size_]()
: nullptr;

        std::copy(other.data_,
other.data_ + other.size_, data_);

        return *this;
    }
};
```

copy-swap 2

```
 foo& operator=(const foo& other)
    {
        if (this == &other)
            return *this;

        delete[] data_;
        data_ = nullptr;

        size_ = other.size_;
        data_ = size_ ? new int[size_]() : nullptr;

        std::copy(other.data_, other.data_ +
other.size_, data_);

        return *this;
    }
};
```

But we are releasing our resources.

This might throw.

Provides basic exception safety guarantee

copy-swap 3

```
 foo& operator=(const foo& other)
    {
        if (this == &other)
            return *this;

        delete[] data_;
        data_ = nullptr;

        size_ = other.size_;
        data_ = size_ ? new int[size_]() : nullptr;

        std::copy(other.data_, other.data_ +
other.size_, data_);

        return *this;
    }
};
```

```
foo& operator=(const foo& other) {

        int size_temp = other.size_;
        int* data_temp = size_ ? new int[size_]()
: nullptr;

        std::copy(other.data_, other.data_ +
other.size_, data_temp);

        delete[] data_;

        data_ = data_temp;
        size_ = size_temp;

        return *this;
    }
```

Provides strong exception safety guarantee

copy-swap 4

Copy constructor

```
 foo(const foo& other)
        : size_(other.size_), data_(size_ ? new
int[size_]():nullptr)
    {
        std::copy(other.data_, other.data_ +
other.size_, data_);
    }
```

Copy assignment operator

```
foo& operator=(const foo& other) {

        int size_temp = other.size_;
        int* data_temp = size_ ? new int[size_]()
: nullptr;
        try {
              std::copy(other.data_, other.data_ +
             other.size_, data_temp);
        }
        else {
          delete [] data_temp;
          throw;
        }
        delete[] data_;
        data_ = data_temp;
        size_ = size_temp;

        return *this;
    }
```

Duplicate code between copy constructor and assignment operator.

copy-swap – avoid duplication

Copy constructor

```
 foo(const foo& other)
        : size_(other.size_), data_(size_ ? new
int[size_]():nullptr)
    {
        std::copy(other.data_, other.data_ +
other.size_, data_);
    }
```

Overload swap

```
friend void swap(foo& lhs, foo& rhs) noexcept
{
    using std::swap;
    swap(lhs.size_, rhs.size_);
    swap(lhs.data_, rhs.data_);
}
```

Copy assignment operator

```
foo& operator=(const foo& other) {
     foo temp(other);
     swap(temp, *this);
     return *this;
}
```

copy-swap – use compiler to do the copy

```
foo& operator=(const foo& other) {
     foo temp(other);
     std::swap(temp, *this);
     return *this;
}
```

Take by value

```
foo& operator=(foo other) {
     std::swap(other, *this);
     return *this;
}
```

copy-swap 5

```
 foo& operator=(const foo& other)
    {
        if (this == &other)
            return *this;

        delete[] data_;
        data_ = nullptr;

        size_ = other.size_;
        data_ = size_ ? new int[size_]()
: nullptr;

        std::copy(other.data_,
other.data_ + other.size_, data_);

        return *this;
    }
};
```

Provides basic exception safety guarantee

```
friend void swap(foo& lhs, foo& rhs) noexcept
{
    using std::swap;
    swap(lhs.size_, rhs.size_);
    swap(lhs.data_, rhs.data_);
}

foo& operator=(foo other) {
      swap(temp, *this);
      return *this;
}
```

Provides strong exception safety guarantee

Templates

Template class and specialization

```
template<typename T>
class foo
{
    T x_;
};
```

Primary template

```
template <>
class foo<float>
{
    float x_;
};
```

Explicit specialization

```
template <typename T>
class foo<T*>
{
    T* x_;
};
```

Partial specialization

Explicit specialization

```
template<typename T>
class foo
{
    T x_;
    const static bool value_ = false;
};
```

Primary template

```
template <>
class foo<float>
{
    float x_;
    const static bool value_ = true;
};
```

Explicit specialization for float

```
template <>
class foo<double>
{
    double x_;
    const static bool value_ = true;
};
```

Explicit specialization for double

Explicit specialization - 2

```
template<typename T>
struct foo
{
    T x_;
    const static bool value_ = false;
};
```

```
template <>
struct foo<float>
{
    float x_;
    const static bool value_ = true;
};
```

```
template <>
struct foo<double>
{
    double x_;
    const static bool value_ = true;
};
```

```
int main()
{
    std::cout << foo<float>::value_ << std::endl; // true
    std::cout << foo<double>::value_ << std::endl; // true
    std::cout << foo<int>::value_ << std::endl; // false
    std::cout << foo<bool>::value_ << std::endl; // false
}
```

Explicit specialization – 3

```
template<typename T>
struct is_floating_point_type
{
    T x_;
    const static bool value_ = false;
};
```

```
template <>
struct is_floating_point_type<float>
{
    float x_;
    const static bool value_ = true;
};
```

```
template <>
struct is_floating_point_type<double>
{
    double x_;
    const static bool value_ = true;
};
```

```
int main()
{
    std::cout << is_floating_point_type<float>::value_ << std::endl;// true
    std::cout << is_floating_point_type<double>::value_ << std::endl;// true
    std::cout << is_floating_point_type<int>::value_ << std::endl;// false
    std::cout << is_floating_point_type<bool>::value_ << std::endl;// false
}
```

CONGRATULATIONS…
YOU HAVE JUST
WRITTEN A
TYPE_TRAIT

Type Traits

What is type trait - Type traits defines a compile-time template-based interface to query or modify the properties of type

Synonym for traits

Off the shelf type_traits (from standard library)

• is_integral	• is_floating_point	• is_array
• is_rvalue_reference	• is_const	• is_same
• is_enum	• is_class	• is_function

https://en.cppreference.com/w/cpp/header/type_traits

Exercise time

Exercise 4

Go to https://en.cppreference.com/w/cpp/header/type_traits and explore the following type_traits…and satisfy yourself that you understand what's going on.

- is_polymorphic
- is_object
- is_reference
- is_lvalue_reference
- is_rvalue_reference
- is_const
- is_function
- is_floating_point
- is_same
- remove_reference
- remove_const

Exercise 5

Step 1. Write a bare bones class called “foo”. Need not have any data member.

Step 2 .Write a bare bones class called “bar”. Have a vector of “foo”s as a data member.

Step 3. within bar, create an alias for foo (typedef or using). Call it child_t

Step 4. write main function and using type traits, can you find a way to tell whether **bar::child_t** is same as **foo?**

{ Code Example }

Exercise 6

```
template<typename T>
void foo(T x)
{
    //only for floating point type
}
```

```
template<typename T>
std::enable_if_t<std::is_floating_point_v<T>>
foo(T x)
{
    //only for floating point type
}
```

```
int main()
{
    int a = 10;
    foo(a); // this should give you compiler error
    float b = 42.2;
    foo(b); // this should work

    getchar();
}
```

{ Code Example }

CONGRATULATIONS…
YOU HAVE JUST USED
SFINAE

SFINAE – substitution failure is not an error

Exercise time

Exercise 5

~~Step 1 – Create a JIRA ticket on IDES2 project with issue type "training". Please use the following format for the summary of the ticket : <gbg_username> C++ workshop 2018 refactoring exercise~~

Step 2 – Create a branch locally on your machines with branch name in following format : IDES2-2243-refactoring-<gbg_username>

Step 3 – Do the following refactoring

- Refactor out unnecessary code in the following classes –
    - ides::ocr::result::character class
    - ides::ocr::result::word class
    - ides::ocr::result::line class
    - ides::ocr::result::paragraph class
    - ides::ocr::result::page class

* NOTE – before refactoring the code…make sure there are unit tests available for those functions. If there are no unit tests write them…and only then refactor the code

Step 4 – commit your changes to your branch

Step 5 – push you branch to git

Git cheatsheet

- Create branch – git checkout -b <branch_name>

- Commit your changes –
  - Step 1 – stage your changes: git add –u
  - Step 2 – commit your changes: git commit –m “commit message”

- Push your branch – git push

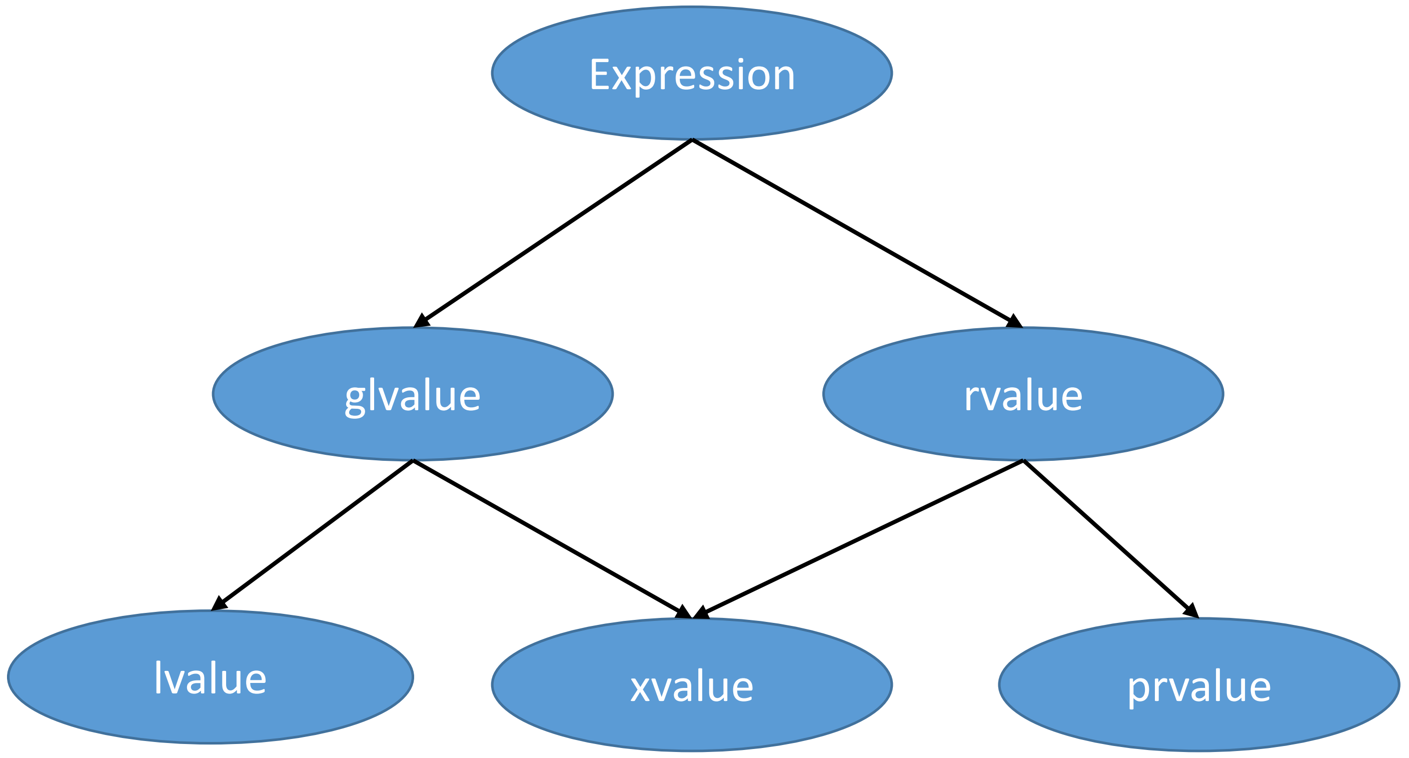
Thank you

References

- Effective Modern C++ - Scott Meyers
- Type Deduction - https://www.youtube.com/watch?v=wQxj20X-tIU
- C++ Templates: The Complete Guide – 2nd Edition
- https://www.fluentcpp.com/2017/01/19/making-code-expressive-lambdas/ - Good use of BIG lambdas
- https://www.fluentcpp.com/2018/07/13/the-incredible-const-reference-that-isnt-const/ - Constness
- https://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Scott-Meyers-Universal-References-in-Cpp11
- https://herbsutter.com/2008/01/01/gotw-88-a-candidate-for-the-most-important-const/
- https://www.youtube.com/watch?v=T5swP3dr190
- https://www.youtube.com/watch?v=7LxepUEcXA4 – copy-swap idiom

Miscellaneous

Value Categories – full picture



Source

1. https://stackoverflow.com/questions/3601602/what-are-rvalues-lvalues-xvalues-glvalues-and-prvalues
2. http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2010/n3092.pdf : section 3.10