MODERN C++ - *New Features*

C++ 11/14/17

C++ 11

- uniform initialization
- auto
- decltype
- for with range based
- lambda function
- constructor can execute other
- override, final
- nullptr
- enum class
- explicit
- variadic template
- union extended by non-trivial objects
- user-defined literals
- default constructor and deleted special functions
- static_assert
- shared_ptr and unique_ptr

Uniform Initialization

```
class Data
{
private:
        string  text;
        int     number;
public:
        Data ()
                : text   ("")
                , number (1)
        { }

        Data (const string& t, const int n)
                : text        (t)
                , number (n)
        { }

        int getNumber (void)
        { return number; }

        operator int() const
        { return number; }
};
```

```
int what (Data());

Data createData(void)
{
        return {"Data", 3};
}

int what (Data (*pointer) () )
{
     return pointer() * 3 + 10;
}

what(&createData)

int varInt {Data{}};
```

Auto

```
std::vector<std::map<std::string, float>> nestedType;

std::vector<std::map<std::string, float>>::iterator it = nestedType.begin();

auto autoIt = nestedType.begin();

auto                    autoSize   = 0u;
unsigned long long      size       = 0u;
autoSize   = 99999999999999ull;
size       = 99999999999999ull;

auto var = {1, 2, 3, 4, 5, 6}; //??
```

Decltype

```
template<class T>
class Object
{
public:
        T x;
};

template<class  T, class R>
void createObjects (T t, R r)
{
        Object<decltype(t)> object1;
        Object<decltype(r)> object2;
}
```

Range-based for

```
std::vector<int> vec   = {1, 2, 3, 4, 5};
int array[]            = {1, 2, 3, 4, 5};
int *dyn_array         = new int[5];

std::copy(array, array+5, dyn_array);

for (const auto& val : vec)
      std::cout<<val<<std::endl;

for (const auto val : array)
      std::cout<<val<<std::endl;

for (auto val : dyn_array)
      std::cout<<val<<std::endl; // ??
```

Lambda function

[=]	()	mutable	throw()	->int
capture clause	*parameter list*	*mutable option*	*exeption option*	*return*

[] captures nothing
[=] captures all automatic variables by const copy and current object by const reference if exists
[&] captures all automatic variables by reference and current object by reference if exists
[a,b,&c] where a and b are captured by copy and c is captured by reference.
[this] captures the current object (*this) by reference

```
auto lambda_fun = [ ] (int x, int y) -> int {return x + y;};
int r = lambda_fun(1, 2);


std::vector<int> vec = {9, 12, 8, 15, 7, 12};

std::for_each(vec.begin(), vec.end(),
              [] (int& item) {item = std::min(10, item);});

auto iterator = std::find_if(vec.begin(), vec.end(),
                             [] (int item) -> bool {
                                 if(item == 9)
                                     return true;
                                 return false;
                             });

std::for_each(vec.begin(), vec.end(),
              [] (const int item) {std::cout<<item<<std::endl;});
```

How will look lambda_function declaration without auto?

Delegating constructors

```
class DelegatingCons
{
public:
	DelegatingCons  (const int _a)
	 : a   (_a)
	{ }
	DelegatingCons  (void)
	 : DelegatingCons   (1)
	{ }
private :
	int a;
};
```

Override final

```
class A
{
public:
	virtual int f1  (int a = 10)  {return a;}
	virtual int f2  (float a)       {return a;}
	virtual int f3  (int a) final  {return a;}
};

class B final : public A
{
public:
	virtual int f1 (int a)       override  {return a;}
	virtual int f2 (double a) override  {return a;}
	virtual int f3 (int a)                       {return a;}
};
```

Nullptr

```
nullptr_t null = nullptr;

auto f1 = [] (int  a) {std::cout<<a<<std::endl;};
auto f2 = [] (int* a) {std::cout<<a<<std::endl;};

f1(NULL);
f2(NULL);

f1(nullptr);
f2(nullptr);

int*   a = NULL;
int    b = NULL;
bool c = NULL;
char d = NULL;

int*   a = nullptr;
int    b = nullptr;
bool c = nullptr;
char d = nullptr;
```

Enum class

```
enum class BasicColors : unsigned char
{
	RED,
	GREEN,
	BLUE
};
enum class Colors : unsigned int
{
	RED,
	GREEN,
	BLUE,
	WHITE,
	BLACK,
	YELLOW = 1233
};
BasicColors color    = BasicColors::RED;
Colors      colorEx    = Colors::YELLOW;
color = colorEx; // ??
int intColor = color; // ??
```

Explicit

```
class A
{
public:
        A ()
        { };
        explicit A (int )
        { };
private:
        A& operator =( const int& )
        {return *this;}
};

A a  = 2; //??
a     = 3; //??
a     = 4; //??
A b  (13);//??
```

Variadic template

```
template<class N, class...>
struct are_same : std::true_type
{
};

template<class N, class T, class... TT>
struct are_same<N, T, TT...>
    : std::integral_constant<bool, std::is_same<N,T>{} &&
are_same<T, TT...>{}>
{
};

template<class N, class... T>
void variadic_template(T ...a)
{
  static_assert(are_same<N, T...>{}, "types aren't that
same");
  auto size      = sizeof...(a);
  N data[size] = {static_cast<N>(a)...};
  for(decltype(size) i = 0; i < size; ++i)
        cout<<data[i]<<endl;
}
```

```
variadic_template<const char
*>("Mobica","Open","Day","2017");
variadic_template<int>(1,2,3,4,5,6,7);
variadic_template<int>(1,1.0f,1.00,2,3.14);
```

static_assert

```
int libraryVersion = 4;

static_assert (sizeof(int)       == 4, "Wrong size of int");
static_assert (sizeof(void *)  == 8, "Wrong pointer size");
static_assert (libraryVersion == 4, "Wrong library version");

assert(sizeof(int)       == 4 && "Wrong size of int");
assert(sizeof(void *)   == 8 && "Wrong pointer size");
assert(libraryVersion == 4 && "Wrong Library version");
```

Union extended by non-trivial objects

```
class Data
{
public:
        Data(int _a, int _b)
        : a     (_a)
        , b     (_b)
        { }
private:
        int a;
        int b;
};
union Union
{
        int        i;
        float      f;
        Data      u;
        Union() {}
        Union(const Data& _u) : u(_u)
        { }
        Union& operator = (const Data& _u)
        { new(&u) Data(_u); return *this; }
};
```

```
Union dataObj   = Data(1,2);
dataObj.u        = {1, 2};
dataObj          = Data(1,4);
```

User-defined literals

```
size_t  operator " " _size_t (long double val)
{
        return static_cast<size_t>(val);
}
size_t  operator " " _size_t (unsigned long long val)
{
        return static_cast<size_t>(val);
}
size_t  operator " " _size_t (const char* val, size_t size)
{
        return static_cast<size_t>(atoi(val));
}
size_t  operator " " _size_t (const char* val)
{
        return static_cast<size_t>(atoi(val));
}

size_t a = 2134_size_t;
size_t b = "2134"_size_t;
```

Default constructor and deleted special functions

```
class Data
{
public:
	Data() = default;
	Data(const Data&) = delete;
	void copy(void)
	{
		Data a(*this);
	}
};
```

```
Data data;

class OldData
{
private:
	OldData(const OldData&)
	{ }
public:
	OldData()
	{ }
	void copy(void)
	{
		OldData a(*this);
	}
};
OldData oldData;
```

What are rule of three and rule of five?

shared_ptr and unique_ptr

```
struct A
{
    int a;
    int b;
    int c;
};

std::unique_ptr<A> uPtr (new A);
std::unique_ptr<A> owner (std::move(uPtr));
owner->a = 10;

std::shared_ptr<A> sPtr = std::make_shared<A>();
sPtr = std::move(owner);

std::shared_ptr<A> s2Ptr = sPtr;
s2Ptr = std::make_shared<A>();
```

C++ 14

- return type deduction and generic lambda
- variable templates
- binary literals and separator
- deprecated

Return type deduction and generic lambda

```
std::vector<int> vec {1, 2, 3, 4, 5};
auto L1 = [ ] (auto a, auto b) -> auto              {return a+b;};
auto L2 = [ ] (auto a, auto b)                      {return a+b;};
auto L3 = [&] (auto index)    -> decltype(auto) {return vec[index];};

auto l1   = L1(1,2);
auto l2   = L2(1,2.2);
auto l2s = L2("Open Day"s, "Mobica"s);
L3(2)     = 123;


template<typename T, typename F>
auto select(const std::vector<T>& c,F f) -> std::vector<decltype(f(c[0]))>
{
   using R = decltype(f(c[0]));
   std::vector<R> v;
   std::transform(std::begin(c), std::end(c), std::back_inserter(v), f);
   return v;
}

auto r2 = select(vec, [ ](auto e){return std::to_string(e);});
```

Return type deduction and generic lambda

```
auto F1(auto x, auto y)
{
        return x+y;
}

template<class T>
struct A
{
        T data;
};

template<class T>
struct B
{
        auto operator + (auto a)
        {
            return data + a.data;
        }

        T data;
};

auto sumAB = F1(B<int>(), A<float>());
```

variable templates

```
template<typename T>
T temp = T(12.12);

unsigned int a = temp<int>; //??
temp<int>       = 43;
float           b = temp<float>;
//??
```

binary literals and separator

```
int         a = 0b1011;
long int    b = 100'000;
float       c = 0b10101;
```

binary literals and separator

```
[[deprecated("f() this function is deprecated")]]
int f (void)
{
	return 0;
}

auto a = f();
```

C++ 17

- type safe union
- auto in templates
- If with initializer
- structure binding

Type safe union

```
using namespace std;
variant <int, float> a;
a = 10;
int i      = get<int>(a);
auto au = get<3> (a);
try
{
    get<float>(a);
}
catch (bad_variant_access&) {}
```

Auto in templates

```
template <auto A>
A f (A a)
{
    return a;
}

f<10> (6);
```

If with initializer

```
using namespace std;
map<int, std::string> m;
if (auto it = m.find(10); it != m.end()) { return it->size(); }
```

structure binding

```
using namespace std;
struct StructA
{
        int       a;
        float     b;
        string  c;
};

StructA getA (void)
{
        return {1,2.2,"Mobica"};
}

auto [a, b, c] = getA();
```

Thank you for your attention