Programming in Modern C++

Module M51: C++11 and beyond: General Features: Part 6: Rvalue & Perfect Forwarding

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Weekly Recap

- Introduced several C++11 general features:
  - auto / decltype
  - suffix return type (+ C++14)
  - Initializer List
  - Uniform Initialization
  - Range for Statement
  - constexpr (+ C++14)
  - noexcept
  - nullptr
  - Inline namespace
  - `static_assert`
  - User-defined Literals (+ C++14)
  - Digit Separators and Binary Literals (+ C++14)
  - Raw String Literals
  - Unicode Support
  - Memory Alignment
  - Attributes (+ C++14)
- Understood the difference between Copying & Moving and Lvalue & Rvalue
- Learnt the advantages of Move in C++11 using Rvalue Reference, Move Semantics, and Copy / Move Constructor / Assignment
- Learnt to implement move semantics in UDTs using `std::move` and to implement `std::move`
- Studied a project to code move-enabled UDTs

Module Objectives

- To understand how Rvalue Reference works as a Universal Reference under template type deduction
- To understand the problem of forwarding of parameters under template type deduction
- To learn how Universal Reference and `std::forward` can work for perfect forwarding of parameters under template type deduction
- To understand the implementation of `std::forward`
- To understand how Move is an optimization of Copy

Module Outline

Universal References

Sources:

- Universal References in C++11 – Scott Meyers, isocpp.org, 2012
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Scott Meyers on C++
- Understanding Move Semantics and Perfect Forwarding: Part 1, Part 2: Rvalue References and Move Semantics, Part 3: Perfect Forwarding, Drew Campbell, 2018

Universal References

Reference Collapsing in Templates: Recap (Module 50)

- In C++03, given

```
template<typename T> void f(T& param);
int x;
f<int&>(x);                              // T is int&
```

- f is initially instantiated as

```
void f(int& & param);                    // reference to reference
```

- C++03's reference-collapsing rule says
  - `T& & => T&`
- So, after reference collapsing, f's instantiation is actually: `void f(int& param);`
- C++11's rules take rvalue references into account:
  - `T& & => T&     // from C++03`
  - `T&& & => T&    // new for C++11`
  - `T& && => T&    // new for C++11`
  - `T&& && => T&& // new for C++11`
- Summary:
  - *Reference collapsing involving a* `&` *is always* `T&`
  - *Reference collapsing involving only* `&&` *is* `T&&`

T&& Parameter Deduction in Templates: Recap (Module 50)

- Function templates with a T&& parameter need not generate functions taking a T&& parameter!

```
template<typename T> void f(T&& param); // note non-const rvalue reference
```

- T's deduced type depends on what is passed to param:
  - **Lvalue** ⇒ T is an lvalue reference (T&)
  - **Rvalue** ⇒ T is a non-reference (T)
- In conjunction with reference collapsing:

```
int x;
f(x);               // lvalue: generates f<int&>(int& &&), calls f(int&)
f(10);              // rvalue: generates f<int>(int&&), calls f(int&&)

TVec vt;            // typedef vector<int> TVec;
                    // TVec createTVec();
f(vt);              // lvalue: generates f<TVec&>(TVec& &&), calls f(TVec&)
f(createTVec());    // rvalue: generates f<TVec>(TVec&&), calls f(TVec&&)
```

Universal References

- `T&&` really is a magical reference type!
  - For lvalue arguments, `T&&` becomes `T&` => lvalues can bind
  - For rvalue arguments, `T&&` remains `T&&` => rvalues can bind
  - For `const/volatile` arguments, `const/volatile` becomes part of `T`
  - `T&&` parameters can bind *anything*
- Two conceptual meanings for `T&&` syntax:
  - **Rvalue reference.** Binds rvalues only

```
void f(Widget&& param);      // takes only non-const rvalue
```

  - **Universal reference.** Binds lvalues and rvalues

```
template<typename T>
void f(T&& param);          // takes lvalue or rvalue, const or non-const
```

    - Really an rvalues reference in a reference-collapsing context

auto&& $\equiv$ T&&

- auto type deduction $\equiv$ template type deduction, so auto&& variables are also universal references:

```
int calcVal();
int x;

auto&& v1 = calcVal(); // deduce type from rvalue => v1's type is int&&

auto&& v2 = x;         // deduce type from lvalue => v2's type is int&
```

- Note that decltype()&& does not behave like a universal references as it does not use template type deduction:

```
decltype(calcVal()) v3; // deduced type is int

decltype(x) v4;         // deduced type is int
```

Rvalue References vs. Universal References

- Read code carefully to distinguish them
  - Both use && syntax: Occus after a POD or UDT for Rvalue References, but after type variable T for Universal References
  - Type deduction for T for Universal References
  - Behavior is different:
    - ▷ Rvalue references bind only rvalues
    - ▷ Universal references bind lvalues *and* rvalues
      - that is, may become either T& or T&&, depending on initializer
- Consider std::vector:

```
template<class T, class Allocator=allocator<T>> // from C++11 Standard
class vector { public: ...
    void push_back(const T& x);          // lvalue reference
    void push_back(T&& x);               // rvalue reference!
    template<class... Args>
    void emplace_back(Args&&... args);   // universal reference
    ...
};
```

Rvalue Ref. vs. Universal Ref.: Illustration

Post-Recording

```
#include <iostream>
using namespace std;

// overloaded functions for test
void test(const int& a) // lvalue ref
{ cout << "lvalue:" << a << endl; }
void test(int&& a)      // rvalue ref
{ cout << "rvalue: " << a << endl; }

template <typename T>
class Data { T _data; public:
    Data(T data): _data(data)
    { cout << "ctor " << endl; }
    Data(Data&& obj) // move ctor - rvalue ref
    { cout << "mtor " << endl; }
    // class template
    void f1(T&& v) { // rvalue ref
        test(forward<T>(v));
    }
    template<typename U> // member fn. template
    void f2(U&& v) { // universal ref
        test(forward<U>(v)); //
    }
};
```

```
void g(int&& param) // simple fn - rvalue ref
{ test(forward<int>(param)); }

template<typename T>
void f(T&& param)   // template fn - universal ref
{ test(forward<T>(param)); }

int main() { int a = 20;
    //g(a);      // cannot bind rvalue reference of
                 // type int&& to lvalue of type int
    g(move(a));      // rvalue: 20
    f(a);            // lvalue: 20
    f(move(a));      // rvalue: 20

    Data<int> d1(10);              // ctor
    Data<int> d2(move(d1)); // mtor: rvalue ref
    //d1.f1(a); // cannot bind rvalue reference of
                // type int&& to lvalue of type int
    d1.f1(move(a)); // rvalue: 20
    d1.f2(a);       // lvalue: 20
    d1.f2(move(a)); // rvalue: 20
}

// For std::forward - See Perfect Forwarding
```

Perfect Forwarding

Sources:

- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Scott Meyers on C++
- Perfect Forwarding, modernescpp.com, 2016
- Understanding Move Semantics and Perfect Forwarding: Part 1, Part 2: Rvalue References and Move Semantics, Part 3: Perfect Forwarding, Drew Campbell, 2018

Perfect Forwarding

Perfect Forwarding

- Goal: one function that *does the right thing:*
  - *Copies* lvalue args
  - *Moves* rvalue args
- Solution is a perfect forwarding function:
  - Templatized function forwarding `T&&` params to members
- **What is Perfect Forwarding?**
  - Perfect forwarding allows a template function that accepts a set of arguments to forward these arguments to another function whilst retaining the lvalue or rvalue nature of the original function arguments
- Let us check an example

Perfect Forwarding Example: (broken)

```
#include <iostream>
class Data { int i; public: Data(): i(0) { } }; // a UDT

void g(const int&) { std::cout << "int& in g" << "; "; }  // binds with lvalue parameter
void g(int&&)      { std::cout << "int&& in g" << "; "; } // binds with rvalue parameter

void h(const Data&) { std::cout << "Data& in h" << std::endl; }  // binds with lvalue parameter
void h(Data&&)      { std::cout << "Data&& in h" << std::endl; } // binds with rvalue parameter

template<typename T1, typename T2>
void f(T1&& p1, T2&& p2) { // universal ref. gets lvalue or rvalue from arg by template type deduction
    g(p1); // always binds with lvalue parameter as p1 is an lvalue in f
    h(p2); // always binds with lvalue parameter as p2 is an lvalue in f
}
int main() { int i { 0 }; Data d;
    f(i, d);                       // (lvalue, lvalue) binds with int& in g; Data& in h
    f(std::move(i), d);            // (rvalue, lvalue) binds with int& in g; Data& in h
    f(i, std::move(d));            // (lvalue, rvalue) binds with int& in g; Data& in h
    f(std::move(i), std::move(d)); // (rvalue, rvalue) binds with int& in g; Data& in h
}
```

- Lvalue arg passed to p1 ⇒ `g(const int&)` receives Lvalue
- Rvalue arg passed to p1 ⇒ `g(int&&)` receives Lvalue
- Lvalue arg passed to p2 ⇒ `h(const Data&)` receives Lvalue
- Rvalue arg passed to p2 ⇒ `h(Data&&)` receives Lvalue

Perfect Forwarding Example: (fixed) by std::forward

```
#include <iostream>
class Data { int i; public: Data(): i(0) { } }; // a UDT

void g(const int&) { std::cout << "int& in g" << "; "; }  // binds with lvalue parameter
void g(int&&)      { std::cout << "int&& in g" << "; "; } // binds with rvalue parameter

void h(const Data&) { std::cout << "Data& in h" << std::endl; }  // binds with lvalue parameter
void h(Data&&)      { std::cout << "Data&& in h" << std::endl; } // binds with rvalue parameter

template<typename T1, typename T2>
void f(T1&& p1, T2&& p2) { // universal ref. gets lvalue or rvalue from arg by template type deduction
    g(std::forward<T1>(p1)); // std::forward forwards lvalue arg to lvalue param and
    h(std::forward<T2>(p2)); // rvalue arg to rvalue param
}
int main() { int i { 0 }; Data d;
    f(i, d);                         // (lvalue, lvalue) binds with int& in g; Data& in h
    f(std::move(i), d);              // (rvalue, lvalue) binds with int&& in g; Data& in h
    f(i, std::move(d));              // (lvalue, rvalue) binds with int& in g; Data&& in h
    f(std::move(i), std::move(d)); // (rvalue, rvalue) binds with int&& in g; Data&& in h
}
```

- Lvalue arg passed to p1 ⇒ `g(const int&)` receives Lvalue
- Rvalue arg passed to p1 ⇒ `g(int&&)` receives Rvalue
- Lvalue arg passed to p2 ⇒ `h(const Data&)` receives Lvalue
- Rvalue arg passed to p2 ⇒ `h(Data&&)` receives Rvalue

Perfect Forwarding

- Despite `T&&` parameters, code fully type-safe:
- Type compatibility verified upon instantiation
  - Only `int`-compatible types valid for call to `g()`
  - Only `Data`-compatible types valid for call to `h()`. For example in the context of

```
...
class DerivedData: public Data { public: DerivedData(): Data() { } };
...
int main() { ... DerivedData d; ... }
```

The code works exactly as before. Whereas for

```
...
class OtherData { int i; public: OtherData(): i(0) { } }; // another UDT
...
int main() { ... OtherData d; ... }
```

The code fails compilation: `error: no matching function for call to h(OtherData&)`

Perfect Forwarding

- The flexibility can be removed via `static_assert` (Module 48) as follows:

```
...
template<typename T1, typename T2>
void f(T1&& p1, T2&& p2) {
    // Asserts that T2 must be of type Data
    static_assert(std::is_same< typename std::decay<T2>::type, Data >::value,
        "T2 must be Data");

    g(std::forward<T1>(p1)); // T1 too may be asserted, if needed
    h(std::forward<T2>(p2));
}
...
class DerivedData: public Data { public: DerivedData(): Data() { } };
...
int main() { ... DerivedData d; ... }
```

Compile-time error: `error: static assertion failed: T2 must be Data`

Perfect Forwarding Example 1: Modified from slide M51.14


```
#include <iostream>
class Data { int i; public: Data(int i = 5): i(i) { }
    operator int() { return i; }              // cast to int
    Data& operator++() { ++i; return *this; } // pre-increment operator
    Data& operator--() { --i; return *this; } // pre-decrement operator
};
void g(int& a) { std::cout << "int& in g: " << ++a << "; ";  }        // binds non-const lvalue param
void g(int&& a) { std::cout << "int&& in g: " << --a << "; "; }       // binds rvalue param

void h(Data& a) { std::cout << "Data& in h: " << ++a << std::endl; }   // binds non-const lvalue param
void h(Data&& a) { std::cout << "Data&& in h: " << --a << std::endl; } // binds rvalue param

template<typename T1, typename T2>
void f(T1&& p1, T2&& p2) {
    g(...); // called on p1 with or without std::forward
    h(...); // called on p1 with or without std::forward
}
int main() { int i { 0 }; Data d;
    f(i, d);        // (lvalue, lvalue)
    f(5, d);        // (rvalue, lvalue)
    f(i, Data(7)); // (lvalue, rvalue)
    f(5, Data(7)); // (rvalue, rvalue)
}
```

Without std::forward

```
int& in g: 1; Data& in h: 6
int& in g: 6; Data& in h: 7
int& in g: 2; Data& in h: 8
int& in g: 6; Data& in h: 8
```

With std::forward

```
int& in g: 1; Data& in h: 6
int&& in g: 4; Data& in h: 7
int& in g: 2; Data&& in h: 6
int&& in g: 4; Data&& in h: 6
```

Perfect Forwarding Example 2: Generic Factory Method/1

- Let us write a *generic factory method* that should be able to create each arbitrary object. That means that the function should have the following characteristics:
  - Can take an arbitrary number of arguments
  - Can accept lvalues and rvalues as an argument
  - Forwards it arguments identical to the underlying constructor

```
#include <iostream>

template <typename T, typename Arg>  // For efficiency reasons, the function template should
T CreateObject(Arg& a) {             // take its arguments by a non-const lvalue reference
    return T(a);
}
int main() {
    int five=5; // lvalues
    int myFive= CreateObject<int>(five);
    std::cout << "myFive: "  << myFive << std::endl;

    int myFive2= CreateObject<int>(5); // rvalues: error: cannot bind non-const lvalue reference
                                       // of type int& to an rvalue of type int
    std::cout << "myFive2: " << myFive2 << std::endl;
}
```

Perfect Forwarding Example 2: Generic Factory Method/2

- To solve the compilation issues, we can go one of two ways:
  - Change the non-const lvalue reference to a const lvalue reference (that can bind an rvalue). But that is not perfect, because we cannot change the function argument, if needed
  - Overload the function template for a const lvalue reference and a non-const lvalue reference. That is preferred

```
#include <iostream>

template <typename T, typename Arg> T CreateObject(Arg& a) { return T(a); }       // binds lvalues

template <typename T, typename Arg> T CreateObject(const Arg& a) { return T(a); } // binds rvalues

int main() {
    int five = 5; // lvalues
    int myFive = CreateObject<int>(five);
    std::cout << "myFive: "  << myFive << std::endl;  // myFive: 5

    int myFive2 = CreateObject<int>(5); // rvalues
    std::cout << "myFive2: " << myFive2 << std::endl; // myFive2: 5
}
```

Perfect Forwarding Example 2: Generic Factory Method/3

- The solution has two conceptual issues:
  - To support n arguments, we need to overload $2^n + 1$ variations of `CreateObject<T>(...)`. "+1" for the function `CreateObject<T>()` without any argument
  - Without the overload, the forwarding problem would appear for rvalue arguments as they will be *copied* instead of being *moved*
- So we need to use universal reference in `CreateObject<T>(...)` with `std::forward`

```
#include <iostream>

template <typename T, typename Arg>
T CreateObject(Arg&& a) {              // Universal reference
    return T(std::forward<Arg>(a)); // std::forward
}
int main() {
    int five = 5; // lvalues
    int myFive = CreateObject<int>(five);
    std::cout << "myFive: "  << myFive << std::endl;  // myFive: 5

    int myFive2 = CreateObject<int>(5); // rvalues
    std::cout << "myFive2: " << myFive2 << std::endl; // myFive2: 5
}
```

Perfect Forwarding Example 2: Generic Factory Method/4

- `CreateObject<T>()` needs exactly one argument perfectly forwarded to the constructor
- For arbitrary number of arguments, we need a *variadic template* (TBD later)

```
#include <iostream>
#include <string>
#include <utility>
template <typename T, typename ... Args> // Variadic Templates can get an arbitrary number of arguments
    T CreateObject(Args&& ... args) { return T(std::forward<Args>(args)...); }
int main() {
    int five = 5, myFive = CreateObject<int>(five); // lvalues
    std::cout << "myFive: "  << myFive << std::endl;                    // myFive: 5
    std::string str { "Lvalue" }, str2 = CreateObject<std::string>(str);
    std::cout << "str2: " << str2 << std::endl;                         // str2: Lvalue

    int myFive2 = CreateObject<int>(5); // rvalues
    std::cout << "myFive2: " << myFive2 << std::endl;                   // myFive2: 5
    std::string str3 = CreateObject<std::string>(std::string("Rvalue"));
    std::cout << "str3: " << str3 << std::endl;                         // str3: Rvalue
    std::string str4 = CreateObject<std::string>(std::move(str3));
    std::cout << "str4: " << str4 << std::endl;                         // str4: Rvalue

    double doub = CreateObject<double>(); // Arbitrary number of args
    std::cout << "doub: " << doub << std::endl;                         // doub: 0
    struct Data { Data(int i, double d, std::string s) { } } d = CreateObject<Data>(2011, 3.14, str4);
}
```

Perfect Forwarding Example 3: apply Functor/1

- Let us design an apply functor that would take a function and its arguments and apply the function on the arguments

```
template<typename F, typename... Ts> // Using variadic template (TBD later)
auto apply(std::ostream& os, F&& func, Ts&&... args)
    -> decltype(func(args...)) { // may not preserves rvalue-ness
    os << "Forwarding:: ";
    return func(args...);        // may not preserves rvalue-ness
}
```

- args... are lvalues, but apply's caller may have passed rvalues:
  - Templates can distinguish rvalues from lvalues
  - apply might call the wrong overload of func

```
class Data { };
Data myData() { return Data(); }

class DataDispatcher { public:
    void operator()(const Data&) { std::cout << "operator()(const Data&) called\n\n"; } // takes lvalue
    void operator()(Data&&) { std::cout << "operator()(Data&&) called\n\n"; }           // takes rvalue
};
int main() { Data d = myData();
    apply(std::cout, DataDispatcher(), d);        // Forwarding:: operator()(const Data&) called
    apply(std::cout, DataDispatcher(), myData()); // Forwarding:: operator()(const Data&) called
                                                  // rvalue forwarded as lvalue!
}
```

Perfect Forwarding Example 3: apply Functor/2

- Naturally, perfect forwarding solves

```
template<typename F, typename... Ts>
auto apply(std::ostream& os, F&& func, Ts&&... args) // return type is same as func's on original args
    -> decltype(func(std::forward<Ts>(args)...)) { // preserves lvalue-ness / rvalue-ness
    os << "Forwarding:: ";
    return func(std::forward<Ts>(args)...);       // preserves lvalue-ness / rvalue-ness
}
...
int main() { Data d = myData();
    apply(std::cout, DataDispatcher(), d);        // Forwarding:: operator()(const Data&) called

    apply(std::cout, DataDispatcher(), myData()); // Forwarding:: operator()(Data&&) called
}
```

- With return type deduction [C++14]

```
template<typename F, typename... Ts>
decltype(auto) apply(std::ostream& os, F&& func, Ts&&... args) { // return type deduction
    os << "Forwarding:: ";
    return func(std::forward<Ts>(args)...);
}
```

Perfect Forwarding Example 3: apply Functor/3

- Perfect forwarding works perfectly with mixed bindings as well

```
#include <iostream>
using namespace std;
class Data { };
Data myData() { return Data(); }

class DataDispatcher { public: // mixed binding for two parameters
    void operator()(const Data&, const Data&){ cout<< "operator()(const Data&, const Data&) called\n\n"; }
    void operator()(const Data&, Data&&){ cout<< "operator()(const Data&, Data&&) called\n\n"; }
    void operator()(Data&&, const Data&){ cout<< "operator()(Data&&, const Data&) called\n\n"; }
    void operator()(Data&&, Data&&){ cout<< "operator()(Data&&, Data&&) called\n\n"; }
};
template<typename F, typename... Ts>
auto apply(ostream& os, F&& func, Ts&&... args) -> decltype(func(forward<Ts>(args)...)) {
    return func(forward<Ts>(args)...);
}
int main() {
    Data d = myData();
    apply(cout, DataDispatcher(), d, d);                  // operator()(const Data&, const Data&) called
    apply(cout, DataDispatcher(), d, myData());           // operator()(const Data&, Data&&) called
    apply(cout, DataDispatcher(), myData(), d);           // operator()(Data&&, const Data&) called
    apply(cout, DataDispatcher(), myData(), myData()); // operator()(Data&&, Data&&) called
}
```

std::forward

Sources:

- Universal References in C++11 – Scott Meyers, isocpp.org, 2012
- std::forward, cppreference.com
- Quick Q: What's the difference between std::move and std::forward?, isocpp.org
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Scott Meyers on C++

std::forward

std::forward

- Let us relook at:

```
template<typename T1, typename T2>
void f(T1&& p1, T2&& p2) { ... h(std::forward<T2>(p2)); }
```

  - `T` a reference (that is,`T` is `T&`) $\Rightarrow$ lvalue was passed to `p2`
    - `std::forward<T>(p2)` should return lvalue
  - `T` a non-reference (that is, `T` is `T`) $\Rightarrow$ rvalue was passed to `p2`
    - `std::forward<T>(p2)` should return rvalue
- `std::forward` is provided in `<utility>` for this
  - Applicable only to *function templates*
  - *Preserves arguments' lvalue-ness / rvalue-ness /* `const`*-ness* when forwarding them to other functions
- Let us take a look at the implementation

std::forward

- C++11 implementations:

```
template<typename T>        // For lvalues (T is T&);
T&&                         // return lvalue reference
forward(typename remove_reference<T>::type& t) noexcept
{ return static_cast<T&&>(t); }

template<typename T>        // For rvalues (T is T);
T&&                         // return rvalue reference
forward(typename remove_reference<T>::type&& t) noexcept
{ return static_cast<T&&>(t); }
```

- By design, param type disables type deduction ⇒ callers must specify T:

```
template<typename T1, typename T2> void f(T1&& p1, T2&& p2)
{ g(std::forward(p1)); ... } // error! Cannot deduce T1 in call to std::forward

template<typename T1, typename T2> void f(T1&& p1, T2&& p2)
{ g(std::forward<T1>(p1)); ... } // fine
```

Move is an Optimization of Copy

Sources:

- Scott Meyers on C++
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses

Move is an Optimization of Copy

Copy Only

- Move requests for copyable types w/o move support yield copies:

```
class MyResource { public: // w/o move support
    MyResource(const MyResource&); // copy ctor

};
class MyClass { public: // with move support
    MyClass(MyClass&& src)  // move ctor
    // request to move r's value
    : w(std::move(src.r)) { ... }
    private: MyResource r; // no move support
};
```

src.r is copied to r:

- std::move(src.r) returns an rvalue of type MyResource
- That rvalue is passed to MyResource's copy constructor

Copy & Move

- If MyResource adds move support:

```
class MyResource { public: // with move support
    MyResource(const MyResource&);     // copy ctor
    MyResource(MyResource&&) noexcept; // move ctor
};
class MyClass { public: // with move support
    MyClass(MyClass&& src) noexcept
    // request to move r's value
    : w(std::move(src.r)) { ... }
    private: MyResource r;
};
```

src.r is moved to r:

- std::move(src.r) returns an rvalue of type MyResource
- That rvalue is passed to MyResource's move constructor via normal overloading resolution

Move is an Optimization of Copy

- Implications:
  - *Giving classes move support can improve performance even for* **move-unaware code**
    - ▷ Copy requests for rvalues may silently become moves
  - *Move requests safe for types without explicit move support*
    - ▷ Such types perform copies instead
      - – For example, all built-in types (POD)
  - *Move support may exist even if copy operations do not*
    - ▷ For example, **Move-only types** like `std::thread` and `std::unique_ptr` that are *moveable*, but not *copyable*
  - *Types should support move when moving cheaper than copying*
    - ▷ Libraries use moves whenever possible (for example, STL)
- In short:
  - **Give classes move support when moving faster than copying**
  - **Use `std::move` for lvalues that may safely be moved from**

Move is an Optimization of Copy: Use Beyond Construction / Assignment

- Move support useful for other functions, e.g., setters:

```
class MyList { public:
    ...
    void setID(const std::string& newId)          // copy param
    { id = newId; }
    void setID(std::string&& newId) noexcept      // move param
    { id = std::move(newId); }
    void setVals(const std::vector<int>& newVals) // copy param
    { vals = newVals; }
    void setVals(std::vector<int>&& newVals)      // move param
    { vals = std::move(newVals); }
    ...
private:
    std::string id;
    std::vector<int> vals;
};
```

- **Note**:
    - As the move `operator=` of `std::string` is `noexcept`, `setId` is declared `noexcept`
    - Whereas `setVals` is not declared `noexcept`, as the move `operator=` of `std::vector` is not declared `noexcept`

Compiler Generated Move Operations

- Move constructor and move `operator=` are *special*:
  - Generated by compilers under appropriate conditions
- Conditions:
  - *All data members and base classes are movable*
    - ▷ Implicit move operations move everything
    - ▷ Most types qualify:
      - – All built-in types (move $\equiv$ copy).
      - – Most standard library types (for example, all containers).
  - Generated operations likely to maintain class invariants
    - ▷ *No user-declared copy or move operations*
      - – Custom semantics for any $\Rightarrow$ default semantics inappropriate
      - – Move is an optimization of copy
    - ▷ *No user-declared destructor*
      - – Often indicates presence of implicit class invariant
- More on this later in the Module discussing `default` and `delete`

Compiler Generated Move Operations: Custom Deletion ⇒ Custom Copying

```
class Widget { private:
    std::vector<int> v;
    std::set<double> s;
    std::size_t sizeSum;
public:
    ~Widget() { assert(sizeSum == v.size()+s.size()); }
  ...
};
```

- If `Widget` had implicitly-generated move operations:

```
    std::vector<Widget> vw;
    Widget w;
    ...                              // put stuff in w's containers
    vw.push_back(std::move(w));      // move w into vw
    ...                              // no use of w
```

- *User-declared dtor ⇒ no compiler-generated move ops for* `Widget`

Compiler Generated Move Operations: Custom Moving ⇒ Custom Copying

copyable & movable type

```
class Widget1 { private:
    std::u16string name; // copyable/movable type
    long long value;     // copyable/movable type
public: explicit Widget1(std::u16string n);

}; // implicit copy/move ctor
   // implicit copy/move operator=
```

copyable type; not movable

```
class Widget2 { private:
    std::u16string name;
    long long value;
public: explicit Widget2(std::u16string n);
    // user-declared copy ctor
    Widget2(const Widget2& rhs);
}; // => no implicit move ops
   // implicit copy operator=
```

- *Declaring a move operation prevents generation of copy operations*
  - Custom move semantics ⇒ custom copy semantics
    - ▷ **Move is an optimization of copy**

```
class Widget3 { private:                    // movable type; not copyable
    std::u16string name;
    long long value;
public:
    explicit Widget3(std::u16string n);
    Widget3(Widget3&& rhs) noexcept;   // user-declared move ctor => no implicit copy ops
    Widget3&                           // user-declared move op=
    operator=(Widget3&& rhs) noexcept; // => no implicit copy ops
};
```

Module Summary

- Learnt how Rvalue Reference works as a Universal Reference under template type deduction
- Understood the problem of forwarding of parameters under template type deduction and its solution using Universal Reference and `std::forward`
- Learnt the implementation of `std::forward`
- Understood how Move works as an optimization of Copy