Modern C++ Programming

15. Advanced Topics

Federico Busato

University of Verona, Dept. of Computer Science
2021, v3.08

Table of Context

Table of Context

Move Semantic

Move semantics refers in transferring ownership of resources from one object to another

Differently from *copy semantic*, *move semantic* does not duplicate the original resource

In C++ every expression is either an **rvalue** or an **lvalue**

- a **lvalue** (left) represents an expression that occupies some identifiable location in memory
- a **rvalue** (right) is an expression that does not represent an object occupying some identifiable location in memory

```
int x = 5;         // "x" is an lvalue, "5" is an rvalue
int y = 10;        // "y" is an lvalue

int z = (x * y); // "z" is an lvalue, (x * y) is an rvalue
```

C++11 introduces a new kind of *reference* called **rvalue reference** `X&&`

- An **rvalue reference** only binds to an **rvalue**, that is a temporary
- An **lvalue reference** only binds to an **lvalue**
- A **const lvalue reference** binds to both **lvalue** and **rvalue**

```
int          x  = 5;         // "x" is an lvalue
int&         r1 = x;         // "r1" is an lvalue reference
// int&      r2 = 5;         // compile error, "5" is an rvalue
const int& cr = (x * y); // "cr" is an const lvalue reference

int&&        rv = (x * y); // "rv" is an rvalue
// int&&     rv1 = x;        // compile error, "x" is NOT an rvalue
```

```cpp
struct A {};

void f(A& a) {}        // lvalue reference

void g(const A& a) {} // const lvalue reference

void h(A&& a) {}       // rvalue reference

A a;
f(a);      // ok, f() can modify "a"
g(a);      // ok, f() cannot modify "a"
// h(a);   // compile error f() does not accept lvalues

// f(A{}); // compile error f() does not accept rvalues
g(A{});    // ok, f() cannot modify the object A{}
h(A{});    // ok, f() can modify the object A{}
```

```cpp
#include <algorithm>
class Array { // Array Wrapper
public:
    Array() = default;

    Array(int size) : _size{size}, _array{new int[size]} {}

    Array(const Array& obj) : _size{obj._size}, _array{new int[obj._size]}  {
        // EXPENSIVE COPY (deep copy)
        std::copy(obj._array, obj._array + _size, _array);
    }

    ~Array() { delete[] _array; }
private:
    int  _size;
    int* _array;
};
```

```cpp
#include <vector>

int main() {
    std::vector<Array> vector;
    vector.push_back( Array{1000} ); // call push_back(const Array&)
}                                    // expensive copy
```

Before C++11: `Array{1000}` is created, passed by `const-reference`, copied, and then destroyed

Note: `Array{1000}` is no more used outside `push_back`

After C++11: `Array{1000}` is created, and moved to `vector` (fast!)

Class prototype with support for *move semantic*:

```
class X {
public:
    X();                        // default constructor

    X(const X& obj);            // copy constructor

    X(X&& obj);                 // move constructor

    X& operator=(const X& obj); // copy assign operator

    X& operator=(X&& obj);      // move assign operator

    ~X();                       // destructor
};
```

Move constructor semantic

```
X(X&& obj);
```

(1) *Shallow copy* of `obj` data members (in contrast to deep copy)

(2) *Release* any `obj` resources and reset all data members (pointer to `nullptr`, size to 0, etc.)

Move assignment semantic

```
X& operator=(X&& obj);
```

(1) *Release* any resources of `this`

(2) *Shallow copy* of `obj` data members (in contrast to deep copy)

(3) *Release* any `obj` resources and reset all data members (pointer to `nullptr`, size to 0, etc.)

(4) Return `*this`

Move constructor

```
Array(Array&& obj) {
    _size       = obj._size;  // (1) shallow copy
    _array      = obj._array; // (1) shallow copy
    obj._size  = 0;           // (2) release obj (no more valid)
    obj._array = nullptr;     // (2) release obj
}
```

Move assignment

```
Array& operator=(Array&& obj) {
    delete[] _array;          // (1) release this
    _size       = obj._size;  // (2) shallow copy
    _array      = obj._array; // (2) shallow copy
    obj._array = nullptr;     // (3) release obj
    obj._size  = 0;           // (3) release obj
    return *this;             // (4) return *this
}
```

Compiler Implicitly Declares

Special Members

compiler implicitly declares

user declares	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

std::move

C++11 provides the method `std::move` (`<utility>`) to indicate that an object may be "moved from"

It allows to efficient transfer resources from an object to another one

```
#include <vector>

int main() {
    std::vector<Array> vector;
    vector.push_back( Array{1000} );    // call "push_back(Array&&)"

    Array arr{1000};
    vector.push_back( arr );            // call "push_back(const Array&)"

    vector.push_back( std::move(arr) ); // call "push_back(Array&&)"
                                        // efficient!!
//  "arr" is not more valid here
}
```

Universal Reference and Perfect Forwarding

Universal Reference

The `&&` syntax has two different meanings depending on the context it is used

- **rvalue reference**
- Either **rvalue reference** or **lvalue reference** (*universal reference*, cit. Scott Meyers)

"Universal references" (also called *forwarding references*) are **rvalues** that appear in a type-deducing context

```
void f1(int&& t) {} // rvalue reference

template<typename T>
void f2(T&& t) {}   // universal reference

int&&  v1 = ...;    // rvalue reference
auto&& v2 = ...;    // universal reference
```

```cpp
struct A {};
void f1(A&& a) {} // rvalue only

template<typename T>
void f2(T&& t) {} // universal reference

A a;
f1(A{});  // ok
// f1(a); // compile error (only rvalue)
f2(A{});  // universal reference
f2(a);    // universal reference

A&&    a2 = A{}; // ok
// A&& a3 = a;   // compile error (only rvalue)
auto&& a4 = A{}; // universal reference
auto&& a5 = a;   // universal reference
```

Universal Reference - Misleading Cases

```
template<typename T>
void f(std::vector<T>&&) {} // rvalue reference

template<typename T>
void f(const T&&) {}        // rvalue reference (const)

const auto&& v = ...;       // const lvalue reference
```

Reference Collapsing Rules

Before C++11 (C++98, C++03), it was not allowed to take a reference to a reference (`A& &` causes a compile error)

C++11, by contrast, introduces the following **reference collapsing rules**:

```
template<typename T>
void f(T&) {} // compile error in C++98/03 (with gcc),
              // no errors in C++11 (and clang with C++98/03)
int a = 3;    //
f<int&>(a);   //
```

Type	Reference		Result
A&	&	→	A&
A&	&&	→	A&
A&&	&	→	A&
A&&	&&	→	A&&

Perfect Forwarding

Perfect forwarding allows preserving argument *value category* and `const`/`volatile` modifiers

`std::forward` (`<utility>`) forwards the argument to another function with the *value category* it had when passed to the calling function (*perfect forwarding*)

```
#include <utility> // std::forward
template<typename T> void f(T& t)  { cout << "lvalue"; }
template<typename T> void f(T&& t) { cout << "rvalue"; } // overloading

template<typename T> void g1(T&& obj) { f(obj); } // call only f(T&)
template<typename T> void g2(T&& obj) { f(std::forward<T>(obj)); }

struct A{};
f ( A{10} ); // print "rvalue"
g1( A{10} ); // print "lvalue"!!
g2( A{10} ); // print "rvalue"
```

Value Categories

Taxonomy (simplified)

Every expression is either an **rvalue** or an **lvalue**

- An **lvalue** (*left* value of an assignment for historical reason or *locator* value) represents an expression that occupies an *identity*, namely a memory location (it has an address)

- An **rvalue** is movable; an **lvalue** is not

glvalue (*generalized lvalue*) is an expression that has an identity

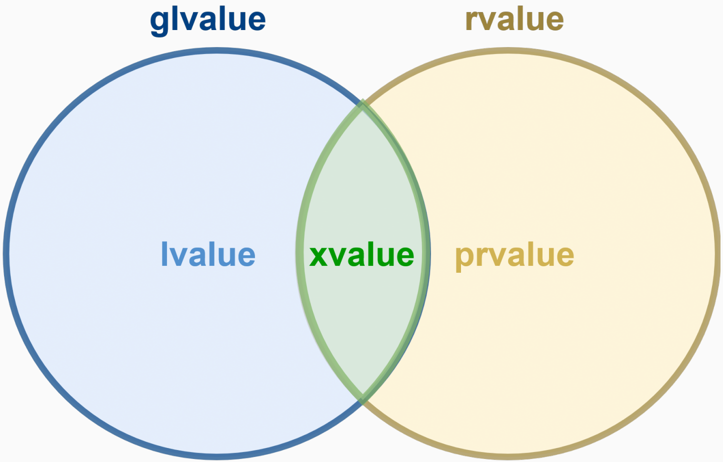
lvalue is a **glvalue** but it is not movable (it is not an **xvalue**). An *named rvalue reference* is a **lvalue**

xvalue (*eXpiring*) has an identity and it is movable. It is a **glvalue** that denotes an object whose resources can be reused. An *unnamed rvalue reference* is a **xvalue**

prvalue (*pure rvalue*) doesn't have identity, but is movable. It is an expression whose evaluation initializes an object or computes the value of an operand of an operator

rvalue is movable. It is a **prvalue** or an **xvalue**

en.cppreference.com/w/cpp/language/value_category


glvalue
rvalue
lvalue
xvalue
prvalue

Examples

```cpp
struct A {
    int x;
};

void f(A&&) {}
A&&  g();
//--------------------------------------------------------------
int a = 4;        // "a" is an lvalue, "4" is a prvalue
f(A{4});          // "A{4}" is a prvalue

A&& b = A{3};     // "A&& b" is a named rvalue reference → lvalue

A c{4};
f(std::move(c)); // "std::move(c)"  is a xvalue
f(A{}.x);         // "A{}.x" is a xvalue
g();              // "A&&" is a xvalue
```

Copy Elision and RVO

Copy elision is a compiler optimization technique that eliminates unnecessary copying/moving of objects (it is defined in the C++ standard)

A compiler avoids omitting copy/move operations with the following optimizations:

- **RVO (Return Value Optimization)** means the compiler is allowed to avoid creating *temporary* objects for return values
- **NRVO (Named Return Value Optimization)** means the compiler is allowed to return an object (with automatic storage duration) without invokes copy/move constructors

Returning an object from a function is *very expensive* without RVO/NVRO:

```
struct Obj {
    Obj() = default;

    Obj(const Obj&) { // non-trivial
        cout << "copy constructor\n";
    }
};

Obj f() { return Obj{}; } // first copy

auto x1 = f();            // second copy (create "x")
```

If provided, the compiler uses the *move constructor* instead of *copy constructor*

RVO Copy elision is always guarantee if the operand is a **prvalue** of the same class type and the *copy constructor* is trivial and non-deleted

```
struct Trivial {
    Trivial()                = default;
    Trivial(const Trivial&) = default;
};

// sigle instance
Trivial f1() {
   return Trivial{};  // Guarantee RVO
}
// distinct instances and run-time selection
Trivial f2(bool b) {
    return b ?  Trivial{} : Trivial{}; // Guarantee RVO
}
```

In C++17, *RVO Copy elision* is always guarantee if the operand is a **prvalue** of the same class type, even if the *copy constructor* is not trivial or deleted

```
struct S1 {
  S1()           = default;
  S1(const S1&) = delete; // deleted
};
struct S2 {
  S2()           = default;
  S2(const S2&) {}         // non-trivial
};

S1 f() { return S1{}; }
S2 g() { return S2{}; }

auto x1 = f(); // compile error in C++14
auto x2 = g(); // RVO only in C++17
```

RVO Example - Where it does NOT work

NRVO is not always guarantee even in C++17

```
Obj f1() {
    Obj a;
    return a; // most compilers apply NRVO
}

Obj f2(bool v) {
    Obj a;
    if (v)
       return a;  // copy/move constructor
    return Obj{}; // RVO
}
```

```
Obj f3(bool v) {
    Obj a, b;
    return v ? a : b; // copy/move constructor
}

Obj f4() {
    Obj a;
    return std::move(a); // force move constructor
}

Obj f5() {
    static Obj a;
    return a;            // only copy constructor is possible
}
```

Type Deduction

Type Deduction

When you call a template function, you may omit any template argument that the compiler can determine or deduce (inferred) by the usage and context of that template function call [IBM]

- The compiler tries to deduce a template argument by comparing the type of the corresponding template parameter with the type of the argument used in the function call
- Similar to function default parameters, (any) template parameters can be deduced only if they are at end of the parameter list

Full Story: IBM Knowledge Center

Example

```
template<typename T>
int add1(T a, T b) {  return a + b; }

template<typename T, typename R>
int add2(T a, R b) { return a + b;  }

template<typename T, int B>
int add3(T a) { return a + B; }

template<int B, typename T>
int add4(T a) { return a + B; }

add1(1, 2);         // ok
// add1(1, 2u);       // the compiler expects the same type
add2(1, 2u);        // ok (add2 is more generic)
add3<int, 2>(1); // "int" cannot be deduced
add4<2>(1);         // ok
```

Type Deduction - Pass-by-Reference

Type deduction with references

```
template<typename T>
void f(T& a) {}

template<typename T>
void g(const T& a) {}

int          x = 3;
int&         y = x;
const int& z = x;
f(x);   // T: int
f(y);   // T: int
f(z);   // T: const int // <-- !! it works...but it does not
g(x);   // T: int       //     for "f(int& a)"!!
g(y);   // T: int       //     (only non-const references)
g(z);   // T: int       // <-- note the difference
```

Type deduction with pointers

```
template<typename T>
void f(T* a) {}

template<typename T>
void g(const T* a) {}

int*       x = nullptr;
const int* y = nullptr;
auto       z = nullptr;
f(x);    // T: int
f(y);    // T: const int
// f(z);    // compile error!! z: "nullptr_t != T*"
g(x);    // T: int
g(y);    // T: int   <-- note the difference
```

Type Deduction - Pass-by-Pointer


```
template<typename T>
void f(const T* a) {} // pointer to const-values

template<typename T>
void g(T* const a) {} // const pointer

int*             x = nullptr;
const int*       y = nullptr;
int* const       z = nullptr;
const int* const w = nullptr;
f(x);    // T: int
f(y);    // T: int
f(z);    // T: int
// g(x); // compile error!! objects pointed are not constant
// g(y); // the same (the pointer itself is constant)
g(z);    // T: int
g(w);    // T: const int
```

Type deduction with values

```
template<typename T>
void f(T a) {}

template<typename T>
void g(const T a) {}

int         x = 2;
const int   y = 3;
const int& z = y;
f(x);   // T: int
f(y);   // T: int!!  (drop const)
f(z);   // T: int!!  (drop const&)
g(x);   // T: int
g(y);   // T: int
g(z);   // T: int!!  (drop reference)
```

```cpp
template<typename T>
void f(T a) {}



int*       x = nullptr;
const int* y = nullptr;
int* const z = x;
f(x);  // T = int*
f(y);  // T = int* !! (const drop)
f(z);  // T = int* const
```

Type Deduction - Array

Type deduction with arrays

```
template<typename T, int N>
void f(T (&array)[N]) {}    // type and size deduced

template<typename T>
void g(T array) {}

int       x[3] = {};
const int y[3] = {};
f(x);   // T: int, N: 3
f(y);   // T: const int, N: 3
g(x);   // T: int*
g(y);   // T: const int*
```

Type Deduction - Conflicts


```
template<typename T>
void add(T a, T b) {}

template<typename T, typename R>
void add(T a, R b) {}

template<typename T>
void add(T a, char b) {}

add(2, 3.0f);          // call add(T, R)
// add(2, 3);          // error!! ambiguous match
add<int>(2, 3);        // call add(T, T)
add<int, int>(2, 3); // call add(T, R)
add(2, 'b');           // call add(T, char) -> nearest match
```

Type Deduction- Conflicts


```cpp
template<typename T, int N>
void f(T (&array)[N]) {}

template<typename T>
void f(T* array) {}

// template<typename T>
// void f(T array) {} // ambiguous

int x[3];
f(x); // call f(T*) not f(T(&)[3]) !!
```

`auto` Deduction

- `auto x =` copy by-value/by-const value
- `auto& x =` copy by-reference/by-const-refernce
- `auto* x =` copy by-pointer/by-const-pointer
- `auto&& x =` copy by-universal reference
- `decltype(auto) x =` automatic type deduction

```cpp
int            f1(int& x) { return x; }
int&           f2(int& x) { return x; }
auto           f3(int& x) { return x; }
decltype(auto) f4(int& x) { return x; }

int  v  = 3;
int  x1 = f1(v);
int& x2 = f2(v);
// int& x3 = f3(v);
int& x4 = f4(v);
```

`const` Correctness

`const` Correctness

`const` correctness refers to guarantee object/variable `const` consistency throughout its lifetime and ensuring safety from unintentional modifications

References:

- Isocpp: const-correctness
- GotW: Const-Correctness
- Abseil: Meaningful ‘const’ in Function Declarations
- const is a contract
- Why const Doesn’t Make C Code Faster
- Constant Optimization?

- `const` entities do not change their values at run-time. This does not imply that they are evaluated at compile-time
- `const T*` is different from `T* const`. The first case means *"the content does not change"*, while the later *"the value of the pointer does not change"*
- Pass *by-const-value* and *by-value* parameters imply the *same* function signature
- Return *by-const-value* and *by-value* have different meaning
- `const_cast` can *break* const-correctness

`const` and member functions:

- `const` member functions do not change the internal status of an object
- `mutable` fields can be modified by a `const` member function (they should not change the external view)

`const` and code optimization:

- `const` keyword purpose is for correctness (*type safety*), not for performance
- `const` may provide performance advantages in a few cases, e.g. non-trivial copy semantic

Function Declarations Example

```cpp
void f(int);
void f(const int); // the declaration is exactly the same of
                   // "void f(int)"!!
void f(int*);
void f(const int*); // different declaration

void f(int&);
void f(const int&); // different declaration
```

```cpp
int          f();
// const int f(); // compile error conflicting declaration
```

`const` Return Example

```
   const int const_value = 3;

   const int& f2() { return const_value; }
// int&       f1() { return const_value; } // WRONG
   int         f3() { return const_value; }         // ok
```

```
struct A {
    void f()       { cout << "non-const"; }
    void f() const { cout << "const";     }
};

const A getA() { return A{}; }

auto a = getA(); // "a" is a copy
a.f();           // print "non-const"

getA().f();      // print "const"
```

struct Example

```
struct A {          // struct A_const { // equal to "const A"
    int* ptr;       //     int* const ptr;
    int  value;     //     const int  value;
};                  // };

void f(A a) {
    a.value  = 3;
    a.ptr[0] = 3;
}
void g(const A a) {  // the same with g(const A&)
//  a.value  = 3;    // compile error
    a.ptr[0] = 3;    // "const" does not apply to "ptr" content!!
}

A a{new int[10]};
f(a);
g(a);
```

Member Functions Example

```
struct A {
   int value = 0;

   int&        f1() { return value; }
   const int& f2() { return value; }

// int&        f3() const { return value; } // WRONG
   const int& f4() const { return value; }

   int        f5() const { return value; } // ok
   const int f6() const { return value; }
};
```

Undefined Behavior

Undefined behavior means that the semantic of certain operations is undefined (outside the language/library specification) or illegal, and the compiler presumes that such operations never happen

Motivations behind undefined behavior:

- *Compiler optimizations*, e.g. signed overflow or NULL pointer deferencing
- *Simplify compile checks*

Some undefined behavior cases provide an *implementation-defined behavior* depending on the compiler and platform. In this case, the code is *not portable*

- What Every C Programmer Should Know About Undefined Behavior
- What are all the common undefined behaviours that a C++ programmer should know about?

- `const_cast` applied to a `const` variables

```
const int        var  = 3;
const_cast<int>(var) = 4;
... // use var
```

- Memory alignment

```
char* ptr  = new char[512];
auto  ptr2 = reinterpret_cast<uint64_t*>(ptr + 1);
ptr2[3]; //  ptr2 is not aligned to 8 bytes (sizeof(uint64_t))
```

- Memory initialization

```
int var;
// use var
auto var2 = new int;
// use var2
```

- **Memory access-related**
  - NULL pointer deferencing
  - Out-of-bound access

- **Platform specific behavior**
  - Endianness

```
union U {
    unsigned x;
    char     y;
};
```

  - Type definition

```
long x = 1ul << 32u; // different behavior depending on the OS
```

  - Intrinsic functions

- **Strict aliasing**

```
float x = 3;
auto  y = reinterpret_cast<unsigned&>(x);
// x, y break the strict aliasing rule
```

- **Lifetime issues**

```
int* f() {
    int tmp[10];
    return tmp;
}
int* ptr = f();
ptr[0];
```

- **Unspecified behavior**
  - A legal operation but the C++ standard does not document the results
  - Signed shift `-2 << x` (before C++20), large-than-type shift `3 << 32`, signed overflow, etc.
  - Operation ordering `f(i++, i++)`
- **One Definition Rule violation**
  - Different definitions of `inline` functions in distinct translation units

Detecting Undefined Behavior

There are several ways to detect undefined behavior at compile-time and at run-time:

- Using GCC/Clang undefined behavior sanitizer (run-time check)
- Static analysis tools
- Use `constexpr` expressions as undefined behavior is not allowed

```
constexpr int   x1 = 2147483647 + 1;   // compile error
constexpr int   x2 = (1 << 32);        // compile error
constexpr int   x3 = (1 << -1);        // compile error
constexpr int   x4 =  3 / 0;           // compile error
constexpr int   x5 = *((int*) nullptr) // compile error
constexpr int   x6 = 6
constexpr float x7 = reinterpret_cast<float&>(x6); // compile error
```

C++ Idioms

Rule of Zero

The **Rule of Zero** is a rule of thumb for C++

> Utilize the *value semantics* of existing types to avoid having to implement *custom* copy and move operations

Note: many classes (such as **std** classes) manage resources themselves and should not implement copy/move constructor and assignment operator

```
class X {
public:
    X(...); // constructor
            // NO need to define copy/move semantic
private:
    std::vector<int>     v; // instead raw allocation
    std::unique_ptr<int> p; // instead raw allocation
};                          // see smart pointer
```

Rule of Three

The **Rule of Three** is a rule of thumb for C++(03)

> If your class needs any of
>
> - a copy constructor `X(const X&)`
> - an assignment operator `X& operator=(const X&)`
> - or a destructor `~X()`
>
> defined explicitly, then it is likely to need all three of them

Some resources cannot or should not be copied. In this case, they should be declared as deleted

```
X(const X&) = delete
X& operator=(const X&) = delete
```

Rule of Five

The **Rule of Five** is a rule of thumb for C++11

If your class needs any of

- a copy constructor `X(const X&)`
- a move constructor `X(X&&)`
- an assignment operator `X& operator=(const X&)`
- an assignment operator `X& operator=(X&&)`
- or a destructor `~X()`

defined explicitly, then it is likely to need all five of them

Singleton

Singleton is a software design pattern that restricts the instantiation of a class to one and only one object (a common application is for logging)

```
class Singleton {
public:
    static Singleton& get_instance() {  // note "static"
        static Singleton instance { ..init.. } ;
        return instance;    // destroyed at the end of the program
    }                       // initiliazed at first use

    Singleton(const& Singleton)      = delete;
    void operator=(const& Singleton) = delete;

    void f() {}
private:
    T _data;

    Singleton( ..args.. ) { ... }  // used in the initialization
}
```

PIMPL - Compilation Firewalls

Pointer to IMPLementation (PIMPL) idiom allows decoupling the interface from the implementation in a clear way

header.hpp

```
class A {
public:
    A();
    ~A();
    void f();
private:
    class Impl;  // forward declaration
    Impl* ptr;   // opaque pointer
};
```

NOTE: The class does not expose internal data members or methods

PIMPL - Implementation

`source.cpp` (`Impl` actual implementation)

```
class A::Impl { // could be a class with a complex logic
public:
    void internal_f() {
        ..do something..
    }
private:
    int   _data1;
    float _data2;
};

A::A()    : ptr{new Impl()} {}
A::~A()    { delete ptr; }
void A::f() { ptr->internal_f(); }
```

PIMPL - Advantages, Disadvantages

Advantages:

- ABI stability
- Hide private data members and methods
- Reduce compile type and dependencies

Disadvantages:

- Manual resource management
  - `Impl* ptr` can be replaced by `unique_ptr<impl> ptr` in C++11
- Performance: pointer indirection + dynamic memory
  - dynamic memory could be avoided by using a reserved space in the interface e.g. `uint8_t data[1024]`

PIMPL - Implementation Alternatives

What parts of the class should go into the `Impl` object?

- *Put all private and protected members into `Impl`*:
  Error prone. Inheritance is hard for opaque objects

- *Put all private members (but not functions) into `Impl`*:
  Good. Do we need to expose all functions?

- *Put everything into `Impl`, and write the public class itself as only the public interface, each implemented as a simple forwarding function*:
  Good

https://herbsutter.com/gotw/_100/

The **Curiously Recurring Template Pattern (CRTP)** is an idiom in which a class `X` derives from a class template instantiation using `X` itself as template argument

A common application is *static polymorphism*

```
template <class T>
struct Base {
    void my_method() {
        static_cast<T*>(this)->my_method_impl();
    }
};

class Derived : public Base<Derived> {
//  void my_method() is inherited
    void my_method_impl() { ... } // private method
};
```

```
#include <iostream>
template <typename T>
struct Writer {
    void write(const char* str) {
        static_cast<const T*>(this)->write_impl(str);
    }
};
class CerrWriter : public Writer<CerrWriter> {
    void write_impl(const char* str) {  std::cerr << str;  }
};
class CoutWriter : public Writer<CoutWriter> {
    void write_impl(const char* str) {  std::cout << str;  }
};
CoutWriter x;
CerrWriter y;
x.write("abc");
y.write("abc");
```

```
template <typename T>
void f(Writer<T>& writer) {
    writer.write("abc);
}

CoutWriter x;
CerrWriter y;
f(x);
f(y);
```

Virtual functions cannot have template arguments, but they can be emulated by using the following pattern

```
class Base {
public:
    template<typename T>
    void method(T t) {
        v_method(t);    // call the actual implementation
    }
protected:
    virtual void v_method(int t)    = 0; // v_method is valid only
    virtual void v_method(double t) = 0; // for "int" and "double"
};
```

Template Virtual Function

Actual implementations for derived class `A` and `B`

```
class AImpl : public Base {
protected:
    template<typename T>
    void t_method(T t) { // template "method()" implementation for A
        std::cout << "A " << t << std::endl;
    }
};

class BImpl : public Base {
protected:
    template<typename T>
    void t_method(T t) { // template "method()" implementation for B
        std::cout << "B " << t << std::endl;
    }
};
```

```cpp
template<class Impl>
class DerivedWrapper : public Impl {
private:
    void v_method(int t) override {
        Impl::t_method(t);
    }
    void v_method(double t) override {
        Impl::t_method(t);
    } // call the base method
};

using A = DerivedWrapper<AImpl>;
using B = DerivedWrapper<BImpl>;
```

```cpp
int main(int argc, char* argv[]) {
    A a;
    B b;
    Base* base = nullptr;

    base = &a;
    base->method(1);   // print "A 1"
    base->method(2.0); // print "A 2.0"

    base = &b;
    base->method(1);   // print "B 1"
    base->method(2.0); // print "B 2.0"
}
```

`method()` calls `v_method()` (pure virtual method of `Base`)
`v_method()` calls `t_method()` (actual implementation)

Smart pointers

Smart Pointers

Smart pointer is a pointer-like type with some additional functionality, e.g. *automatic memory deallocation* (when the pointer is no longer in use, the memory it points to is deallocated), reference counting, etc.

C++11 provides three smart pointer types:

- **`std::unique_ptr`**
- **`std::shared_ptr`**
- **`std::weak_ptr`**

Smart pointers prevent most situations of memory leaks by making the memory deallocation automatic

Smart Pointers Benefits

- If a smart pointer goes *out-of-scope*, the appropriate method to release resources is called automatically. The memory is not left dangling

- Smart pointers will automatically be set to `nullptr` if not initialized or when memory has been released

- `std::shared_ptr` provides automatic reference count

- If a special `delete` function needs to be called, it will be specified in the pointer type and declaration, and will automatically be called on delete `std::unique_ptr` is used to manage any dynamically allocated object that is not shared by multiple objects

```
#include <iostream>
#include <memory>
struct A {
    A()  { std::cout << "Constructor\n"; } // called when A()
    ~A() { std::cout << "Destructor\n";  } // called when u_ptr1,
};                                        // u_ptr2 are out-of-scope
int main() {
    auto raw_ptr = new A();
    std::unique_ptr<A> u_ptr1(new A());
    std::unique_ptr<A> u_ptr2(raw_ptr);
//  std::unique_ptr<A> u_ptr3(raw_ptr); // no compile error, but wrong!!
                                        // (same pointer)
//  u_ptr1 = &raw_ptr;  // compile error (unique pointer)
//  u_ptr1 = u_ptr2;    // compile error (unique pointer)
    u_ptr1 = std::move(u_ptr2); // delete u_ptr1;
}                               // u_ptr1 = u_ptr2;
                                // u_ptr2 = nullptr
```

`std::unique_ptr` methods

- `get()` returns the underlying pointer
- `operator*` `operator->` dereferences pointer to the managed object
- `operator[]` provides indexed access to the stored array (if it supports random access iterator)
- `release()` returns a pointer to the managed object and releases the ownership
- `reset(ptr)` replaces the managed object with `ptr`

Utility method: `std::make_unique<T>()` creates a unique pointer of a class `T` that manages a new object

```
#include <iostream>
#include <memory>
struct A {
    int value;
};
int main() {
    std::unique_ptr<A> u_ptr1(new A());
    u_ptr1->value;       // dereferencing
    (*u_ptr1).value;     // dereferencing

    auto u_ptr2 = std::make_unique<A>();  // create a new unique pointer

    u_ptr1.reset(new A());             // reset
    auto raw_ptr = u_ptr1.release();  // release
    delete[] raw_ptr;

    std::unique_ptr<A[]> u_ptr3(new A[10]);
    auto& obj = u_ptr3[3];             // access
}
```

Implement a custom deleter

```
#include <iostream>
#include <memory>
struct A {
    int value;
};
int main() {
    auto DeleteLambda = [](A* x) {
        std::cout << "delete" << std::endl;
        delete x;
    };

    std::unique_ptr<A, decltype(DeleteLambda)>
        x(new A(), DeleteLambda);
} // print "delete"
```

`std::shared_ptr` is the pointer type to be used for memory that can be owned by multiple resources at one time

`std::shared_ptr` maintains a reference count of pointer objects. Data managed by `std::shared_ptr` is only freed when there are no remaining objects pointing to the data

```
#include <iostream>
#include <memory>
struct A {
    int value;
};
int main() {
    std::shared_ptr<A> sh_ptr1(new A());
    std::shared_ptr<A> sh_ptr2(sh_ptr1);
    std::shared_ptr<A> sh_ptr3(new A());
    sh_ptr3 = nullptr; // allowed, the underlying pointer is deallocated
                       //   sh_ptr3 : zero references
    sh_ptr2 = sh_ptr1; // allowed // sh_ptr1, sh_ptr2: two references
    sh_ptr2 = std::move(sh_ptr1); // allowed // sh_ptr1: zero references
}                                           // sh_ptr2: one references
```

`std::shared_ptr` methods

- `get()` returns the underlying pointer
- `operator*` `operator->` dereferences pointer to the managed object
- `use_count()` returns the number of objects referring to the same managed object
- `reset(ptr)` replaces the managed object with `ptr`

Utility method: `std::make_shared()` creates a shared pointer that manages a new object

```cpp
#include <iostream>
#include <memory>
struct A {
    int value;
};
int main() {
    std::shared_ptr<A> sh_ptr1(new A());
    auto sh_ptr2 = std::make_shared<A>();  // std::make_shared
    std::cout << sh_ptr1.use_count();  // print 1

    sh_ptr1 = sh_ptr2;                 // copy
//  std::shared_ptr<A> sh_ptr2(sh_ptr1); // copy (constructor)
    std::cout << sh_ptr1.use_count();  // print 2
    std::cout << sh_ptr2.use_count();  // print 2

    auto raw_ptr = sh_ptr1.get();      // get
    sh_ptr1.reset(new A());            // reset
    (*sh_ptr1).value = 3;              // dereferencing
    sh_ptr1->value = 2;                // dereferencing
}
```

A `std::weak_ptr` is simply a `std::shared_ptr` that is allowed to dangle (pointer not deallocated)

```
#include <iostream>
#include <memory>
struct A {
    int value;
};
int main() {
    auto ptr = new A();
    std::weak_ptr<A>   w_ptr(ptr);
    std::shraed_ptr<A> sh_ptr(new A());

    sh_ptr = nullptr;
//  delete sh_ptr.get(); // double free or corruption

    w_ptr = nullptr;
    delete w_ptr; // ok valid
}
```

`std::weak_ptr` - Weak Pointer

It must be converted to `std::shared_ptr` in order to access the referenced object

`std::weak_ptr` methods

- `use_count()` returns the number of objects referring to the same managed object
- `reset(ptr)` replaces the managed object with `ptr`
- `expired()` checks whether the referenced object was already deleted (`true`, `false`)
- `lock()` creates a `std::shared_ptr` that manages the referenced object

```cpp
#include <iostream>
#include <memory>
struct A {
    int value;
};
int main() {
    auto sh_ptr1 = std::make_shared<A>();
    std::cout << sh_ptr1.use_count();  // print 1
    std::weak_ptr<A> w_ptr = sh_ptr1;
    std::cout << w_ptr.use_count();    // print 1

    auto sh_ptr2 = w_ptr.lock();
    std::cout << kk.use_count();       // print 2 (sh_ptr1 + sh_ptr2)

    sh_ptr1 = nullptr;
    std::cout << w_ptr.expired();     // print false
    sh_ptr2 = nullptr;
    std::cout << w_ptr.expired();      // print true
}
```

Concurrency

Overview

C++11 introduces the **Concurrency** library to simplify managing OS threads

```cpp
#include <iostream>
#include <thread>

void f() {
    std::cout << "first thread" << std::endl;
}

int main(){
    std::thread th(f);
    th.join();          // stop the main thread until "th" complete
}
```

How to compile:

```
$g++ -std=c++11 main.cpp -pthread
```

Example

```cpp
#include <iostream>
#include <thread>
#include <vector>
void f(int id) {
    std::cout << "thread " << id << std::endl;
}
int main() {
    std::vector<std::thread> thread_vect; // thread vector
    for (int i = 0; i < 10; i++)
        thread_vect.push_back( std::thread(&f, i) );

    for (auto& th : thread_vect)
        th.join();

    thread_vect.clear();
    for (int i = 0; i < 10; i++) {  // thread + lambda expression
        thread_vect.push_back(
            std::thread( [](){ std::cout << "thread\n"; } );
    }
}
```

Thread Methods

Library methods:

- `std::this_thread::get_id()` returns the thread id
- `std::thread::sleep_for( sleep_duration )`
  Blocks the execution of the current thread for at least the specified `sleep_duration`
- `std::thread::hardware_concurrency()` returns the number of concurrent threads supported by the implementation

Thread object methods:

- `get_id()` returns the thread id
- `join()` waits for a thread to finish its execution
- `detach()` permits the thread to execute independently from the thread handle

Thread Methods 2/2

```
#include <chrono>   // the following program should (not deterministic)
#include <iostream> // produces the output:
#include <thread>   //    child thread exit
                   //    main thread exit
int main() {
    using namespace std::chrono_literals;
    std::cout << std::this_thread::get_id();
    std::cout << std::thread::hardware_concurrency(); // e.g. print 6

    auto lambda = []() {
        std::this_thread::sleep_for(1s); // t2
        std::cout << "child thread exit\n";
    };
    std::thread child(lambda);
    child.detach(); // without detach(), child must join() the
                    // main thread (run-time error otherwise)
    std::this_thread::sleep_for(2s);    // t1
    std::cout << "main thread exit\n";
}
// if t1 < t2 the should program prints:
```

Parameters Passing

Parameters passing *by-value* or *by-pointer* to a thread function works in the same way of a standard function. *Pass-by-reference* requires a special wrapper (`std::ref` , `std::cref`) to avoid wrong behaviors

```cpp
#include <iostream>
#include <thread>
void f(int& a, const int& b) {
    a = 7;
    const_cast<int&>(b) = 8;
}
int main() {
    int a = 1, b = 2;
    std::thread th1(f, a, b);                    // wrong!!!
    std::cout << a << ", " << b << std::endl;    // print 1, 2!!

    std::thread th2(f, std::ref(a), std::cref(b)); // correct
    std::cout << a << ", " << b << std::endl;    // print 7, 8!!
    th1.join(); th2.join();
}
```

Mutex (The Problem)

The following code produces (in general) a value < 1000:

```
#include <chrono>
#include <iostream>
#include <thread>
#include <vector>
void f(int& value) {
    for (int i = 0; i < 10; i++) {
        value++;
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
}
int main() {
    int value = 0;
    std::vector<std::thread> th_vect;
    for (int i = 0; i < 100; i++)
        th_vect.push_back( std::thread(f, std::ref(value)) );
    for (auto& it : th_vect)
        it.join();
    std::cout << value;
}
```

C++11 provide the **`mutex`** class as synchronization primitive to protect shared data from being simultaneously accessed by multiple threads

`mutex` methods:

- **`lock()`** locks the *mutex*, blocks if the *mutex* is not available
- **`try_lock()`** tries to lock the *mutex*, returns if the *mutex* is not available
- **`unlock()`** unlocks the *mutex*

More advanced mutex can be found here: `en.cppreference.com/w/cpp/thread`

C++ includes three mutex wrappers to provide safe copyable/movable objects:

- `lock_guard` (C++11) implements a strictly scope-based mutex ownership wrapper
- `unique_lock` (C++11) implements movable mutex ownership wrapper
- `shared_lock` (C++14) implements movable shared mutex ownership wrapper

```cpp
#include <thread> // iostream, vector, chrono

void f(int& value, std::mutex& m) {
    for (int i = 0; i < 10; i++) {
        m.lock();
        value++;   // other threads must wait
        m.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
}
int main() {
    std::mutex m;
    int value = 0;
    std::vector<std::thread> th_vect;
    for (int i = 0; i < 100; i++)
        th_vect.push_back( std::thread(f, std::ref(value), std::ref(m)) );
    for (auto& it : th_vect)
        it.join();
    std::cout << value;
}
```

Atomic

`std::atomic` (C++11) template class defines an atomic type that are implemented with lock-free operations (much faster than locks)

```
#include <atomic> // chrono, iostream, thread, vector
void f(std::atomic<int>& value) {
    for (int i = 0; i < 10; i++) {
        value++;
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
}
int main() {
    std::atomic<int> value(0);
    std::vector<std::thread> th_vect;
    for (int i = 0; i < 100; i++)
        th_vect.push_back( std::thread(f, std::ref(value)) );
    for (auto& it : th_vect)
        it.join();
    std::cout << value;    // print 1000
}
```

Task-based parallelism

The **`future`** library provides facilities to obtain values that are returned and to catch exceptions that are thrown by *asynchronous* tasks

Asynchronous call: `std::future async(function, args...)`
runs a function asynchronously (potentially in a new thread)
and returns a `std::future` object that will hold the result

`std::future` methods:

- `T get()` returns the result
- `wait()` waits for the result to become available

`async()` can be called with two launch policies for a task executed:

- `std::launch::async` a new thread is launched to execute the task asynchronously
- `std::launch::deferred` the task is executed on the calling thread the first time its result is requested (lazy evaluation)

Task-based parallelism 2/2

```
#include <future> // numeric, algorithm, vector, iostream
template <typename RandomIt>
int parallel_sum(RandomIt beg, RandomIt end) {
    auto len = end - beg;
    if (len < 1000)    // base case
        return std::accumulate(beg, end, 0);

    RandomIt mid = beg + len / 2;
    auto handle = std::async(std::launch::async, // right side
                             parallel_sum<RandomIt>, mid, end);
    int sum = parallel_sum(beg, mid);              // left side
    return sum + handle.get();                     // left + right
}
int main() {
    std::vector<int> v(10000, 1); // init all to 1
    std::cout << "The sum is " << parallel_sum(v.begin(), v.end());
}
```