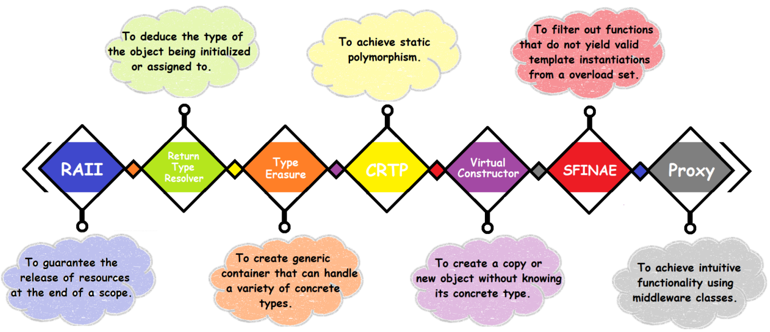
7 Advance C++ Concepts & Idiom Examples You Should Know


To deduce the type of the object being initialized or assigned to.
To achieve static polymorphism.
To filter out functions that do not yield valid template instantiations from a overload set.
RAII
Return Type Resolver
Type Erasure
CRTP
Virtual Constructor
SFINAE
Proxy
To guarantee the release of resources at the end of a scope.
To create generic container that can handle a variety of concrete types.
To create a copy or new object without knowing its concrete type.
To achieve intuitive functionality using middleware classes.

7 Advanced C++ programming styles and idioms
Reading Time: 13 minutes

So I have started updating myself with Modern C++ a while ago & since my post 21 new features of Modern C++ to use in your project & All about lambda function in C++ was popular I decided to write about advance C++ concepts & idioms which I have learned from this wikibook & course.

There are many other advance C++ concepts & idioms as well but I consider these 7 as "should-know". To explain them, I have taken a more pragmatic approach than sophistication. So, I have weighed more on readability, simplicity over other fancy features, syntax sugars and complexity.

Note: There are also drawbacks of using some of these techniques which I have not discussed here or maybe I am not qualified enough.

1. RAII

Intent: To guarantee the release of resource(s) at the end of a scope.
Implementation: Wrap resource into a class; resource acquired in the constructor immediately after its allocation; and automatically released in the destructor; resource used via an interface of the class;
Also known as: Execute-around object, Resource release is finalization, Scope-bound resource management

Problem

- **R**esource **A**cquisition **I**s **I**nitialization idiom is the most powerful & vastly used idiom although the name is really terrible as the idiom is rather about resource release than acquisition.
- RAII guarantee the release of resource at the end of a scope/destruction. It thus ensures no resource leaks and provides basic exception safety guarantee.

```
struct resource
{
    resource(int x, int y) { cout << "resource acquired\n"; }
    ~resource() { cout << "resource destroyed\n"; }
};

void func()
{
    resource *ptr = new resource(1, 2);
    int x;
    std::cout << "Enter an integer: ";
    std::cin >> x;
    if (x == 0)
        throw 0; // the function returns early, and ptr won't be deleted!
    if (x < 0)
        return; // the function returns early, and ptr won't be deleted!
    // do stuff with ptr here
    delete ptr;
}
```

- In the above code, the early `return` or `throw` statement, causing the function to terminate without `ptr` being deleted.
- In addition, the memory allocated for variable `ptr` is now leaked (and leaked again every time this function is called and returns early).

Solution

```
template<class T>
class smart_ptr
{
    T* m_ptr;
public:
    template<typename... Args>
    smart_ptr(Args&&... args) : m_ptr(new T(std::forward<Args>(args)...)){}
    ~smart_ptr() { delete m_ptr; }

    smart_ptr(const smart_ptr& rhs) = delete;
    smart_ptr& operator=(const smart_ptr& rhs) = delete;

    smart_ptr(smart_ptr&& rhs) : m_ptr(exchange(rhs.m_ptr, nullptr)){}
    smart_ptr& operator=(smart_ptr&& rhs){
        if (&rhs == this) return *this;
        delete m_ptr;
        m_ptr = exchange(rhs.m_ptr,nullptr);
        return *this;
    }

    T& operator*() const { return *m_ptr; }
    T* operator->() const { return m_ptr; }
};

void func()
{
    auto ptr = smart_ptr<resource>(1, 2); // now ptr guarantee the release of resource
    // ...
}
```

- Note that no matter what happens after `ptr` declaration, `ptr` will be destroyed when the function terminates (regardless of how it terminates).
- As the `ptr` is a local object, the destructor will be called while the function stack frame rewinds. Hence, we are assured that the `resource` will be properly cleaned up.

Usecases

- Using RAII, resources like `new/delete`, `malloc/free`, acquire/release, mutex lock/unlock, file open/close, count ++/--, database connect/disconnect or anything else that exists in limited supply can easily be managed.
- Examples from C++ Standard Library include `std::unique_ptr`, `std::ofstream`, `std::lock_guard`, etc.

2. Return Type Resolver

Intent: To deduce the type of the object being initialize or assign to.
Implementation: Uses templatized conversion operator.
Also known as: Return type overloading

Issue

```
int from_string(const char *str) { return std::stoi(str); }
float from_string(const char *str) { return std::stof(str); } // error
```

- A function can not overloaded only by its return type.

Solution

```
class from_string
{
    const string m_str;

public:
    from_string(const char *str) : m_str(str) {}
    template <typename type>
    operator type(){
        if constexpr(is_same_v<type, float>)        return stof(m_str);
        else if (is_same_v<type, int>)              return stoi(m_str);
    }
};

int n_int = from_string("123");
float n_float = from_string("123.111");
// Will only work with C++17 due to `is_same_v`
```

If you are unaware of constexpr, I have written a short post on when to use const vs constexpr in c++.

Usecases

- So, when you use **`nullptr`**(introduced in C++11), this is the technique that runs under the hood to deduce the correct type depending upon the pointer variable it is assigning to.
- You can also overcome the function overloading limitation on the basis of a return type as we have seen above.
- Return Type Resolver can also used to provide a generic interface for assignment, independent of the object assigned to.

3. Type Erasure

Intent: To create generic container that can handle a variety of concrete types.
Implementation: Can be implemented by `void*`, templates, polymorphism, union, proxy class, etc.
Also known as: Duck-typing

Problem

- C++ is a statically typed language with strong typing. In statically typed languages, object type known & set at compile-time. While in dynamically typed languages the type associated with run-time values.
- In other words, in strongly typed languages the type of an object doesn't change after compilation.
- However, to overcome this limitation & providing a feature like dynamically typed languages, library designers come up with various generic container kind of things like `std::any`(C++17), `std::variant` (C++17), `std::function`(C++11), etc.

Different Type Erasure Techniques

- There is no one strict rule on how to implement this idiom, it can have various forms with its own drawbacks as follows:

⇒ Type erasure using void* (like in C)

```
void qsort (void* base, size_t num, size_t size,
            int (*compare)(const void*,const void*));
```

Drawback: not safe & separate compare function needed for each type

⇒ Type erasure using C++ templates

```
template <class RandomAccessIterator>
  void sort(RandomAccessIterator first, RandomAccessIterator last);
```

Drawback: may lead to many function template instantiations & longer compilation time

⇒ Type erasure using polymorphism

```
struct base { virtual void method() = 0; };

struct derived_1 : base { void method() { cout << "derived_1\n"; } };
struct derived_2 : base { void method() { cout << "derived_2\n"; } };

// We don't see a concrete type (it's erased) though can dynamic_cast
void call(base* ptr) { ptr->method(); };
```

Drawback: run-time cost (dynamic dispatch, indirection, vtable, etc.)

⇒ Type erasure using union

```
struct Data {};

union U {
    Data d;         // occupies 1 byte
    std::int32_t n; // occupies 4 bytes
    char c;         // occupies 1 byte

    ~U() {}         // need to know currently active type
}; // an instance of U in total occupies 4 bytes.
```

Drawback: not type-safe

Solution

- As I mentioned earlier that standard library already has such generic containers.
- To understand type erasure better let's implement one i.e. `std::any`:

```
struct any
{
    struct base {};

    template<typename T>
    struct inner: base{
        inner(T t): m_t{std::move(t)} {}
        T m_t;
        static void type() {}
    };

    any(): m_ptr{nullptr}, typePtr{nullptr} {}

    template<typename T>
    any(T && t): m_ptr{std::make_unique<inner<T>>(t)}, typePtr{&inner<T>::type} {}

    template<typename T>
    any& operator=(T&& t){
        m_ptr = std::make_unique<inner<T>>(t);
        typePtr = &inner<T>::type;
        return *this;
    }

private:
    template<typename T>
    friend T& any_cast(const any& var);

    std::unique_ptr<base> m_ptr = nullptr;
    void (*typePtr)() = nullptr;
};

template<typename T>
T& any_cast(const any& var)
{
    if(var.typePtr == any::inner<T>::type)
        return static_cast<any::inner<T>*>(var.m_ptr.get())->m_t;
    throw std::logic_error{"Bad cast!"};
}

int main()
{
    any var(10);
    std::cout << any_cast<int>(var) << std::endl;

    var = std::string{"some text"};
    std::cout << any_cast<std::string>(var) << std::endl;

    return 0;
}
```

Especially, the thing here to note is how we are leveraging empty static method i.e. `inner<T>::type()` to determine template instance type in `any_cast<T>`.

Usecases

- Employ to handle multiple types of the return value from function/method(Although that’s not recommended advice).

4. CRTP

Intent: To achieve static polymorphism.
Implementation: Make use of base class template spcialization.
Also known as: Upside-down inheritance, Static polymorphism

Problem
```
struct obj_type_1
{
    bool operator<(const value &rhs) const {return m_x < rhs.m_x;}
    // bool operator==(const value &rhs) const;
    // bool operator!=(const value &rhs) const;
    // List goes on. . . . . . . . . . . . . . . . . . . . .
private:
    // data members to compare
};

struct obj_type_2
{
    bool operator<(const value &rhs) const {return m_x < rhs.m_x;}
    // bool operator==(const value &rhs) const;
    // bool operator!=(const value &rhs) const;
    // List goes on. . . . . . . . . . . . . . . . . . . . .
private:
    // data members to compare
};

struct obj_type_3 { ...
struct obj_type_4 { ...
// List goes on. . . . . . . . . . . . . . . . . . . . .
```

- For each comparable objects, you need to define respective comparison operators. This is redundant because if we have an `operator <` , we can overload other operators on the basis of it.
- Thus, `operator <` is the only one operator having type information, other operators can be made type independent for reusability purpose.

Solution

- **C**uriously **R**ecurring **T**emplate **P**attern implementation rule is simple, ***separate out the type-dependent & independent functionality and bind type independent functionality with the base class using self-referencing template.***
- Above line may seem cryptic at first. So, consider the below solution to the above problem for more clarity:

```
template <class derived>
struct compare {};

struct value : compare<value> {
    int m_x;
    value(int x) : m_x(x) {}
    bool operator<(const value &rhs) const { return m_x < rhs.m_x; }
};

template <class derived>
bool operator > (const compare<derived> &lhs, const compare<derived> &rhs) {
    // static_assert(std::is_base_of_v<compare<derived>, derived>); // Compile time safety measures
    return (static_cast<const derived&>(rhs) < static_cast<const derived&>(lhs));
}

/*  Same goes with other operators
    == :: returns !(lhs < rhs) and !(rhs < lhs)
    != :: returns !(lhs == rhs)
    >= :: returns (rhs < lhs) or (rhs == lhs)
    <= :: returns (lhs < rhs) or (rhs == lhs)
*/

int main() {
    value v1{5}, v2{10};
    cout << boolalpha << "v1 > v2: " << (v1 > v2) << '\n';
    return 0;
}

// Now no need to write comparator operators for all the classes,
// Write only type dependent `operator <` &  use CRTP
```

Usecases

- CRTP widely employed for static polymorphism without bearing the cost of virtual dispatch mechanism. Consider the following code we have not used virtual keyword & still achieved the functionality of polymorphism(specifically static polymorphism).

```
template<typename specific_animal>
struct animal {
    void who() { implementation().who(); }
private:
    specific_animal& implementation() {return *static_cast<specific_animal*>(this);}
};

struct dog : public animal<dog> {
    void who() { cout << "dog" << endl; }
};

struct cat : public animal<cat> {
    void who() { cout << "cat" << endl; }
};

template<typename specific_animal>
void who_am_i(animal<specific_animal> & animal) {
    animal.who();
}
```

- CRTP can also used for optimization as we have seen above it also enables code reusability.

Update: The above hiccup of declaring multiple comparisons operator will permanently be sorted from C++20 by using spaceship(<=>)/Three-way-comparison operator.

5. Virtual Constructor

Intent: To create a copy or new object without knowing its concrete type.
Implementation: Exploits overloaded methods with polymorphic assignment.
Also known as: Factory method/design-pattern.

Problem

- C++ has the support of polymorphic object destruction using it's base class's virtual destructor. But, equivalent support for creation and copying of objects is missing as C++ doesn't support virtual constructor, copy constructors.
- Moreover, you can't create an object unless you know its static type, because the compiler must know the amount of space it needs to allocate. For the same reason, copy of an object also requires its type to known at compile-time.

```
struct animal {
    virtual ~animal(){ cout<<"~animal\n"; }
};

struct dog : animal {
    ~dog(){ cout<<"~dog\n"; }
};

struct cat : animal {
    ~cat(){ cout<<"~cat\n"; }
};

void who_am_i(animal *who) { // not sure whether dog would be passed here or cat

    // How to `create` the object of same type i.e. pointed by who ?
    // How to `copy` object of same type i.e. pointed by who ?

    delete who; // you can delete object pointed by who
}
```

Solution

- The Virtual Constructor technique allows polymorphic creation & copying of objects in C++ by delegating the act of creation & copying the object to the derived class through the use of virtual methods.
- Following code is not only implement virtual constructor(i.e. `create()`) but also implements virtual copy constructor (i.e. `clone()`) .

```
struct animal {
    virtual ~animal() = default;
    virtual std::unique_ptr<animal> create() = 0;
    virtual std::unique_ptr<animal> clone() = 0;
};

struct dog : animal {
    std::unique_ptr<animal> create() { return std::make_unique<dog>(); }
    std::unique_ptr<animal> clone() { return std::make_unique<dog>(*this); }
};

struct cat : animal {
    std::unique_ptr<animal> create() { return std::make_unique<cat>(); }
    std::unique_ptr<animal> clone() { return std::make_unique<cat>(*this); }
};

void who_am_i(animal *who) {
    auto new_who = who->create();// `create` the object of same type i.e. pointed by who ?

    auto duplicate_who = who->clone(); // `copy` object of same type i.e. pointed by who ?

    delete who; // you can delete object pointed by who
}
```

Usecases

- To provide a generic interface to produce/copy a variety of classes using only one class.

6. SFINAE and std::enable_if

Intent: To filter out functions that do not yield valid template instantiations from a set of overloaded functions.
Implementation: Achieved automatically by compiler or exploited using std::enable_if.
Also known as:

Motivation

- **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror is a language feature(not an idiom) a C++ compiler uses to filter out some templated function overloads during overload resolution.
- During overload resolution of function templates, when substituting the explicitly specified or deduced type for the template parameter fails, the specialization discarded from the overload set instead of causing a compile error.
- Substitution failure happens when type or expression ill-formed.

```
template<class T>
void func(T* t){ // Single overload set
    if constexpr(std::is_class_v<T>){ cout << "T is user-defined type\n"; }
    else { cout << "T is primitive type\n"; }
}

int primitive_t = 6;
struct {char var = '4';}class_t;

func(&class_t);
func(&primitive_t);
```

- Imagine if you want to create two sets(based on primitive type & user-defined type separately) of a function having the same signature?

Solution

```
template<class T, typename = std::enable_if_t<std::is_class_v<T>>>
void func(T* t){
    cout << "T is user-defined type\n";
}

template<class T, std::enable_if_t<std::is_integral_v<T>, T> = 0>
void func(T* t){ // NOTE: function signature is NOT-MODIFIED
    cout << "T is primitive type\n";
}
```

- Above code snippet is a short example of exploiting SFINAE using `std::enable_if`, in which first template instantiation will become equivalent to `void func<(anonymous), void>((anonymous) * t)` and second, `void func(int * t)`.
- You can read more about `std::enable_if` here.

Usecases

- Together with `std::enable_if`, SFINAE is heavily used in template metaprogramming.
- The standard library also leveraged SFINAE in most type_traits utilities. Consider the following example which checks for the user-defined type or primitive type:

```
// Stolen & trimmed from https://stackoverflow.com/questions/982808/c-sfinae-examples.
template<typename T>
class is_class_type {
    template<typename C> static char test(int C::*);
    template<typename C> static double test(...);
public:
    enum { value = sizeof(is_class_type<T>::test<T>(0)) == sizeof(char) };
};

struct class_t{};

int main()
{
    cout<<is_class_type<class_t>::value<<endl;    // 1
    cout<<is_class_type<int>::value<<endl;        // 0
    return 0;
}
```

- Without SFINAE, you would get a compiler error, something like "`0` cannot be converted to member pointer for a non-class type `int`" as both the overload of `test` method only differs in terms of the return type.
- Because `int` is not a class, so it can't have a member pointer of type `int int::*` .

7. Proxy

Intent: To achieve intuitive functionality using middleware class.
Implementation: By use of temporary/proxy class.
Also known as: `operator []`(i.e. subscript) proxy, double/twice operator overloading

Motivation

- Most of the dev believes this is only about the subscript operator (i.e. `operator[ ]`), but I believe type/class that comes in between exchanging

data is proxy.
- We have already seen a nice example of this idiom indirectly above in type-erasure(i.e. class `any::inner<>`). But still, I think one more example will add concreteness to our understanding.

operator [ ] solution

```
template <typename T = int>
struct arr2D{
private:
    struct proxy_class{
        proxy_class(T *arr) : m_arr_ptr(arr) {}
        T &operator[](uint32_t idx) { return m_arr_ptr[idx]; }

    private:
        T *m_arr_ptr;
    };

    T m_arr[10][10];

public:
    arr2D::proxy_class operator[](uint32_t idx) { return arr2D::proxy_class(m_arr[idx]); }
};

int main()
{
    arr2D<> arr;
    arr[0][0] = 1;
    cout << arr[0][0];
    return 0;
}
```

Usecases

- To create intuitive features like double operator overloading, `std::any` etc.

Summary by FAQs

When to actually use RAII?

When you have set of steps to carry out a task & two steps are ideal i.e. set-up & clean-up, then that’s the place you can employ RAII.

Why can’t functions be overloaded by return type?

You can’t overload on return types as it is not mandatory to use the return value of the functions in a function call expression. For example, I can just say

```
get_val();
```

What does the compiler do now?

When to use return type resolver idiom?

You can apply return type resolver idiom when your input types are fixed but output types may vary.

What is type erasure in C++?

- Type erasure technique is used to design generic type which relies on the type of assignment(as we do in python).
- By the way, do you know `auto` or can you design one now?

Best scenarios to apply type erasure idiom?

- Useful in generic programming.
- Can also be used to handle multiple types of the return value from function/method(Although that's not recommended advice).

What is the curiously recurring template pattern (CRTP)?

CRTP is when a class `A` has a base class. And that base class is a template specialization for the class `A` itself. E.g.

```
template <class T>
class X{...};
class A : public X<A> {...};
```

It *is* curiously recurring, isn't it?

Why Curiously Recurring Template Pattern (CRTP) works?

I think this answer is very appropriate.

What is SFINAE?

Substitution **F**ailure **I**s **N**ot **A**n **E**rror is a language feature(not an idiom) a C++ compiler uses to filter out some templated function overloads during overload resolution.

What is Proxy Class in C++?

A proxy is a class that provides a modified interface to another class.

Why do we not have a virtual constructor in C++?

- A virtual-table(vtable) is made for each Class having one or more 'virtual-functions'. Whenever an object is created of such class, it contains a 'virtual-pointer' which points to the base of the corresponding vtable. Whenever there is a virtual function call, the vtable is used to resolve to the function address.
- A constructor can not be virtual, because when the constructor of a class is executed there is no vtable in the memory, means no virtual pointer defined yet. Hence the constructor should always be non-virtual.

Can we make a class copy constructor virtual in C++?

Similar to "Why do we not have a virtual constructor in C++?" which already answered above.

What are the use cases & need for virtual constructor?

To create & copy the object(without knowing its concrete type) using a base class polymorphic method.

Do you like it☝️? Get such articles directly into the inbox…!?

Advance C++ Concepts
Advance C++ Concepts
Advance C++ Concepts
Advance C++ Concepts
Advance C++ Concepts
Advance C++ Concepts
Advance C++ Concepts