生命周期与类型安全
Lifetime and Type Safety

现代C++基础
Modern C++ Basics

Jiaming Liang, undergraduate from Peking University

- **Lifetime**
- **Inheritance Extension**
- **Type safety in C++**
- **Type-safe union and void***

Lifetime and Type Safety

Lifetime

Lifetime

- Lifetime is the core of many program bugs.
  - To put it simply, if you use a variable out of its lifetime, then no one knows what happens! All in all, it has been “dead”.
  - For example:

```
const char* rawPtr = std::string{ str }.c_str();
strlen(rawPtr);
```

    - Oops, the temporary variable goes out of its lifetime suddenly after this statement, and the string is invalid afterwards!
  - They may be more subtle than this, e.g.
    - Return reference to local variable, which is destroyed after exiting the function!
    - Accessing reference to some variables in another thread, which has been joint.
    - You capture local variables in lambda by reference, but it goes out of the current scope (like assigning to `std::function&` to parent). ATTENTION!
    - Keep in mind reference used in `std::bind`!
    - …
  - So let’s see what’s lifetime.

Storage duration

- Storage duration means how long the storage used by the object will exist.
- There are four kinds of storage duration:
  - Static storage duration: global variables, static variables in a function/class. It will be constructed before entering `int main`, and destructed after exiting `main`. **Abortion will not call destructors.**
  - Automatic storage duration: variables that belong to a block scope or function arguments. They’ll be constructed when they’re defined, and destructed when the current scope exits (e.g. the function exits).
  - Dynamic storage duration: you can create objects by using new or some other allocation functions. You can only possess the corresponding pointer, and destroying the pointer will not destroy the dynamic memory automatically.
  - Thread storage duration: constructed when the thread is created, and destructed when it exits; we’ll cover it in the future.

Lifetime

- The lifetime of an object begins when the storage with proper alignment and size is allocated and the object is initialized.
  - Including types that can be default-initialized (e.g. `int`, class with a default initializer).
- The lifetime of an object ends when it’s destroyed, or the dtor is called, or its storage is released or reused by non-nested object.
  - If `a` is a member of `b`, then `a` is nested within `b`.
- Particularly, though reference ends the lifetime only when it exits the scope, **using the underlying object that has ended the lifetime is invalid**, which is called dangling reference.
  - This is because accessing reference is same as accessing the underlying object, and accessing out of lifetime is UB.

To be exact, some operations are still allowed, e.g. binding the reference to another reference, accessing static members, but it’s improper to utilize them.

Lifetime

- Temporary objects, as we've seen, are only alive until the statement ends (i.e. until `;`).
  - That's why `const&` or `std::function_ref` is safe to be used as **parameter**, even if the passed functor is temporary.
  - Also, the lifetime of **returned temporaries** can be extended by some references, e.g. we've learnt `const&`.
    - NOTE AGAIN: this requires "returned temporaries"; returned reference or pointer to local variable is still **wrong**.

```
struct A{ };
A bar() { return A{}; };
const A& a = bar();
```

- Corner case: after the ctor/dtor is called and before it finishes, the lifetime of the object has ended, and this is the only place that we can still use its members (with some restrictions).
  - E.g. we've known that we cannot call virtual functions in ctor & dtor.

Lifetime

- So, once you define a variable, it owns the corresponding storage duration and its lifetime begins.
  - However, you can occupy the storage of some objects to create new objects! For example:
    ```
    int buffer[500];
    void* ptr = ConstructOnBuffer<A>(buffer);
    ```
  - The storage still exists, until it exits the current scope; but the original objects in the array (i.e. `int`) have been dead, and lifetime of new object begins.
  - This is just "the storage is reused".
    - You need to manually destruct `A` (without releasing the buffer!) by calling `~A()` before exiting scope.
    - In practice, we've learnt `std::vector`; it will allocate memory (i.e. `capacity`) before inserting elements (i.e. `size`). You cannot use `vec[size]` when `capacity > size` since the lifetime of that element doesn't begin.

Lifetime

```
int buffer[500];
void* ptr = ConstructOnBuffer<A>(buffer);
```

- So, can you use `buffer[0]` to access `int` where new objects are constructed as if it's still a complete `int` array?
  - **Of course not**, the lifetime of element has ended since its storage is reused, and accessing out-of-lifetime object is invalid.
  - So in modern C++, **pointer is far beyond address**; it has type `T*`, and you can hardly ever access some address by it when there are no underlying objects of type `T` alive.
  - **But** you can still use the original array `buffer` to access other elements whose storage is not reused.

```
union U { int a; float b; };

int main()
{
    U u; u.a = 1; std::cout << u.b;
```

- Similarly, for union type, it's **illegal** to access an object that's not in its lifetime (it's only allowed in C)!
  - Here `u.a` is in its lifetime, while `u.b` is not.
  - You should use `std::memcpy` or `std::bit_cast` since C++20 to make them bitwise equivalent.

Placement new

- The `ConstructOnBuffer` is in fact **placement new**, which won’t allocate memory, but only create the object at the place.
  - `new(buffer) Type Initializer`, where `Initializer` is optional.
    - Compared with `new`, it only adds a `(buffer)`.
  - Of course, you need to make sure the alignment satisfies the requirement of the type, so you can use keyword `alignas`.

```
// Statically allocate the storage with automatic storage duration
// which is large enough for any object of type “T”.
alignas(T) unsigned char buf[sizeof(T)];

T* tptr = new(buf) T; // Construct a “T” object, placing it directly into your
                      // pre-allocated storage at memory address “buf”.

tptr->~T();           // You must **manually** call the object's destructor
                      // if its side effects is depended by the program.
                      // Leaving this block scope automatically deallocates “buf”.
```

Preparation…

- Before we go on, we need some preparations…
  - `std::byte`: defined in `<cstddef>` since C++17; it's just an enumeration class and explicitly represents a byte (before we may use `unsigned char`).
  - Trivial dtor:
    - We say a class has a trivial dtor if:
      - It's implicitly declared or declared with `=default`.
      - It's non-virtual and all non-static data members have trivial dtor.
    - For example, they have a trivial dtor:
      - `struct A{ int a; float b; };`.
      - `class A{ int a; public: float b; ~A() = default; };`.
    - And they don't have:
      - `class A{ int a; public: float b; virtual ~A() = default; };`.
      - `class B : public A{};`, since the dtor is still virtual.
      - `class A{ ~A() {}; };`, though dtor does nothing and just behaves like default, it's defined so it's not trivial.
      - `class A{ std::unique_ptr<int> ptr; };`, though the dtor is default, there is at least one data member that doesn't have trivial dtor.

Lifetime

- Corner cases:
  - **Case1**: if you construct an object that has the same type as the original object (ignoring cv-qualifier), and they occupies exactly same storage, then the original name, pointers and references are still valid (it's just very similar to use assignment operators…).
    - **Particularly**, if the alignment or padding is not same, then "exactly same storage" is violated, and it's still invalid.
    - Besides, the original object should not be *const complete object* (i.e. `const T` that is not a normal member, but just a static member or a local or global variable), otherwise compilers will utilize its `const` to optimize (i.e. assume the object never change)!
  - Case2: It's best to reuse storage of plain types like `int` or classes that have trivial dtor. For other types, since exiting the scope will call the dtor, you need to guarantee original objects are still there.
    - So you have to record the original objects before construct new objects, and restore them after you destruct new objects. But why bother???
  - Case3: It's illegal to reuse memory of `const` objects that have determined their value in the compilation time;

Lifetime

  - We’ve learnt in ICS that they may exist in read-only segment of the program, which forbids writing.
  - However, some `const` variables that cannot be determined at compilation time (e.g. `const` member constructed in ctor; or allocated on heap; etc.) is still reusable.
- **Case4**: `unsigned char/std::byte` array is explicitly regulated to be able to *provide storage*.
  - The only difference is that new object is seen as nested within the array, so the array doesn’t end its lifetime even if you occupy the storage by other objects!
  - This property is important for some classes that need a storage with construction of another type.
    - Lifetime ending of partial members will cause the whole object ends the lifetime.
    - So, if you choose to use e.g. an array as buffer so that it’s a member dataset, but it’s not an `unsigned char/std::byte` array, then when you construct the new object, the buffer lifetime ends so that the total class will be out-of-lifetime! Then it’s illegal to continue to use the object.

Lifetime

- **Case5**: It's legal to access the underlying object by pointers without the same type in these cases (*type punning/aliasing*):
  - 1. add/remove cv-qualification, of course.
  - 2. decayed arrays, i.e. a pointer can be used to access array.
  - 3. If the underlying type is integer, then using the pointer of its signed/unsigned variant to access it is OK.
  - 4. If convert it to `(unsigned) char*/std::byte*`, i.e. it's legal to view an object as a byte array.
    - However, in this case, it's possibly illegal to write the element, which may end the lifetime of the original object because of storage reuse.
- **Case6**: If you have an old pointer where you've constructed a new object, but you want to use the old pointer to get the new pointer, you can use `std::launder` defined in `<new>` since C++17.
  - E.g. for our `ConstructOnBuffer()` example, you can also use `std::launder(reinterpret_cast<A*>(buffer))` to get the actual valid pointer.

Strict aliasing rules

- Based on lifetime, we can do optimizations on pointers.
  - For example, it's impossible for `int* a` and `float* b` to refer to the same object, thus the compiler can assume they're different.
    - So `*a += *b; *a += *b;` can be optimized as `*a += 2 * *b;`, without worrying that they refer to the same object so that it's finally `*a += 3 * *b`.
- This is called **strict aliasing rules**, i.e. if pointers are not aliased or not contained as member, then compilers can freely assume that they're different objects.
  - For instance, a pointer to class and to its member type will also not be optimized, e.g. `class A{ int a; float b; }; A*` with `float*`.
- Compilers may optimize it out in the future even if it currently not, e.g. stackoverflow question, Clang14 does it.

You can use –fno-strict-aliasing to disable it.

Strict aliasing rules

You can use `restrict` keyword in C (strangely it's not in C++, but all compilers support it) to show the pointer doesn't overlap with other things, e.g. `uint8_t* restrict target`.

- Notice that `std::(u)int8_t` is just `(un)signed char`, so they will disable this optimization surprisingly. For example:
- Compared with non-member version (i.e. `Unpack(uint8_t* target, char* src, int size)`), it's 15% slower.
- The assembly:

```
mov     rcx,rax
shr     rcx,0x9
and     ecx,0x7
mov     QWORD PTR [rdi+0x18],rcx
```

```
mov     rcx,rax
mov     rdx,QWORD PTR [rdi] // Load, BAD!
shr     rcx,0x18
and     ecx,0x7
mov     BYTE PTR [rdx+0x8],cl
```

```cpp
struct T
{
    uint8_t* target;
    char* source;
    void Unpack(int size) {
        while (size > 0) {
            uint64_t t; std::memcpy(&t, source, sizeof(t));
            target[0] = t & 0x7;
            target[1] = (t >> 3) & 0x7;
            target[2] = (t >> 6) & 0x7;
            target[3] = (t >> 9) & 0x7;
            /* Some other code. */
            source += 6, size -= 6, target += 16;
        }
    }
};
```

- That's because here compiler considers an extreme case, i.e. `target` is an alias of `this`. It will then always reload `target[i]` instead of caching a qword to prevent change of `target`.

Credit: https://stackoverflow.com/questions/26295216/using-this-pointer-causes-strange-deoptimization-in-hot-loop

Lifetime

- Wait, there is one more thing…what about `std::malloc`?
  - Normally, we use `malloc` like this:

```
A* arr = (A*)std::malloc(500 * sizeof(A));
for (int i = 0; i < 500; i++)
    arr[i].a = 1; // and other operations...
```

  - You get a `void*` from the function, what's the underlying object?
    - It's not `A`, since `malloc` only allocates memory!
    - So how can you use `arr[i].a`? There is no object in its lifetime!
  - It's really a shame to say that it's UB before C++20…
    - C++20 adds a small patch, i.e. operations like `std::malloc/calloc/realloc` or allocators will implicitly begin the lifetime, and then you can use operations like above to make objects suitably constructed.
    - Particularly, array of `unsigned char/std::byte` can be used to implicitly create objects too, which means such code is Okay:

```
alignas(float) std::byte arr[20];
float* ptr = reinterpret_cast<float*>(arr);
*ptr = 1.0f;
```

Lifetime

```
alignas(std::max(alignof(float), alignof(int))) std::byte arr[20];
float* ptr = reinterpret_cast<float*>(arr);
*ptr = 1.0f;
int* ptr2 = reinterpret_cast<int*>(arr);
// std::cout << *ptr2; // -> illegal
```

- But such code is still not legal:
  - The reason is still lifetime; a `float` cannot be accessed by a `int*`.
    - But you can `int* ptr2 = new(arr) int{2}`, which then begins the lifetime of `int` and ends lifetime of `float` (so `ptr` is illegal). Then you can normally read `*ptr2`.
    - In a word: read before beginning lifetime is still UB.

- C++23 makes a final refinement, i.e. you can use `T* arr = std::start_lifetime_as_array<T>(void*, size)` for the array, and `T* ptr = std::start_lifetime_as<T>(void*)` for a single object or array with determined size (e.g. `T=int[5]`), so that the implicit lifetime is begun manually.
  - E.g. `std::start_lifetime_as<int>`, `std::start_lifetime_as<int[5]>`, or `std::start_lifetime_as_array<int>(ptr, n)`.

Lifetime

- So, since C++23, those libraries that write their own memory allocators (like we do in ICS) can also allocate memory, and users use `std::start_lifetime_as(_array)` to make objects begin lifetime, and then use them without UB.
    - Or they may provide a template so that they call `std::start_lifetime_as` themselves.
    - This is also the way to start lifetime of types with memory from platform-dependent functions, e.g. `mmap`.
- Of course, not all types can begin lifetime in such a way.
    - We first need to introduce **trivial** special member functions.
        - For normal ctor or move/copy ctor/assignment, if
            - It’s implicitly declared or declared with `=default`.
            - All non-static data members and base classes have trivial one.
            - Class has no virtual base class and virtual member function.

Lifetime

Additionally, trivially copiable type is **exactly** the type that can be safely `std::memcpy`, `std::memmove` and `std::bit_cast`. Otherwise it's not safe to copy byte-wise.

- For dtor, as we've said, if
  - It's implicitly declared or `=default`.
  - It's non-virtual and all non-static data members have trivial dtor.
- Then trivially copyable means:
  - For move/copy ctor/assignment, at least one of them exist and all existing ones are trivial;
  - For dtor, it needs to be trivial.
  - Or if it's not a class, it can be scalar types (like integers), or an array of trivially copyable objects.
  - You can use `std::is_trivially_copyable_v<T>` to check it.

- Roughly speaking, an object can begin its implicit lifetime by `std::start_lifetime_as(_array)` if it's trivially copyable.
  - It may be more complex, but we don't cover it at all here.

Lifetime

- Final word: It’s a pity that C++ uses UB instead of e.g. compilation error to check lifetime problem, which means you can usually use out-of-lifetime objects, but you may get strange result in runtime.
  - This problem is very like Garbage Collection; C++ has no way to implement real Garbage Collection, and also cannot check lifetime in compilation time.
  - Rust just tries to correct that, with stricter rules so that lifetime check can be done. But it’s quite miserable for you to make it compile even if there aren’t lifetime problems, since “stricter rules”.
    - Rust uses ownership to manage lifetime.
  - Flexibility v.s. safety, this is a trade-off.
    - C++ - professional can do better, novices mess up in complex problems; Rust – novices cannot compile.

Attention to lambda lifetime

- We reinforce some concerns about lifetime.

- Lambda lifetime should always be shorter than reference captures!
- Even if you capture by values (i.e. =), it’s possibly wrong!
  - If you capture in the class, then this only captures members by `this`, which may be invalid after destruction!
    - C++20 forces you to capture `this` explicitly, too; or *this to copy all members.
  - That’s why Scott Meyer recommends you to write all captures explicitly…

Attention to view lifetime

- Sometimes you may return a view generated by range adaptor in a function (e.g. `v | stdv::reverse`).
  - So what's the lifetime of view?
  - For lvalue, it's same as the lvalue itself, i.e. `v` here.
    - So, if you create a local variable and return view to it, then invalid.
  - For rvalue, it'll same as return a value, so that it's always safe.
    - E.g. `std::vector{1,2,3} | stdv::reverse`; `std::move(v) | stdv::reverse`.
    - We'll tell you what `std::move` is in the future!
  - The essence is `stdv::all`; all range adaptors will first try to convert a range to view by `stdv::all`.
    - For views, it'll do nothing; for lvalue range, it'll create a `stdr::ref_view`; for rvalue range, it'll create a `stdr::owning_view`.
    - From the name, you'll know the lifetime problem! `ref_view` is equivalent to a reference, while `owning_view` holds the value itself.

Lifetime and Type Safety

Inheritance Extension

Inheritance Extension

- Inheritance Extension
  - Slicing problem
  - Multiple Inheritance

Slicing problem in inheritance

- Observing code:
- Is `student1Ref = student2` same as `student1 = student2`?
- **No!**
- There exists implicit conversion in `student1Ref = student2` so actually it calls `Person::operator=(const Person&)`.

```
int main()
{
    Student student1{ ... }, student2{ ... };
    Person& student1Ref = student1;
    student1Ref = student2;
    return 0;
}
```

- Can decorating `operator=` with `virtual` help?
- **Still, No!**
- `Person::operator=` needs `const Person&` but `Student::operator=` accepts `const Student&`; different param types make `virtual` fail.

Slicing problem in inheritance

- This is called “slicing” because such operation will only affect the base slice but not the initial object as a whole.
- Some real examples in my experience:
- Example1: CVML processor.

```cpp
BaseClass CreateNewObject(std::wstring& label)
{
    if (label == L"Function")
        return Function{};
    else if (label == L"Variable")
        return Variable{};
    else if (label == L"ArrayMember")
        return ArrayMember{};
    else [[unlikely]] // 可能是File，不单独存储其信息
        return BaseClass{};
}
```

Slicing problem in inheritance

- Example2: My uncommitted code in our laboratory's project [TOLD-Radiation method]
- `AudioObjectState` is a derived class of `ObjectState`.

```cpp
// 错误代码
void AudioObjectState::SaveState(ObjectState& state)
{
    state = AudioObjectState {
        .vertices = vertices.cpu(),
        .accTimeStep = accTimeStep,
        .animationTimeStep = animationTimeStep
    };
    return;
}
```

```cpp
// 正确代码:
void AudioObjectState::SaveState(ObjectState& state)
{
    AudioObjectState& actualState = static_cast<AudioObjectState&>
(state);
    actualState = AudioObjectState {
        .vertices = vertices.cpu(),
        .accTimeStep = accTimeStep,
        .animationTimeStep = animationTimeStep
    };
    // 或者:
    // actualState.vertices = vertices.cpu();
    // actualState.accTimeStep = accTimeStep;
    // actualState.animationTimeStep = animationTimeStep;

    return;
}
```

Slicing problem in inheritance

- This problem is so likely to be left out that [cpp core guideline]() suggests that
  - Polymorphic base class should hide their copy & move functions if it has data member, otherwise deleting them.
  - For hiding, make copy & move functions `protected` so derived class can call them.
  - This forces you to use derived type to assign, or return by pointer so polymorphism will be preserved.
  - If you have to implement a copy feature , define a new function like `virtual std::unique_ptr<Base> clone();` to utilize polymorphism.

```cpp
class Base
{
private:
    int num;
public:
    virtual ~Base() = default;
    virtual int GetNum() { return num; }
protected:
    // callable only for derived class.
    Base(const Base&) = default;
    Base(Base&&) = default;

    Base& operator=(Base&&) = default;
    Base& operator=(const Base&) = default;
};
```

Inheritance Extension

- Inheritance Extension
  - Slicing problem
  - Multiple Inheritance

Multiple inheritance

- Multiple inheritance is controversial since its birth.
- C++ provides powerful multiple inheritance syntax, which considers most possible cases and allows you to achieve anything you want.
- However, freedom brings with cost; it’s usually complicated if you use multiple inheritance randomly, and destroys the usability and sometimes causes astonishing things.
- In many other OOP languages like C# and Java, only single inheritance is allowed, while *interface* is provided for a special case of multiple inheritance.
  - It’s called *Mixin Pattern*; this is the usual case and the most recommended way if you want to use multiple inheritance in C++.

Multiple inheritance

- As its name, a class has many base classes (公若不弃，布愿拜为义父). Thus, inheritance is then not a chain, but a graph.

```cpp
// Just a joke:
class Elephant {}; // 大象
class Seal {}; // 海狮
class ElephantSeal : public Elephant, public Seal{}; // 海象
```

- Particularly, multiple inheritance will have a new vptr for a new polymorphic base class.

Multiple inheritance

- It will inherit all of the members of base classes.
  - What if base classes have members with the same name?
  - You need to designate explicitly, just like calling `Base::Method` to call the base class method instead of derived one in other cases.
  - Aughhh…too ugly, remember to use `std::invoke`.
- Sometimes, you may want to directly use the function definition in base classes, e.g.

```cpp
class Elephant { public: void test() {}; };
class Seal { public: void test() {}; };
class ElephantSeal : public Elephant, public Seal{
    // I want to specify ElephantSeal::test == Seal::test!
};
```

```cpp
// Just a joke:
class Elephant { public: int weight; }; // 大象
class Seal { public: int weight; }; // 海狮
class ElephantSeal : public Elephant, public Seal{}; // 海象

int main()
{
    ElephantSeal e;
    e.*(&Elephant::weight) = 1;
    // ambiguous: e.weight;
    return 0;
}
```

Using declaration

- Of course, you can write a `test()` with calling `Seal::test()`.
- But you can just use `using Seal::test!`
  - Not only normal methods; operators, type alias (e.g. `using ValType = int;` then `using Seal::ValType`.) and data member are all OK.

```
class ElephantSeal : public Elephant, public Seal{
    using Seal::test;
};
```

- Notice that this still needs accessibility from derived class, i.e. `Seal::test` cannot be `private`.
- Sometimes this technique is also used in single inheritance, e.g. if ctor just constructs the parent (other things are all default-constructed), then just `using`.
  - Then the ctor of `Child` will accept same params.

```
class Child : public Parent {
    using Parent::Parent;
    int aux = 0;
};
```

Multiple inheritance

- Particularly, if you want to `private` inherit the class, while expose only several methods, you can also use `using`.
- Besides, compiler-generated ones won't be inherited.
  - i.e. here `A a{1,2,3}` is valid, while `B b{1,2,3}` is not.

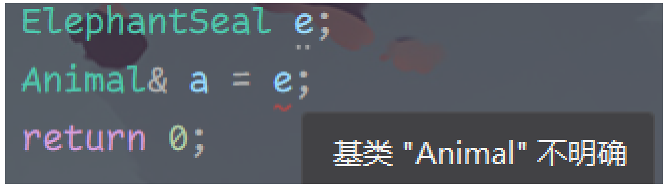
```cpp
struct A {int a, b, c;};
struct B : A {using A::A;};
```

- Ah, let's go back and add more features.
- They're all animals, so what about adding a base class `Animal` for `Elephant` and `Seal`?

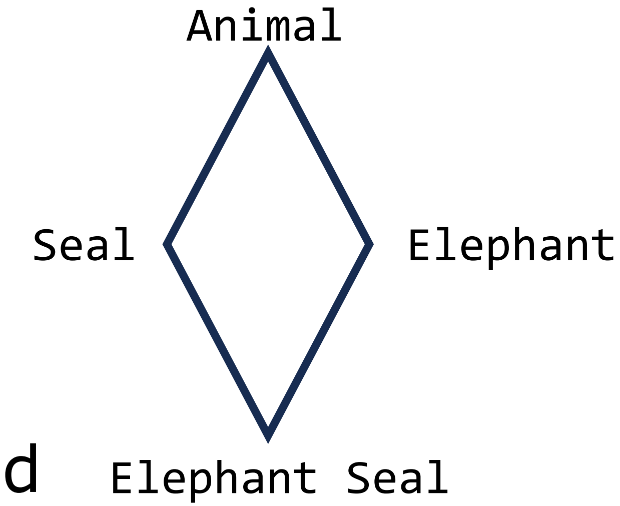
```cpp
class Animal { public: int weight; };
class Elephant : public Animal { }; // 大象
class Seal : public Animal { }; // 海狮
class ElephantSeal : public Elephant, public Seal{}; // 海象
```

Multiple inheritance

- You happily try to reference Elephant seal by the base class Animal, but compile error!



- This is called "dreaded diamond", since the inheritance graph is a diamond.
  - "Ambiguous" means that `Seal` has `Animal` part, and `Elephant` also has `Animal` part, so which `Animal` is referenced?



- The reason lies on that there is a single base class with multiple instantiation.
- So C++ introduces *virtual inheritance*; that is, all virtual bases will be merged and seen as the same.

Multiple inheritance

- Like this:

```
class Animal { public: int weight; };
class Elephant : public virtual Animal { }; // 大象
class Seal : virtual public Animal { }; // 海狮
class ElephantSeal : public Elephant, public Seal{}; // 海象
```

```
ElephantSeal e;
Animal& a = e;
```

- Notice that this may cause some astonishment; You use `A&` to change its member, but members of `B&` also change!
  - Virtual base usually has worse space & time performance, too.
- Anyway, it's usually discouraged to use multiple inheritance with such complexity; Simple ones can simplify your design enough.
- One typical and useful pattern is *Mixin Pattern*.

Multiple inheritance

- That is, you define many ABCs, which tries to reduce data members and non-pure-virtual member functions as much as you can.
  - Of course, the best is that all pure-virtual functions (except for virtual dtor) and no data members, which is simplest and most useful one.
    - Interface in e.g. C# and Java is like this.
- They usually denote "-able" functionality, i.e. some kind of ability in one dimension.
- For example, consider a war simulation game.
  - When you want to use it to show the ability of "attack and defense", then use `Attacker` and `Defender` to refer members.
  - When it's time for users to move, then only movable ones can do it, and non-movable ones may have other functionalities.

Multiple inheritance

- A possible way is:

```cpp
class Attacker { public: virtual void Attack() = 0; virtual ~Attacker() = default; };
class Defender { public: virtual void Defense() = 0; virtual ~Defender() = default;};
class Movable { public: virtual void Move() = 0; virtual ~Movable() = default; };
class NonMovable {};

class Knight : public Attacker, public Movable {};
class Turret : public Attacker, public NonMovable {};
class FloatingCastle : public Defender, public Movable {};
class Headquater : public Defender, public NonMovable {};
```

- So that you can describe a class in multi-dimensions

Lifetime and Type Safety

Type Safety

Category of language

- Statically/Dynamically Typed Language :
  - The type is bound to the variable or not.
    - Static yep, dynamic nope.
  - C++ is a statically typed language.
- Strongly/Weakly Typed Language :
  - **Unfortunately, no universally agreed definition!**
  - Roughly speaking, “strong” means types cannot be converted freely.
    - The language will regulate and its compiler will check them.
  - In this sense, C++ is a weakly typed language, since you can always force the conversion by pointers no matter whether it’s reasonable.
  - However, some conversions will lead to UB (especially conversions related to pointers), which leads to our topic – Type Safety.

Type Safety

- Type Safety
  - Implicit conversion
  - static_cast
  - dynamic_cast and RTTI
  - const_cast
  - reinterpret_cast
  - C-style cast

Implicit conversion

- Some implicit conversions are automatic (i.e. standard conversions), and others are user-defined.
  - We've learnt e.g. `operator float()`. Of course, if you declare it as `explicit`, then implicit conversion will be disabled.
  - Not that important: roughly speaking, implicit conversion will first try standard conversions; and if the expression is still illegal, it will then try user-defined conversions; and if the expression is still illegal, try standard conversions again. Usually if an s-u-s sequence is legal, expression is legal.
- We now cover standard conversions; there are four kinds:
  - 1. Lvalue-to-rvalue conversion, array-to-pointer conversion, function-to-pointer conversion.
  - 2. Numeric promotions and conversions.
  - 3. (Exception-specified) function pointer conversion.
  - 4. Qualification conversions.
  - Standard conversions will try to do them one by one if it's needed to fulfill the expression.

Implicit conversion

- For the first kind:
  - Lvalue-to-rvalue conversion: we haven't covered value category because it's heavily related to move semantics, so just talk about it roughly.
    - Before, what we learn is that lvalues are things that can be LHS in assignment (ignoring `const`), and rvalues cannot (e.g. `2 = a` is illegal).
    - This conversion means that a general lvalue can be seen as a pure rvalue, and intuitively, that's because lvalue can appear at the RHS.
      - We don't dig into that anymore here; just remember general lvalue can be implicitly seen as pure rvalue to make expressions legal.
  - Array-to-pointer conversion: `T arr[N]` can drop its size and be implicitly converted to pointer `T*`.
    - That's why you can do e.g. `arr + 3` like doing pointer arithmetic operations.
    - This is also called "array to pointer **decay**".
    - Notice that only the first dimension is decay-able, e.g. `int a[3][4]` can only be converted to `int* a[4]` implicitly.

Implicit conversion

Non-reference template parameter will also decay, e.g. `(T a, T b)` will never deduce `T = xx&` automatically (though you may explicitly specify it by `<T&>`).

In other words, type deduction of `auto` and type parameter is completely same (except for `std::initializer_list`, template will never deduce it).

- Function-to-pointer conversion: Function type (excluding member functions) can be converted to the pointer to it implicitly.
  - That's why e.g. `FuncPtr a = func` and `FuncPtr a = &func` are both OK, but member functions need explicit `&`.
- **So decay actually means**:
  - Array/Function -> pointer
  - Or for other types, remove references first, remove cv-qualifiers next.
  - You can use `std::decay_t<T>` to get the decayed type.
  - `auto a = xx;` will also decay the deduced type, while `auto&` will not!
    - Sometimes it's called *decay-copy*.
    - But structured binding (`auto [a, b]=xx`) "seems" not decay? When a member is `const`, what you get is still `const`.
      - Remember? It's equivalent to `auto anonymous = xx;` with `a, b` as its member alias, so decay only happens at `anonymous` (which is invisible to users).

Implicit conversion

- 2. Conversions about numeric types:
  - The first one is promotion; promotion has higher precedence and will not lose precision.
  - We've learnt in the first lecture that integer promotion will happen in arithmetic operations, e.g. `b + c` where `b` and `c` are `unsigned short` will in fact silently promote them to `int` and then do `+`.
    - To be exact, promotion happens for pure rvalues; but since lvalue-to-rvalue conversion can happen before, they're also feasible to general lvalues.
    - Integer promotion will promote e.g. `(unsigned/signed) char, short`.
    - Besides, for some character types like `wchar_t` (wide characters, we'll cover them in the future), they will be promoted to the least feasible integer type (but not smaller than `int`).
    - Finally, `bool` can be promoted to `int`, with `false` to be 0 and `true` to be 1.
  - Notice that `char` to `short/long long` is not promotion; the promotion unit is **minimal** one that starts from **`int`**.

Implicit conversion

- The reason why we need promotion is largely due to compatibility with C; C has this obscure setting, so C++ has to have it.
- We say promotion is precedent to conversion:
- There is no ambiguity; it’s the first.
- However, if you change `int` to `long long`, it’s ambiguous since both of them are conversion instead of promotion.
- Finally, for floating point, `float` can be promoted to `double`.

```cpp
void f(int) { std::cout << "Int.\n"; };
void f(short) { std::cout << "Short.\n"; };

int main() {
    char c = 'x';
    f(c);
    return 0;
}
```

- Another one is numeric conversion.
  - The implicit numeric conversion still needs “up-conversion” (i.e. signed to unsigned, smaller to larger), but they may lose precision.

Implicit conversion

- To be exact:
  - Signed value has negative values while unsigned ones don’t, but conversion may happen.
    - `short` -1 -> `long long` will still be -1, while ->`unsigned short/int` will be 0xFF.
  - `float` cannot represent all `int/…`, but `int` can be converted to `float` implicitly.
  - Any scalar types can be converted to `bool`, and it only leads to `true` for non-zero value or `false`.
  - Besides, pointers can be converted to `void*` or pointers to **base** class (if no ambiguity), and `nullptr` can be converted to pointer directly.
  - Pointer to member of derived class can also be converted to pointer to member of base class, i.e. `Derived::int*`->`Base::int*`
- There are also some numeric conversions that need explicit cast (as we’ve learnt in ICS), list here too:
  - Narrow integer conversion will $\bmod 2^n$, i.e. truncate higher bits.
  - Narrow floating conversion will be rounded, e.g. round to nearest.
  - Floating to integer will truncate the digits after dot; **UB** if truncated integer is not representable by the converted type.

Implicit conversion

- Notice: though derived to base class is implicit cast, don't forget slicing problem and if base doesn't have virtual dtor, `delete Base*` is wrong!
- 3. (Exception-specified) function pointer conversion since C++17: we'll cover exception in the future, but basically you can refer to a function with "noexcept" by normal function pointer, but the reversed is not OK.
- 4. Qualification conversion: i.e. you can convert a non-const/non-volatile to a const/volatile one.

- Finally, for e.g. `if(…)`, it's in fact contextual conversion, i.e. the context needs it to be `bool`. This also applies on &&/||/! and some other operations that obviously needs `bool`.

Type Safety

- Type Safety
  - Implicit conversion
  - static_cast
  - dynamic_cast and RTTI
  - const_cast
  - reinterpret_cast
  - C-style cast

static_cast

- To show explicitly the functionality of cast so that users can check it cautiously, C++ divides C-style cast into four parts.
  - `static_cast` is the most powerful one, which can process almost all of normal conversions.
  - `dynamic_cast` is used to process polymorphism specially.
  - `const_cast` is dangerous, since you are trying to see a `const` thing as non-`const`.
  - `reinterpret_cast` is even more dangerous; it's used to convert pointers or references to different types, while lifetime will always forbid you to do insane things by UB.
- C-style conversions will mix them up, which may let you omit the danger of const-correctness and lifetime problem.

static_cast

- So let's demystify `static_cast<TargetType>(Exp)` first!
- 1. It's a kind of explicit conversions, so of course all implicit conversions can be explicitly denoted by `static_cast`.
  - You can also do inverse operations too, even if it's narrow (e.g. `int->short`, `double->float`). We've said how C++ processes narrow numeric conversions.
  - The first and third class of standard conversion is not inversible, i.e. you cannot turn pointer back to function/array.
- 2. Scoped enumeration can be converted to/from integer or floating point, which is same as the underlying integer type.
  - Remember? You can e.g. `enum class A: std::uint8_t{…}`.
  - Notice that enumeration has limit, not whole range of underlying type.
    - Comparison is also the underlying type comparison.

static_cast

- 3. Inheritance-related conversions
  - There are two kinds of conversions: upcast and downcast.
  - As their names, upcast means conversion to base class, while downcast means conversion to derived class.
    - Here we refer to reference or pointer conversion; otherwise it's a new object.
    - Obviously, upcast is safer since a derived object is always a base object; so this is also implicit conversion.
      - But only public inheritance is convertible; private one is not.
    - Downcast is dangerous so it needs explicit conversion; you **must** ensure the original object is just the derived object, so that the pointer/reference is safe.

```
B b;
A& bRef = b; // implicit is OK.
B& bRef2 = static_cast<B&>(bRef);
```

      - If e.g. the original one is just a base object, then UB (of course, since there is no derived object in its lifetime!).
      - It won't do any check, so be certain if you really want!
      - Virtual base or ambiguous base cannot be downcast, too.

Preparation…

- Before going on, we first introduce another property of class…
- A class is said to be **standard-layout**, if:
  - All non-static data members have the same accessibility and are also standard-layout.
    - This is because the layout of members that have different accessibility are unspecified (before C++23); e.g. as the sequence of declaration or first all public members and second all private members.
  - No virtual functions or non-standard-layout base classes.
  - The base class is not the type of the first member data.
  - There is at most one class in the inheritance hierarchy that has non-static member variable.
    - That's because layout of inheritance is not regulated.
- Purpose of this property: If a class is standard-layout, then its layout is same as `struct` in C (as we've learnt in ICS!).

Preparation…

You can use `std::is_standard_layout_v<T>` to check whether `T` is standard-layout.

- Examples:

```
struct A { int a, b; }; // right, just like in C.
class B { int a, b; }; // right, all private
class C { public: int a, b; }; // right
class D { int b; public: int a; };
// no, non-uniform access control.
class E { int a, b; virtual ~E() = default; }; // no, virtual
class F : E { }; // no, base class is not standard-layout.
class G : E { E f; }; // no, first member is base class.
class H : A { static int a; }; // right
class I : A { int b; };
// no, both base and derived have their own non-static members.
```

- Just remember, those classes that are close to C `struct` (almost no inheritance, no polymorphism, same access control) are standard-layout.

static_cast

- 4. Pointer conversion
  - In C, you can convert freely among any types of pointers.
  - C++ discourages that, so it splits the whole bunch of things.
  - For `static_cast`, besides inheritance-related pointer conversion, it also processes `void*`.
    - You can convert any **object** pointer to `void*` (this is also implicit conversion).
    - You can also convert explicitly `void*` to any object pointer.
    - **BUT**, this requires the underlying object of type `U` and the converted pointer `T*` (ignoring cv) to have the relationship (called *pointer-interconvertible*) as:
      - `T == U`.
      - `U` is a union type, while `T` is type of its member (though using it still needs this member to be in its lifetime).
      - `U` is standard-layout, while `T` is type of its **first** member or its base class.
      - Or all vice versa/transitivity (i.e. you can swap `T` and `U` above; after all, "inter").

static_cast

For instance, `static_cast<Base*>(void*)` where `void*` is in fact `Derived` is not safe. **Non-pointer-interconvertible cast will keep the original address**, but it's not determined (especially in multiple inheritance) that value of `Base*` is same as `Derived*`.

- Notice that pointer conversion doesn't guarantee the stored address stays same; it just guarantees that you get the corresponding pointer (for example, pointer to the first member of standard-layout struct).
  - Again, pointer is not address itself in modern C++!
  - Particularly, if you convert `T*` to `void*`, and back to `T*` (this is the usual way in C to implement pseudo-template), you'll just get the same pointer as the original one.
- Final word: you need to pay attention to alignment; same types with different alignment should guarantee the pointer fulfills the converted alignment.

static_cast

- There are several small functionalities, just list here.
- 5. Convert to `void`: just discard value, nothing happens.
  - In C, you may have seen things like `(void)(SomeExp)`.
    - This is to disable compiler’s warning when the variable seems not used (but in fact useful), or `[[nodiscard]]` return value needs discarded in fact.
    - The former can use `[[maybe_unused]]`, the latter cannot.
  - Notice that `operator void()` won’t be called.
- 6. Construct new object: if the object ctor can accept a single parameter, which is convertible from the expression, then it in fact constructs a new object.
  - E.g. `static_cast<A>(1)` for `A(int)`.
- 7. Transform value category; we’ll talk about it in *Move Semantics*.

static_cast

- Final word: `static_cast` can be used to specify which overload of functions is used by function-to-pointer conversion.
  - If you don't write the cast, compile error.

```cpp
int MyReduce(int a, int b) { return a + b; }
float MyReduce(float a, float b) { return a + b; }

int main()
{
    std::vector<int> a;
    std::partial_sum(a.begin(), a.end(), a.begin(),
        static_cast<int(*)(int, int)>(MyReduce));
    return 0;
}
```

- Summary:
  - Implicit conversion and their inverse operation, except for array/function decay.
  - Enumeration conversion.
  - Inheritance-related conversion.
  - Pointer conversion.

Type Safety

- Type Safety
  - Implicit conversion
  - static_cast
  - dynamic_cast and RTTI
  - const_cast
  - reinterpret_cast
  - C-style cast

Dynamic cast and RTTI

- We've seen that `static_cast` doesn't check validity along inheritance chain; you can cast to the derived class even if there is no actual one in its lifetime.
  - It just uses UB to regulate, but that's too weak!
- `dynamic_cast` tries to solve that; the conversion will fail when it's inappropriate.
  - To be specific, reference conversion failure will `throw std::bad_cast` exception, while pointer conversion failure will return `nullptr`.
  - This is stronger than UB, and more convenient to find bugs!
- So, to do type check in run time, RTTI (*Run-Time Type Information/Identification*) is preserved.
  - Notice that `dynamic_cast` can only be used in polymorphic types, since RTTI relies on things like virtual pointer.

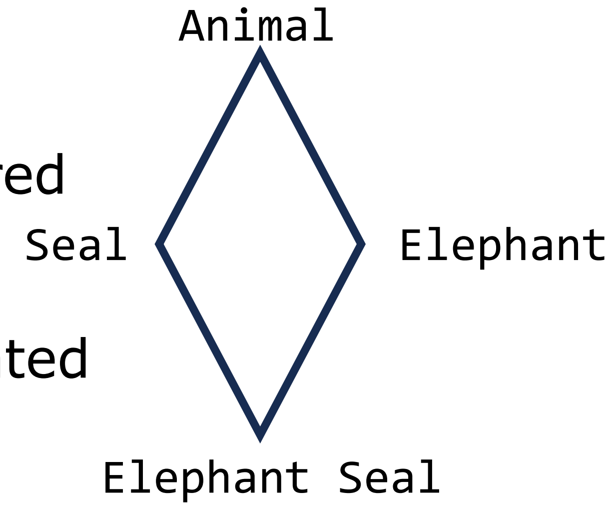
dynamic_cast

```cpp
void func(Shape* shape)
{
    Circle* circle = dynamic_cast<Circle*>(shape);
    // cast failure for it's not a circle.
    if(circle != nullptr)
    {
        // ... do things for circle.
    }
    // ... else other code.
}
```

- However, safety comes with cost.
  - `dynamic_cast` that uses RTTI is at least 10 times slower as the `static_cast`, and can be even hundreds or thousands slower in some cases!
  - The former happens when it's a inheritance chain and converts to the exact underlying type (usual case); the latter happens for "*sidecast*" in a inheritance graph (i.e. multiple inheritance).
  - Often, `dynamic_cast` means some defects in design, i.e. the original type cannot represent all behaviors by polymorphism; so it's usually better to not use it frequently.
    - Performance-critical projects may write their own downcast (e.g. LLVM).
- 1. You can do downcast in polymorphic types.
  - We assume that the real type of underlying object is `T`, and we now has a `B*`.
  - If we convert `B*` to another type which doesn't derive `T`, fail.
  - Otherwise, the conversion is successful, and you can get the pointer.
  - Notice that `dynamic_cast` shouldn't be used in ctor/dtor, since its vptr is not complete.

dynamic_cast

- 2. You can also do sidecast in polymorphic types with multiple inheritance.
  - For example, we have an Elephant Seal, which is referred by `Elephant&`.
  - Now you need to use it as `Seal&`; you cannot use e.g. `static_cast<Seal&>` since `Seal` and `Elephant` aren't related directly.
    - i.e. the inheritance path is incomplete after upcast; you cannot know whether it's an elephant seal.
    - So you have to use `static_cast<ElephantSeal&>`, and implicitly casts it to `Seal&`.
  - This is efficient, but in some complex cases, you may not know the exact type so it's dangerous, then casting to a side path directly is needed.
  - Here, you can just use `dynamic_cast<Seal&>`.



dynamic_cast

- The inner process is still converting it to the underlying type (i.e. here `ElephantSeal`), and upcast.
  - Of course, it still needs to check whether the upcast is ambiguous.
  - If upcast fails, the whole conversion fails.
- Some other minor functionalities:
  - 3. You can use `dynamic_cast` to convert a pointer to the most derived class (i.e. the pointer to the underlying object).
    - By `dynamic_cast<void*>(…)`.
    - But this is rarely used, since you still need the exact type to `static_cast` because `void*` cannot do nothing.
  - 4. You can also convert from derived class to base class, just like implicit conversions.
    - These two don't need RTTI, so as fast as `static_cast` (though useless…).

If you're interested in how `dynamic_cast` is implemented in GCC and Clang, you can see [这下可以安心使用 dynamic_cast 了：dynamic_cast 的实现方法分析以及性能优化 - Lancern的文章 - 知乎]; and their time cost, see [Github].
The author even improves the implementation and commit it to LLVM!

RTTI

- C++ also provides a way for you to utilize the underlying runtime type information directly by `<typeinfo>`.
  - Still, use it in restricted session, since it's costly in performance.
- You can use operator `typeid(xxx)` to get `const std::type_info`.
  - This operator is like `sizeof`.
  - `std::type_info` can compare equality (i.e. check whether two types are same), call `.name()` to get its name, and call `.hash_code()` to get hash, and `.before()` to compare in strong order (platform-dependent).
    - But it cannot be directly used in associative containers, since it doesn't specialize `std::hash/operator<`.
  - If you really want to use it as keys in associative container, you can use `std::type_index` defined in `<typeindex>`, which is just a wrapper.
    - It accepts `std::type_info` as parameter of ctor.

RTTI

- Notice that `type_info` objects are read-only; you cannot copy them or construct them.
  - i.e. RTTI information is totally held by the compiler.
- You can use it to debug, especially by `.name()`.

- Finally, RTTI is unfriendly to shared library (i.e. cross "*module boundary*").
  - E.g. since GCC 3.0, symbols are compared equality by address instead of names. So to preserve only one symbol across many `.obj` file, it merges them when linking (like in static library).
  - However, shared library usually uses runtime loading (e.g. `dlopen`, as we've seen in ICS).

RTTI

- So to load shared library quickly, many procedures are omitted, which includes resolving different RTTI symbols.
  - e.g. If you don't `| RTLD_GLOBAL` in `dlopen`, it'll be `RTLD_LOCAL` **by default**.
  - Then, symbols are not relocated against other symbols, then you get two different RTTI symbol addresses!
- So when you construct an object in the program, while it's passed into a function loaded from the shared library where `dynamic_cast` happens, then you'll get unexpected behaviors.
  - That is, even if you pass into a `Base*` with an underlying `Derived` object, `dynamic_cast<Derived*>` will return `nullptr`. That's because `dynamic_cast` finds the program's RTTI symbol address, and comparing it with the library's RTTI symbol leads to inequality (i.e. conversion failure).
  - Similarly, constructing an object from the library and grasping it in the program and `dynamic_cast` it is also dangerous.

RTTI

- `typeid` comparison has similar problems; you may refer to [GCC doc] for how to solve it.
- Final word: anyway, RTTI is slow no matter in runtime or loading time (to use it with crossing module boundary), which is discouraged in many projects.
  - E.g. Qt uses `qobject_cast`.
  - Remember C++ philosophy "Only pay for what you use"? RTTI is seen to violate this rule by many.
- All in all, use RTTI carefully, and rethink whether your design can be improved if you want to use it, especially when you're writing a library or performance-critical sessions.

Type Safety

- Type Safety
  - Implicit conversion
  - static_cast
  - dynamic_cast and RTTI
  - const_cast
  - reinterpret_cast
  - C-style cast

const_cast

- As its name, it tries to drop the cv-qualifiers.
- This is of course dangerous; you’ve marked it as “unchangeable”, it usually has its own consideration.
- It’s almost only restricted to be used in these two scenarios:
  - When you **explicitly** know it’s not read-only initially, and parameters force you to accept a `const` one, but you in fact need a non-`const` one.
    - For example, in a FTP program I’ve written.
      - Theoretically it’s unnecessary to change the user’s command, so processing handles use `std::string_view`, which is a `const char*`.

```
std::getline(std::cin, userCommand);
std::string_view commandView(userCommand);
std::invoke(handle, &processor, commandView.substr(headEnd));
```

      - However, only one exception: command open, i.e. `open IP port`.
      - For socket library (we’ve learnt it in ICS), it needs `‘\0’` end to read in an IP, so to reduce copy, I decide to overwrite the space between IP and port to be `‘\0’`.
      - So I use `const_cast` here; this view is in fact original `userCommand`, which is allocated on heap and not read-only.

```
const char* nativePtr = commandView.data();
const_cast<char*>(nativePtr)[IPend] = '\0';
```

const_cast

- The second case is when you use library; the author forgets the `const` in parameter, but it in fact doesn’t write it (which is **explicitly** documented or you can view the source code).
  - Then Okay, use it.
- `volatile` is similar; stripping it will make writing through this pointer not definitely visible to memory, which disobeys the principle of `volatile`.
- If you write a read-only variable through `const_cast`, then it’s UB.
  - E.g. `const int a = 2; const_cast<int&>(a) = 1;`
- All in all, it’s not enough no matter how much attention you pay when you want to use `const_cast`.

Type Safety

- Type Safety
  - Implicit conversion
  - static_cast
  - dynamic_cast and RTTI
  - const_cast
  - reinterpret_cast
  - C-style cast

reinterpret_cast

- `reinterpret_cast` is used to process pointers of different types, which is dangerous because of lifetime.
- 1. converting from an object pointer to another type, i.e. `reinterpret_cast<T*>(xxx)`.
  - This is same as `static_cast<T*>(static_cast<(cv) void*>(xxx))`, so you still needs `xxx` and `T*` to be *pointer-interconvertible*.
  - Remember? If you want to convert an old pointer that loses its lifetime to a new pointer, you may also need to use `std::launder`.
- 2. converting from a pointer to function to another type of pointer to function; or pointer to member to another one.
  - Except for implicit ones (e.g. to base class), only round-trip conversion is valid. i.e. If you want to call the function, you must convert it back to the original type.

reinterpret_cast

    - In many platforms (e.g. POSIX ones), it’s legal to convert them to `void*` and back to get the original pointer; but it’s not forced so this round-trip conversion may compile error in some obscure systems.
    - Pay attention to possible alignment too.
- 3. converting pointer to integer or vice versa.
    - A pointer can be converted to integer by `reinterpret_cast` if the integer is large enough.
        - In C++ `<cstdint>`, `std::uintptr_t` is defined as such integer type.
    - This integer can be converted back to get the original pointer.
        - If you pass into a random integer, or one with insufficient size, then UB.
    - Finally, `reinterpret_cast` from 0/NULL is UB; just use `nullptr`/implicit conversion/`static_cast`.
- 4. reference is also convertible; it’s same as converting pointer.

reinterpret_cast

- You may find that its functionality is hugely restricted due to lifetime.
  - More loosely, aliasing ones are also okay, as we’ve learnt in strict aliasing rules, e.g. you can use `reinterpret_cast<unsigned int&>` to refer to `int`.
  - You cannot do e.g. `reinterpret_cast<float&>` to view the binary (as you might do before); you need `std::bit_cast` or `std::memcpy`.
  - But these regulations are all UB-related, which means it doesn’t forbid you to do illegal things. So be careful if you really use it!

Type Safety

- Type Safety
  - Implicit conversion
  - static_cast
  - dynamic_cast and RTTI
  - const_cast
  - reinterpret_cast
  - C-style cast

C-style cast

- It’s discouraged to use such explicit cast in C++, so here we just talk about it a little.
- `(Type)x` or `Type(x)/Type{x}` are both OK, but the second looks similar to constructing new object or declaring new function, which will cause ambiguity sometimes.
  - We don’t cover the resolution of ambiguity, just use the first way.
  - BTW, C++23 adds a patch, i.e. you can use `auto{…}/auto(…)` to get a decayed pure rvalue of the expression. It’s usually used to mean decay copy in the standard wording.
- The functionality is like:
  - `const_cast` first.
  - `static_cast + const_cast` (and can omit access control, i.e. private inheritance doesn’t affect C-style cast).
  - `reinterpret_cast + const_cast`.

Lifetime and Type Safety

Type-safe union and void*

Type-safe union and void*

- We’ve known that `union` and `void*` are quite dangerous.
  - Due to lifetime, you can only use the active member in the union; use another one is UB.
  - Similarly, if you convert `void*` to a non-pointer-interconvertible one, it’s also UB.
  - Oh gosh, UB, UB, why isn’t there some clear signal of that?
- Since C++17, `<variant>` and `<any>` are introduced to guarantee the safety.
  - `std::variant` can be seen as a `union` with a `size_t` index, which will inspect whether the member is in its lifetime when getting.
  - `std::any` can be seen as a `void*` with the original type “stored” magically, so that you’ll fail to grasp the inner object with the wrong type.
  - Their functionalities are also enriched.

Union behavior*

- In C++, we have ctor/dtor, which makes union more complex.
- Since union itself doesn't know which member is active:
  - If a member doesn't have trivial special member functions, then union has no corresponding functions.
  - You then need to define them yourself (without knowing which member is active), which is impossible…
  - The only viable way is to define a struct with both a union and tag (i.e. showing which member is active), and define special member functions as a whole.

Type-safe union and void*

- Type-safe union and void*
  - variant
  - any

variant

Reference/C-array/incomplete type (like `void`) cannot be member types.

- You can define the union "members" (called "alternative" in `variant`) by template:

```cpp
std::variant<int, float, int, std::vector<int>> v{ 1.0f };
```

```cpp
union Union {
    int _1;
    float _2;
    int _3;
    std::vector<int> _4;
};
```

- For construction:
  - By default, the first alternative is value-initialized.
  - You can also assign a value with the same type of some alternative, then that's the active alternative.
  - You can also construct the member in place, i.e. by `(std::in_place_type<T>, args…)`.
    - This is similar to `std::move_only_function`, which is fit for construction-only class.
    - If there are more than one alternative of this type, then the above two are disabled.
  - You can construct by index, i.e. `(std::in_place_index<Index>, args…)`.
    - E.g. `(std::in_place_index<3>, 4, 1)` to construct a vector with four 1.
- Only when all alternatives support copy/move will the variant support copy/move.

variant

- To access or check the existence of alternative, methods below are provided:
  - `.index()`: return the index of active alternative.
  - These methods need the examined type **unique** in type params:
    - `std::hold_alternative<T>(v)`: return Boolean that denotes whether the active alternative is of type `T`.
    - `std::get<T>(v)`: return the active alternative of type `T`. If the active one is not of type `T`, exception `std::bad_variant_access` is thrown.
    - `std::get_if<T>(v)`: return the **pointer** to the active alternative of type `T`. If the active one is not of type `T`, return `nullptr`.
  - If not unique, you can also use index-based access:
    - `std::get<I>(v)/std::get_if<I>(v)`.
    - The reason why ctor uses `std::in_place_index` is that ctor cannot specify template parameter when calling!
  - Through these methods, member access is type-safe.

variant

- Besides type safety, the most important extension of `std::variant` than `union` is that it implements ***visitor pattern***.
  - This is still for template parameters with all unique types.
  - Visitor pattern roughly means that you don't need to know who is the visitor; it just needs to know what the visitor should do!
  - For `std::variant`, it means you don't need to care about which alternative is valid; you just need to prepare overloaded methods for each possible alternative, and it'll be called according to the underlying object.
  - This is very like polymorphism; we don't care about `Base&` is `Base, Derived` or `MoreDerived`; we just use virtual functions to always call the right function.
    - But more powerfully, it doesn't need these types to be related by inheritance!
- Take an example of my previous code.
  - In OpenGL, a "framebuffer" (like what your screen shows!) has many color attachments, which can be either a "texture" or "renderbuffer".

BTW, the performance of mainstream implementation of `std::visit` is $O(1)$.

variant

- They're all represented by an `unsigned int` descriptor.
- So, I create a `std::vector<AttachType>` like this:

```
struct RenderBuffer { unsigned int buffer; };
struct RenderTexture { unsigned int buffer; };
using AttachType = std::variant<RenderBuffer, RenderTexture>;
```

- Then, when the user wants to get the underlying descriptor, you can easily achieve it:

```
unsigned int GetBufferFromAttachType_(const AttachType& buf) const {
    return std::visit([](const auto& arg) { return arg.buffer; }, buf);
}
```

- Similarly, to construct these buffer, `std::variant<RenderBufferConfig, TextureConfig>` is used.
- Then just call by overload:

```
void Framebuffer::GenerateAndAttachColorBuffer_(RenderBufferConfigCRef ref, int id) {
void Framebuffer::GenerateAndAttachColorBuffer_(TextureConfigCRef config, int id) {
```

```
for (int i = 0, len = static_cast<int>(colorConfigs.size()); i < len; i++)
{
    std::visit([this, i](auto&& arg) {
        GenerateAndAttachColorBuffer_(arg, i);
    }, colorConfigs[i]);
}
```

variant

- There is a special state for `std::variant`; when an exception is thrown during assignment, then possibly no valid state exist!
  - The original value loses, while the new value isn't constructed.
  - After that, `.valueless_by_exception()` returns `true`, and the `.index()` will be `std::variant_npos` (i.e. `static_cast<size_t>(-1)`).
  - You need to re-assign a valid value to make it valid again.
- If you need an explicit "empty" state, you can use the type `std::monostate`.
  - For example, "depth buffer" can be absent, so I use:

```
std::variant<std::monostate, RenderBufferConfigCRef, TextureConfigCRef>
std::visit([this](auto&& arg) {
    using T = std::remove_cvref_t<decltype(arg)>;
    if constexpr (!std::is_same_v<T, std::monostate>)
        GenerateAndAttachDepthBuffer_(arg);
}, depthConfig);
```

It's Okay if you don't understand the function; just remember it can be combined with template, which will be covered in the future!

variant

- Finally, there are some other helpers:
  - `.emplace<T>(args…)` or `.emplace<Index>(args…)`, which will destroy the original alternative and construct new alternative in place.
    - `operator=` will call assignment operator instead of dtor + ctor if the alternative index remains same.
  - `.swap(v2)/std::swap(v1,v2)`.
  - Comparisons, which needs every alternative comparable.
    - If the indices are not same, then it in fact compares indices (particularly, `std::variant_npos` is seen as smallest, `<` is always `true`.).
    - If they're same, then it will compare the underlying object.
  - Hash, which needs every alternative hashable.
    - It's not same as hashing the underlying object; it may additionally consider index!
  - `std::variant_size_v<V>`: get the number of alternatives in the `variant`.
  - `std::variant_alternative_t<Index, V>`: get the type of the `Index`th alternative.

Type-safe union and void*

- Type-safe union and void*
  - variant
  - any

any

```cpp
std::any a{ 1 };
a = 2.0f;
a = "test";
```

- You can use it to load any object, like:
  - To be exact, `std::any` needs the underlying object to have a copy ctor.
    - The type will also be decayed.
  - You can also default-construct it or call `.reset()`, then it holds nothing.
    - You can use `.has_value()` to check it.
  - For ctor, you can also use `(std::in_place_t<T>, args…)`;
- When you need to get the underlying object, you need to use `std::any_cast<T>(a) or (&a)`.
  - This is a function, not a keyword.
  - Except that `T` can add cv-qualifier and reference, you must cast to the exact same type as stored!
    - E.g. `a = 1LL` cannot use `std::any_cast<int>(a)`; you must use `static_cast<int>(std::any_cast<long long>(a))`.
    - Otherwise, exception `std::bad_any_cast` will be thrown (for `&a`, i.e. pass a `std::any*` to `std::any_cast<T>`, `nullptr` will be returned.).
    - So it's not like e.g. variable in Python; it's just a safe `void*`, you still need the exact type!

any

- `std::any` can have SBO (small buffer optimization) like `std::function`.
  - Normally, `sizeof(std::any)` should be `sizeof(void*)`.
  - But due to SBO, GCC is 16 bytes, Clang is 32 bytes, and MSVC is 32/64 bytes (depend on x86/x64)!
  - That is, small objects that are “noexcept move-constuctible” will be stored on stack directly instead be allocated on heap, which will reduce time cost a little.
- Finally, some helpers:
  - `.swap/std::swap/.emplace`, just like `std::variant`;
  - `.type()`, as if `typeid` of the underlying object.
  - `std::make_any()`, same as constructing `std::any`.

Summary

- Lifetime
  - Storage duration.
  - Storage reuse and corner case.
  - Type punning and strict aliasing rules.
  - unsigned char and std::byte, special ones!
  - Implicit lifetime and std::start_lifetime_as(_array).
- Inheritance Extension
  - Slicing problem
  - Multiple Inheritance
- Type safety
  - Implicit conversion
  - static_cast
  - dynamic_cast and RTTI
  - const_cast
  - reinterpret_cast
  - C-style cast
- Type-safe union and void*
  - std::variant, visitor pattern
  - std::any.

Next lecture…

- We'll cover programming in multiple files…
  - Header files, source files, and how they interact with functions and variables…
  - How to make libraries with build tools
  - A taste for modules