错误处理
Error Handling

现代C++基础
Modern C++ Basics

Jiaming Liang, undergraduate from Peking University

- **Error Code Extension**
- **Exception**
- **Assertion**
- **Debug Helpers**
- **Unit Test**

Error Code

- In C, if a function may fail to finish its task, users will know that by error code.
  - But error code has many variations:
  - By return value: return 0 if success, 1 if fail for insufficient memory, 2 if network connection fails, etc.
  - By special value: like `EOF` returned by `scanf`, `nullptr` in `fopen`.
  - By pointer: like `strtod(begin, &end)` and `end == begin`.
  - And they sometimes cooperate with `errno`, i.e. a global error number and can use some functions to give an explicit message.
    - For example, `strtod` may set `ERANGE` if the value isn't representable.
    - BTW, `errno` is unique for any thread since C11, so it's safe to use without race.

Error Code

- But error code has its limitation:
  - Users often have to pass a pointer to accept the real return value.
  - Users have to know different behaviors from the document, e.g. some functions use non-zero as success while some use zero.
  - If users cannot process the error, it has to return the error code to the caller again, which needs manual propagation.
  - Sometimes you only need -1 as invalid, which will consume half of unsigned values.
  - Error information is usually limited.
  - …

Error Handling

- So how do other languages handle errors?
- Go: support multiple return value and the second one is err.
  - You need to omit it explicitly by `result, _ = xx`.

```go
a, b := 10, 0
result, err := Divide(a, b)
if err != nil {
    switch {
    case errors.Is(err, ErrDivideByZero):
        fmt.Println("divide by zero error")
    default:
        fmt.Printf("unexpected division error: %s\n", err)
    }
    return
}
```

- Rust: by two mechanisms
  - `Result<T, Err>` for recoverable errors.
  - `panic!` for unrecoverable errors (exit with stack trace).

```rust
let username_file_result = File::open("hello.txt");

let mut username_file = match username_file_result {
    Ok(file) => file,
    Err(e) => return Err(e),
};
```

- Python and many other languages: exceptions.
- C++ provides similar solutions.

Error Handling

Error code extension

Error Handling

- Error code extension
  - optional
  - expected

Optional

Defined in <optional>, since C++17

- First, `std::optional<T>` is not strictly a way for "error handling", but to denote "`T` is allowed and legal to be absent".
  - It's even not that recommended to use it to handle error in C++, but it's somewhat similar to `std::expected`, so we mention it here.
  - It uses an additional `bool` to denote "exist or not", so for some not-nullable types (like integers), you can utilize any of its values instead of introducing a special value for "error".
    - Empty value is then introduced as `std::nullopt`, which essentially makes the underlying `bool` to be `false`.
- For example, you may add a `Get()` method for `std::map` which returns a `std::optional<T>` in case that "key doesn't exist", instead of using `.find()` and judge whether the iterator is `end()`.
  - That's what `Map` in Java / `dict` in Python do.

Optional

```cpp
template<typename Key, typename Val>
std::optional<Val> Get(const std::map<Key, Val>& map, const Key& key)
{
    if(auto it = map.find(key); it != map.end())
        return it->second;
    return std::nullopt;
}
```

- So let's have a look at its basic methods!
  - Ctor/`operator=/swap/emplace/std::swap/std::make_optional`.
    - Ctor can also accept `(std::in_place, Args to construct T)`.
      - Since type is deterministic, it doesn't uses `std::in_place_type<T>` like variant/any.
    - By default, it's constructed as `std::nullopt` (similar to `nullptr`, denote "no value").
      - You can use also `= std::nullopt`, or `.reset()` to make an `optional` null.
  - `operator<=>`; Similarly, no value is considered as smallest.
  - `std::hash`; unlike `std::variant`, it's guaranteed for `std::optional` to have the same hash as `std::hash<T>` if it's not `std::nullopt`.

Optional

- You can just use `std::optional` as a nullable pointer:
  - `operator->/operator*/operator bool`, as if a `T*`.
    - The behavior is undefined for `->/*` if it's in fact `std::nullopt`.
  - You can also use methods `.has_value()` (just same as `operator bool`) and `.value()` (which will throw `std::bad_optional_access` instead if `std::nullopt`).
  - `.value_or(xx)` can provide a default value.
    - E.g. `std::optional<double> opt{1.0}`, then `opt.value_or(3.0)` is also `1.0`; but if `opt` is `std::nullopt`, then it returns `3.0`.
- Beyond pointer, optional itself is self-explainable and can accept an temporary.
  - That's still naïve, we need more convenient operations!

Monad

- Monadic operations in C++23 are for that!
  - It’s a concept from category theory（范畴论） in mathematics and propagated by functional programming.
  - Basically, monadic operations will transform a `Maybe<T>` to a `Maybe<U>`, so that all operations can be chained no matter whether there is a value.
    - Here `Maybe` is just `std::optional`.
  - There are two operations, one for transforming normal value and another for null value; if it doesn’t match, then this operation will be jumped.
  - For normal value:
    - `.and_then(F)`, where `F` accepts `T` or its references and returns `optional<U>`;
    - `.transform(F)`, where `F` accepts `T` or its references and returns `U` (automatically wrapped as `optional<U>{retval};`).
  - For nullable value:
    - `.or_else(F)`; since it’ll only be called when `nullopt`, `F` doesn’t need any parameter, but just returns an `optional<T>`.

Monad

- For example:

```cpp
std::optional<UserProfile> fetchFromCache(int userId);
std::optional<UserProfile> fetchFromServer(int userId);
std::optional<int> extractAge(const UserProfile& profile);

int main() {
    const int userId = 12345;

    const auto ageNext = fetchFromCache(userId)
        .or_else([&]() { return fetchFromServer(userId); })
        .and_then(extractAge)
        .transform([](int age) { return age + 1; });

    if (ageNext)
        cout << format("Next year, the user will be {} years old", *ageNext);
    else
        cout << "Failed to determine user's age.\n";
}
```

Credit: CppStories.

Optional

- Note1: most of types in Java and C# are nullable, which makes them “optional” automatically.
  - However, this makes them inefficient in many occasions.
- Note2: `std::optional` (& `std::expected`), `std::any` and `std::variant` are sometimes called “vocabulary type”.
  - All of them **cannot** use reference type for template parameter (e.g. `std::optional<int&>`).
    - unless P2988?
- Note3: though `std::optional` only stores an additional `bool`, the alignment and padding will make it in fact larger.
  - So a bunch of `std::optional` in a struct may be sub-optimal; it’s better to store `bool`s and values separately.

Containers

- Error code extension
  - optional
  - expected (since C++23)

Expected

- `std::expected` is very similar to `std::optional`, except that:
  - It uses an Error type (i.e. `std::expected<T, E>`) instead of “null” to denote absent value.
  - And it’s recommended to be used to denote error, i.e. absence means illegal result. So it should be the correct one to handle errors.
    - Just similar to `Result` in Rust.
- You may construct it with:
  - For normal value, just use `T`, or `std::in_place` with arguments.
  - For error value, you can use `std::unexpected{xx}`, or `std::unexpect` with its arguments.
- You can also use `operator=/.emplace()/.swap()/std::swap()`.

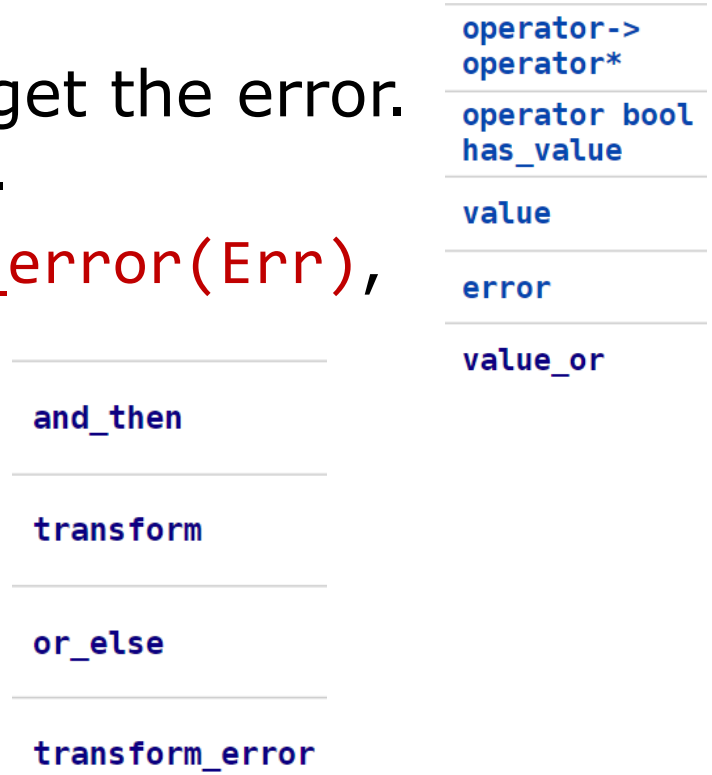
Expected

- For example:

```cpp
std::expected<double, std::errc> parse_number(std::string_view& str)
{
    double result;
    auto begin = str.data();
    auto [end, ec] = std::from_chars(begin, begin + str.size(), result);

    if(ec != std::errc{})
        return std::unexpected{ ec };
    if(std::isinf(result)) // we regard inf as out of range too.
        return std::unexpected{ std::errc::result_out_of_range };

    str.remove_prefix(end - begin);
    return result;
}
```

Expected

- Notice that it only supports `operator==/!=` and doesn't support `std::hash`.
- Other operations: only adds an `.error()` to get the error.
  - And `.value()` may `throw std::bad_expect_access`.
- Monadic operations: only adds a `.transform_error(Err)`, which transform an error **to another error** (Yes, not `E1->T2`, but `E1->E2`).



Monad

- To conclude:
  - `<T1,E1>.and_then(T1)` needs to return `<T2,E1>`;
    - For `std::optional`, it's obligated to return `std::optional<T2>`.
  - `<T1,E1>.transform(T1)` needs to return `T2`, which will construct `<T2,E1>` automatically.
    - For `std::optional`, it's obligated to return `T2`, which will construct `<T2>`.
  - `<T1,E1>.or_else(E1)` needs to return `<T1,E2>`;
    - For `std::optional`, it's obligated to return `std::optional<T1>`.
  - `<T1,E1>.transform_error(E1)` needs to return `E2`, which will construct `<T1, E2>` automatically.

Expected

- For example:

```cpp
auto process = [](std::string_view str)
{
    std::print("str: {:?}, ", str);
    parse_number(str)
        .transform([](double val){
            std::println("value: {}", val);
            return val;
        })
        .transform_error([](std::errc err){
            if(err == std::errc::invalid_argument)
                std::println("error: invalid input");
            else if(err == std::errc::result_out_of_range)
                std::println("error: overflow");
            return err;
        });
};

for (auto src : {"42", "42abc", "meow", "inf"})
    process(src);
```

```
str: "42", value: 42
str: "42abc", value: 42
str: "meow", error: invalid input
str: "inf", error: overflow
```

Pattern Matching*

- Matching in C++ is very weak (`switch` on integers), and thus it's unable to match `expected/optional/variant/`…
  - P1371 or P2688 will solve that.
  - If one of them is adopted, you can use e.g.
  - They have much more utilities, don't cover here since we don't know whether they'll be in C++26 currently (2024/5).

```
inspect (v) {
  <int32_t> i32 =>
    std::print("got int32: {}", i32);
  <int64_t> i64 =>
    std::print("got int64: {}", i64);
  <float> f =>
    std::print("got float: {}", f);
  <double> d =>
    std::print("got double: {}", d);
};
```

```
v match {
  int32_t: let i32 =>
    std::print("got int32: {}", i32);
  int64_t: let i64 =>
    std::print("got int64: {}", i64);
  float: let f =>
    std::print("got float: {}", f);
  double: let d =>
    std::print("got double: {}", d);
};
```

Error Handling

Exception

Error Handling

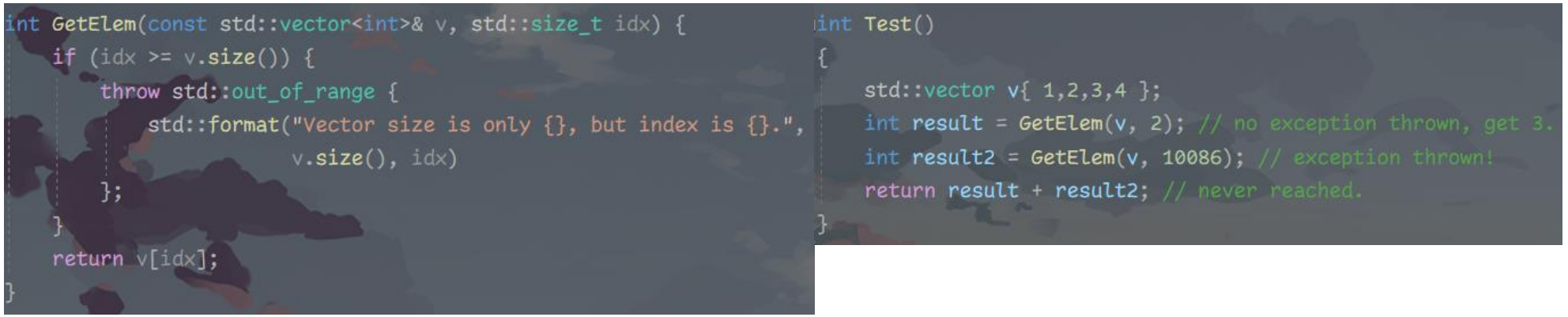
- Exception
  - basics
  - exception safety
  - noexcept

Exception

- Let’s see what `std::expected` has solved:
  - ~~Users often have to pass a pointer to accept the real return value.~~
  - ~~Users have to know different behaviors from the document, e.g. some functions use non-zero as success while some use zero.~~
    - Basically solved, since error type is usually self-explainable, which is more friendly than an integer from nowhere.
  - ~~Sometimes you only need -1 as invalid, which will consume half of unsigned values.~~
  - ~~Error information is usually limited.~~
  - If users cannot process the error, it has to return the error code to the caller again, which needs manual propagation.
    - Still need manual propagation!

Exception

- Exception is a technique that will automatically propagate to the caller if it's omitted.
  - For example, function chain A -> B -> C -> D, if D **throws** an exception, and D doesn't **catch** it, then C needs to do so; if C doesn't, B needs to do so; etc.
    - If main function doesn't process it, then the program will be terminated, which is equivalent to call `std::terminate()`.
  - This propagation process is **stack unwinding**, i.e. stack turns back to the last level over and over again. Dtors will be normally called as if a `return`.



```
int GetElem(const std::vector<int>& v, std::size_t idx) {
    if (idx >= v.size()) {
        throw std::out_of_range {
            std::format("Vector size is only {}, but index is {}.",
                        v.size(), idx)
        };
    }
    return v[idx];
}
```

```
int Test()
{
    std::vector v{ 1,2,3,4 };
    int result = GetElem(v, 2); // no exception thrown, get 3.
    int result2 = GetElem(v, 10086); // exception thrown!
    return result + result2; // never reached.
}
```

Exception

- **try – catch** block to catch an exception:

```
int main()
{
    try {
        Test();
    }
    catch (const std::out_of_range& error) {
        std::cout << "Encountering error: ";
        std::cout << error.what() << "\n";
    }
    return 0;
}
```

If you have multiple exceptions to be caught, you can code multiple catch:

```
try {
    Test();
}
catch (const std::out_of_range& error) {
    std::cout << "Encountering error: ";
    std::cout << error.what() << "\n";
}
catch (const std::invalid_argument& error) {
    // ...
}
catch (...){ /* ... */ }
```

1. `.what()` is a virtual method of `std::exception` (next slides!).
2. `catch(…)` means to catch all exceptions, and you cannot know what the exception really is.

Exception

- Note1: you only need to catch exception when **this method can handle it**.
  - That is to say, after catching the exception, your program is still in a valid state and can run normally; otherwise just terminate your program!
  - For example, `std::bad_alloc` is thrown when memory is not enough; this is possibly not something you can handle so just let it go.
  - But if it's possible for you to get more memory (for example, flush the data back to the disk and free some existing memory), then you may catch it and do so.
  - Thus, it's usually discouraged to use `catch(…)`, since you almost always can only handle some specific exception instead of all.

Exception

- Note2: though you can throw any type, it's recommended to throw a type inherited from `std::exception`.
  - Many general exceptions have been defined in `<stdexcept>` & `<exception>`; you need an error string to construct them.
  - `std::bad_optional_access` / `std::bad_alloc` /… are all inherited from `std::exception`.

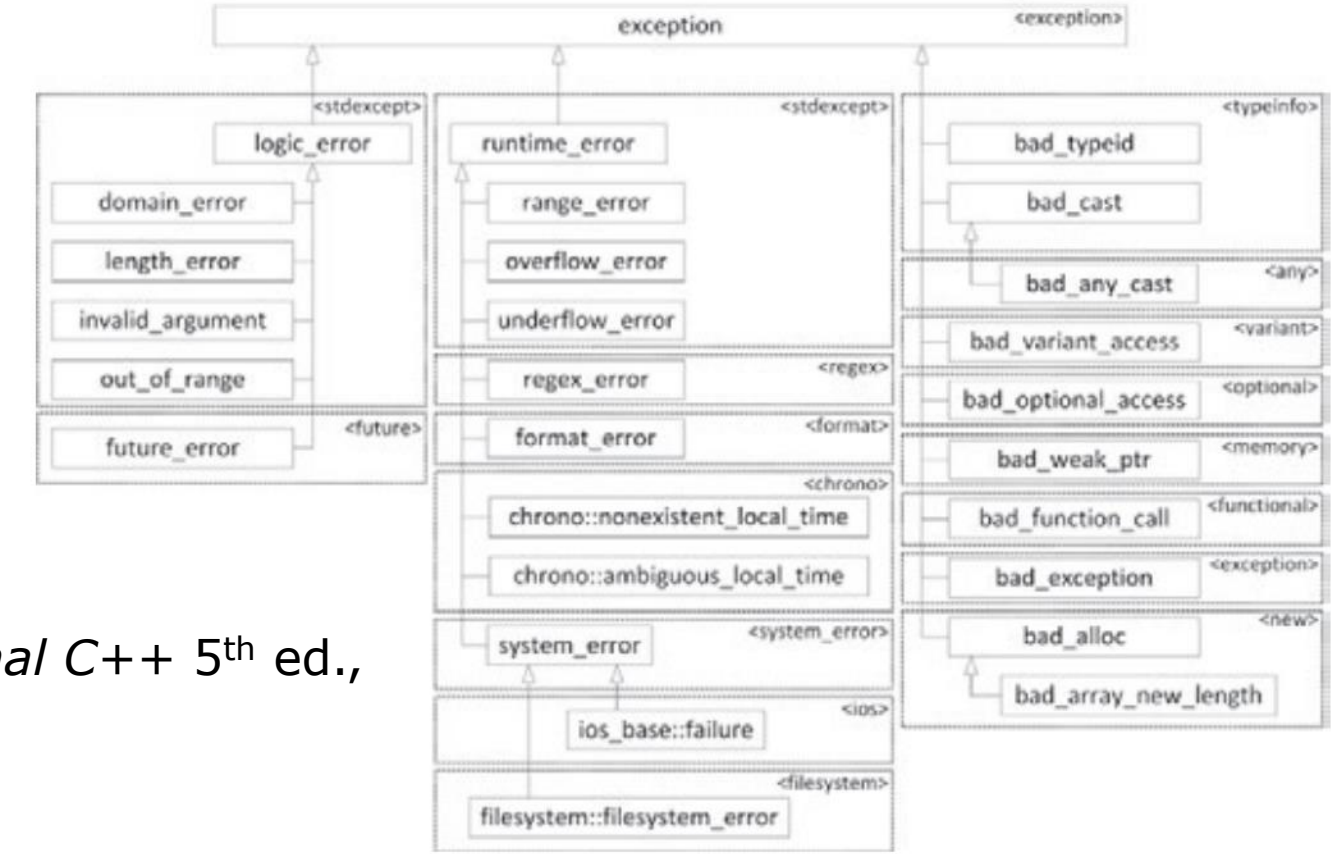
`logic_error`	exception class to indicate violations of logical preconditions or class invariants (class)
`invalid_argument`	exception class to report invalid arguments (class)
`domain_error`	exception class to report domain errors (class)
`length_error`	exception class to report attempts to exceed maximum allowed size (class)
`out_of_range`	exception class to report arguments outside of expected range (class)
`runtime_error`	exception class to indicate conditions only detectable at run time (class)
`range_error`	exception class to report range errors in internal computations (class)
`overflow_error`	exception class to report arithmetic overflows (class)
`underflow_error`	exception class to report arithmetic underflows (class)

Inherited from `logic_error`.

Inherited from `runtime_error`.

Exception

- A total view of standard exceptions (until C++20):



Credit: *Professional C++* 5th ed., Marc Gregoire

Note: Not all exceptions only have a `.what()`; e.g. `std::future_error` will have an additional error code defined in `<system_error>`.

Exception

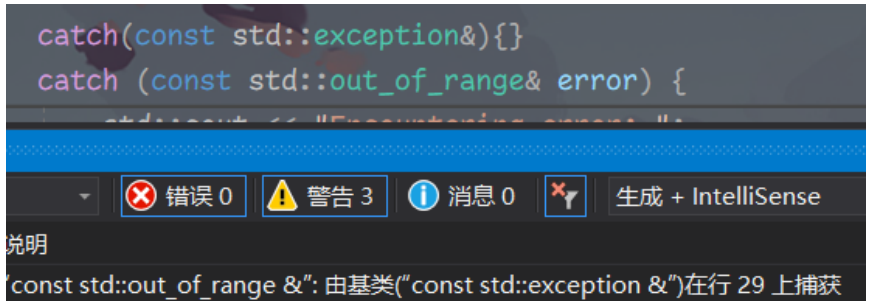
- Reason: base class can also match derived class exception to catch it, so you can always `catch(const std::exception&)` and print `.what()` to know information.
  - The rules to match are a little bit complex, but if you obey Note2, you never need to know the details.

- Customized exception example:
  - You can also override `.what()` if needed.

```
class FormatError : public std::runtime_error
{
public:
    FormatError(std::string formatName) :
        std::runtime_error{ "Wrong " + formatName + " format."}{};
};
```

Exception

- Note3: catch block is matched one by one, so pay attention whether some previous caught block covers later ones.
  - Usually compilers will emit warnings.



- Note4: though it's allowed to catch with or without `const/&`, exception **should definitely** be caught by `const Type&`.
  - For example, if you catch by value instead of reference, then slice problem will happen. That is, if your customized exception has more data, copy to base class will make them disappear; virtual methods are invalid too.

Exception

- Note5: if you're in a `catch` block and find that the caught exception still cannot be handled, you can use a single `throw;` to throw this exception again.
  - This is preferred over `catch(const T& ex) { throw ex;}`, since the latter may lose the type information.

```
void Func()
{
    try { throw std::runtime_error{ "test" }; }
    catch (const std::exception& ex) { throw; }
}
```

```
try { Func(); }
catch (const std::runtime_error& ex) {
    std::cout << "Here1.\n";
}
catch (const std::exception& ex) {
    std::cout << "Here2.\n";
}
```

`throw;` -> Here1.
`throw ex;` -> Here2.

- Note6: if another exception is thrown during internal exception handling (e.g. dtor throws an exception during stack unwinding), `std::terminate` will also be called.

Error Handling

- Exception
  - basics
  - exception safety
  - noexcept

Exception safety

- Exception safety means that when an exception is thrown and caught, program is still in a valid state and can correctly run.
- According to the level of safety guarantee, there are four kinds of exception safety.
- No guarantee: Oops, when an exception is thrown, the program is half-dead; it may:
  - Leak some resources, like memory leak.
  - Destroy invariant assumption of program, like a function will incrementally add a value to 32, but throwing an exception will corrupt it.
  - Corrupt memory, e.g. a memory is written partially and an exception is thrown.
  - When there is no guarantee, you should never make the program continue; just terminate it.

Exception safety

- Basic guarantee: at least program can run normally, no resources leak, invariants are maintained, etc.
- RAII is a really important technique for basic guarantee!
- For example:

```
void science(double* x, int N) {
  double* y = new double[N];
  double* z = new double[N];
  calculate(x, y, z, N);
  delete[] z;
  delete[] y;
}
```

Does it have basic guarantee?

No, because when `calculate` throws, memory leaks happen.

What if `calculate` never throws? Is it safe now?

No, if the second `new` throws `std::bad_alloc`, then memory of `y` leaks.

Credit: CppCon 2022, *Back to Basics: C++ Smart Pointers* by David Olsen

Exception safety

- Another example:

```
std::optional<std::string> GetPetName(int petID)
{
    pmutex.lock();
    auto it = p.find(petID);
    if(it == p.end()){
        pmutex.unlock();
        return std::nullopt;
    }
    auto& pet = it->second;
    auto name = pet.GetFirstName() + " " + pet.GetLastName();
    pmutex.unlock();
    return name;
}
```

- These may be naïve, but real methods are far more complicated and it's impossible for a human to consider all cases.
  - For example, if there are 4 locks and you may return at any point, then you need to unlock 4/3/2/1 locks at different positions.

Exception safety

- Solution: use destructor!
  - No matter when an exception throws, objects that have been constructed will always call their dtors.
- RAII (Resource acquirement is initialization): acquire resources in ctor and release them in dtor.
  - `std::unique_ptr` to manage heap memory instead of `new/delete`.
  - `std::lock_guard` to manage mutex instead of `lock/unlock`.
  - `std::fstream` to manage file instead of `FILE* fopen/fclose`.
  - Your class should also obey this rule!
- Any question?
  - What if ctor throws an exception? Will dtor be called?
  - 还没活，又怎么能死呢？

Exception safety

- To sum up, all members that have been fully constructed will be destructed, but dtor of itself won't be called.
  - Notice that members are constructed in the same order as declaration, not same as member initializer (that is, member initializer will be rearranged).
  - For example: initialization order is `id -> sth -> name`, even if you swap `sth{}` and `name{init_name}`.

```
class Person
{
public:
    Person(const std::string& init_name) :
        sth{}, name{ init_name }
    { std::cout << "I may throw too!\n"; }

private:
    int id = 0;
    SomeClassMayThrow sth;
    std::string name;
};
```

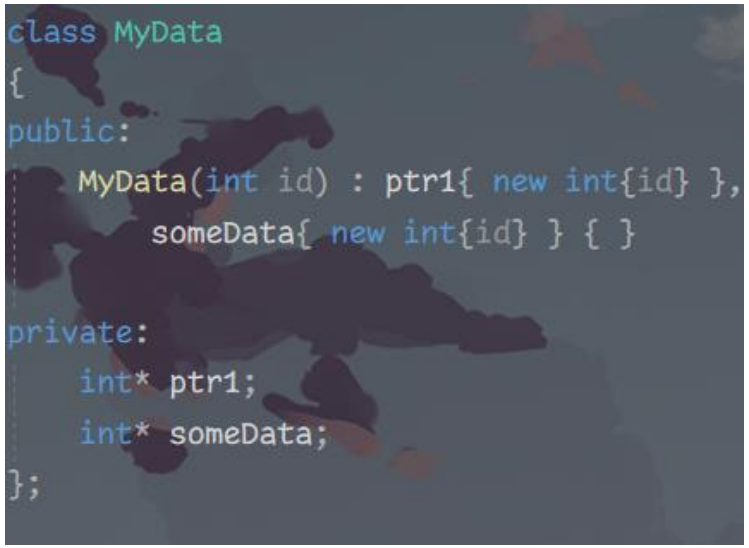
If ctor of `sth` throws, then only `id` will be destructed (which actually does nothing)

If ctor of `name` throws, then `sth` will also be destructed.

If `std::cout <<` throws, then `sth, name` will all be destructed, but dtor of `Person` will still not be called..

Exception safety

- Similarly, if parent class is fully constructed, then dtor of parent class will also be called.
- Not calling dtor of current object may violate exception safety too in a subtle way.
  - For example:



```
class MyData
{
public:
    MyData(int id) : ptr1{ new int{id} },
        someData{ new int{id} } { }

private:
    int* ptr1;
    int* someData;
};
```

Problem? RAII!

```
MyData(int id) : ptr1{ new int{id} },
    someData{ new int{id} } { }

~MyData() { delete ptr1; delete someData; }
```

Now correct?

If `new` of `someData` throws, then `ptr1` will leak since its `delete` will not be called.

Best solution: use `std::unique_ptr<int>` ptr1.

Exception safety

- If you have to own a raw pointer that has ownership to the memory (which is weird), then don’t initialize it in the list.
  - Still utilize RAII!

```
MyData(int id) : ptr1{ nullptr }, someData{ nullptr } {
    auto init_ptr1 = std::unique_ptr<int>{ new int {id} };
    auto init_someData = std::unique_ptr<int>{ new int {id} };
    // will never throw below
    ptr1 = init_ptr1.release(); // release ownership, so dtor of
    someData = init_someData.release(); // unique_ptr does nothing
}

~MyData() { delete ptr1; delete someData; }
```

  - BTW, if `new` fails to allocate memory and throws `std::bad_alloc` (or sometimes `std::bad_array_new_length` for `new[]`, which inherits `std::bad_alloc`), then the allocation won’t happen, so memory won’t leak.
  - BTW2, `new(std::nothrow)` will return `nullptr` instead of throwing exception, e.g. `new(std::nothrow) int{id}`.
    - But you still need to check it (nothrow ≠ noerror)!

Exception safety

- Containers utilize similar techniques by wrapping pointer to base class / member.
  - Remember? `std::vector` has three pointers – first, last, end.
  - Many ctors need to construct new objects to fill in the allocated memory.
    - If one of them throws exception, we need to ensure the memory is released.
  - So `std::vector` wrap these three pointers to a base class, e.g. `std::vector_base`, so that when ctor fails, dtor of parent class will be called and memory is released.
    - Note that the actual implementation is `vector_base -> vector_impl`, and `std::vector` owns a member of type `vector_impl`.
    - This is to compress the size of allocator while make allocator not base of vector; you'll know the details in the future.

- The third level of exception safety is strong exception guarantee.
  - That is, if the function throws an exception, the state of the program is rolled back to the state just before the function call.
  - Most of methods in STL obey strong exception guarantee.
    - E.g. `vector::push_back()`, `vector` has same elements as before even if exception is thrown ({1,2,3} —push 4 but exception thrown→ still {1,2,3}).
  - Remember our question before?
    - However, MS-STL's implementation of insertion doesn't use this way (we'll tell you why in the future); It is:
      - Reallocate if needed(same as normal insertion).
      - push_back all elements one by one.
      - **Rotate** them to the insertion point.
    - MS hopes to provide an exception safety for `.insert()` between basic and strong one when copy ctor / "move ctor" may throw;
      - That is, if exception happens when elements are inserted at back and before rotation, then strong guarantee is applied.
      - This increases time cost by a little; gcc-libstdc++ doesn't do so, which just have basic guarantee with a better performance.

Copy-and-swap idiom

- A technique to maintain strong exception guarantee in assignment operator is copy-and-swap idiom.
- For example, implement a simple vector.
- So how to implement `operator=`?
  - Correct?

```
Vector& operator=(const Vector& another) {
    delete[] first_;
    auto size = another.size();
    std::unique_ptr<T[]> arr{ new T[size] };
    first_ = arr.release();
    std::ranges::copy(another.first_, another.last_, first_);
    last_ = end_ = first_ + size;
    return *this;
}
```

We assume `T` can be default constructed here.

```
template<typename T>
class Vector
{
public:
    Vector(std::size_t num, const T& val){
        std::unique_ptr<T[]> arr{ new T[num] };
        std::ranges::fill(arr.get(), arr.get() + num, val);
        first_ = arr.release();
        last_ = end_ = first_ + num;
    }

    std::size_t size() const noexcept { return last_ - first_; }
    auto& operator[](std::size_t idx) noexcept { return first_[idx]; }
    const auto& operator[](std::size_t idx) const noexcept { return first_[idx];

    Vector(const Vector& another){
        auto size = another.size();
        std::unique_ptr<T[]> arr{ new T[size] };
        std::ranges::copy(another.first_, another.last_, arr.get());
        first_ = arr.release();
        last_ = end_ = first_ + size;
    }
private:
    T* first_, *last_, *end_;
};
```

Copy-and-swap idiom

- What if `new` throws?
  - Though no memory is leaked, invariants have changed! Memory between `first_` and `end_` is released.
    - E.g. `{1,2,3}` is released but `.size()` is still 3.
  - Now correct?

```
Vector& operator=(const Vector& another) {
    delete[] first_;
    first_ = last_ = end_ = nullptr;
    auto size = another.size();
    std::unique_ptr<T[]> arr{ new T[size] };
    first_ = arr.release();
    std::ranges::copy(another.first_, another.last_, first_);
    last_ = end_ = first_ + size;
    return *this;
}
```

  - Still nope! What if `copy(…)` throws (i.e. type `T` throws when copying)?
    - `first_` changes but `last_` and `end_` are still `nullptr`, so `.size()` is garbage.
    - More importantly, we release `arr` too early, memory still leaks…

Copy-and-swap idiom

- Now?

```
Vector& operator=(const Vector& another) {
    delete[] first_;
    first_ = last_ = end_ = nullptr;
    auto size = another.size();
    std::unique_ptr<T[]> arr{ new T[size] };
    std::ranges::copy(another.first_, another.last_, arr.get());
    // only release when all exceptions are possibly thrown!
    first_ = arr.release();
    last_ = end_ = first_ + size;
    return *this;
}
```

- Okay, at least basic guarantee is fulfilled.
  - However, when some exception is thrown, all previous contents are lost, so no strong guarantee.
  - Any solution?
  - Only `delete[]` when all possible exceptions are thrown!

Copy-and-swap idiom

- Finally:

```
Vector& operator=(const Vector& another) {
    auto size = another.size();
    std::unique_ptr<T[]> arr{ new T[size] };
    std::ranges::copy(another.first_, another.last_,
    delete[] first_;
    first_ = arr.release();
    last_ = end_ = first_ + size;
    return *this;
}
```

```
Vector(const Vector& another){
    auto size = another.size();
    std::unique_ptr<T[]> arr{ new T[size] };
    std::ranges::copy(another.first_, another.last_, arr.get());
    first_ = arr.release();
    last_ = end_ = first_ + size;
}
```

- But…isn't it very similar to copy ctor?
  - Solution: copy-and-swap idiom

1. Though defined in class, ADL can ensure it's callable everywhere.
2. `std::swap` and `stdr::swap` are equivalent here, since they're just pointers.
   BUT, `stdr::swap` can call customized `swap` here, while `std::swap` cannot.
3. `noexcept` here is necessary.

```
friend void swap(Vector& vec1, Vector& vec2) noexcept {
    std::ranges::swap(vec1.first_, vec2.first_);
    std::ranges::swap(vec1.last_, vec2.last);
    std::ranges::swap(vec1.end_, vec2.end_);
}

Vector& operator=(const Vector& another) {
    Vector vec{another};
    swap(vec, *this);
    return *this;
}
```

Copy-and-swap idiom

- Pros:
  - Provide strong exception guarantee.
  - Add a `swap()` method, which can be used by users (possibly by indirect `std::ranges::swap`).
  - Increase code reusability, reduce code redundance.
- Cons:
  - Allocating memory before releasing, which increases peak memory.
  - Swap cost is slightly higher than direct assignment.
  - May be not optimal for performance, e.g. if here `first_ ~ end_` has enough memory, then a direct copy is better.
    - Otherwise we need an additional allocation but get a smaller capacity!
    - So standard containers don’t adopt it, just basic guarantee.
- It’s your design to determine whether to use copy-and-swap idiom.

Exception safety

```cpp
class A
{
public:
    A(const std::string& init_str) : str{init_str} {}
private:
    std::string str;
};
```

- One more thing in exception for ctor…
  - It seems that we cannot capture exceptions in initializer.
    - i.e. here `str{ init_str }` throws;
  - C++ provides function-try-block to capture them.

```cpp
A(const std::string& init_str)
    try: str{init_str} { std::cout << "Ctor.\n"; }
    catch(const std::exception& ex) { std::cout << ex.what(); }
```

```cpp
try
    : <ctor-initializer>
{
    /* ... constructor body ... */
}
catch (const exception& e)
{
    /* ... */
}
```

  - Difference with normal `try – catch`: `catch` has to rethrow the current exception or throw a new exception. If the `catch` statement doesn't do this, the runtime automatically rethrows the current exception.
    - Reason: member construction fails so it's still incomplete.
    - In catch block, you shouldn't use any uninitialized member either.

Anyway, just let you know; I never use this feature due to its limitation.

- To sum up, exception safety of containers are:
  - **All** read-only & `.swap()` don't throw at all.
    - This excludes some deliberate exceptions, e.g. `vector.at(index)` when `index >= size` will `throw std::out_of_range`.
  - For `std::vector`, `.push_back/emplace_back()`, or `.insert/emplace/insert_range/append_range()` only one element at back provide strong exception guarantee.
    - For `.insert/emplace/…`, if you guarantee copy / move ctor & assignment / iterator move not to throw, then still strong exception guarantee. (Why?)
    - Similarly, `.shrink_to_fit/reserve/resize()` only require move ctor.
    - Otherwise only basic exception guarantee.
    - Similarly, could you know `.pop_back()` or `.erase()`?
  - For `std::list/forward_list`, all strong exception guarantee.
  - For `std::deque`, it's similar to `std::vector`, adding push at front.
  - For associative containers, `.insert/…` a node / only a single element has strong exception guarantee. (What about erase?)
    - `.rehash()` of unordered ones has strong guarantee too.
    - Otherwise basic exception guarantee.

Error Handling

- Exception
  - basics
  - exception safety
  - noexcept

noexcept

- The strongest exception safety is of course nothrow exception guarantee.
- If a method is considered to never throw an exception, then you can add a `noexcept` specifier.
  - For example:

```
int square(int num) noexcept {
    return num * num;
}
```

  - If your function is labeled as `noexcept` but it throws exception, then `std::terminate` will be called.
  - `noexcept` is also an operator, e.g. `noexcept(v1.push_back(xx))` will be evaluated as `false`, since it may throw for reallocation.
  - `noexcept` will facilitate some optimizations; we'll talk more about `noexcept` in *Move Semantics*.

noexcept

- IMPORTANT: destructor & deallocation is **always** assumed to be `noexcept` by standard library; you **must** obey it.
  - Dtor is the only function by default `noexcept` without any explicit specifier if all dtors of members are `noexcept`.
  - **Compiler-generated** ctor / assignment operators are also `noexcept` if all corresponding ctors / assignment operators of members are `noexcept`.
    - But non-default ones need explicit `noexcept`.
    - Here we mean `=default` or just using in-class member initializers; `A(){}` isn't compiler-generated.
    - You can use `std::is_nothrow_constructible_v<A, Args…>` to test it.
- Since C++17, noexcept function pointer is also supported, e.g. `using Ptr = void(*)(int) noexcept; Ptr ptr = square;`
  - It can convert to `void(*)(int)` implicitly, but the reversed one cannot.

noexcept

- Though `noexcept` may enable some optimizations, you don’t need to try to add it everywhere.
- General rule: for normal methods, only when the operation **obviously** doesn’t throw should you add `noexcept`.
  - For example, merely read-only methods in containers (like `.size()`) are marked as `noexcept`.
    - Even `.pop_back()` doesn’t add it though it never throws…
    - What about `map.find()`? Depend on whether comparison will throw! So it’s still not marked as `noexcept`.
  - Beyond that, `swap` should always be `noexcept` too.
  - There are some more special cases, and we’ll tell you in the future.

When to use exception?

- It seems that exceptions have all advantages, so should we use them everywhere?
  - Some languages like Java/C# are just like this!
  - 那么古尔丹，代价是什么呢？
- Exception is relatively costly compared with other error handling mechanisms, like it relies on heap allocation.
  - So, you need to ensure throwing exception is a rarely-happened case.
  - Take an example of game; when game is running, users don’t like to get stuck; so it’s better not to throw for some wrong input of users.
  - But for initialization, it’s Okay since users will wait anyway, and exception is convenient if some e.g. configuration file doesn’t exist.
  - In a word, don’t let your hot path rely on exception.

SPECIAL NOTICE that **exception doesn’t affect happy path at all**.
See C++中异常的额外开销体现在哪些方面？ - 江东某人的回答 - 知乎.

When to use exception?

- Besides, in current typical implementations, stack unwinding of exception needs a global lock, which is really unfriendly to multi-threading programs.
  - GCC 13 has improved this part thanks to Thomas Neumann (who criticizes exception mechanism but still improves it); it's done by a B-tree and all threads can operate on it in parallel.
  - Clang and MSVC seem not currently.
  - There is a proposal for "deterministic exception" to improve performance, but that's a huge language shift and possibly not accepted in near future.
- Also, exception is highly dependent on platform (just like RTTI); if you hope to catch an exception thrown from a shared library, you need to ensure the toolset is same.

When to use exception?

- To sum up:
  - Pros:
    - Propagate by stack unwinding; only process exception when the method can.
    - Force programmers to pay attention to errors (terminate the program).
  - Cons:
    - Not good for performance-critical sessions; not proper to use in hot path.
    - Not convenient for cross-module try-catch.
    - Many compilers don't optimize it for multi-threading programs.
    - Actually one more: code size may bloat (but this is usually not cared currently).
  - BTW:
    - if you're a programmer on embedding system / operating system, there is no way for "stack unwinding" because it depends on OS, so exception should be always disabled.
      - E.g. in gcc `-fno-exceptions`.
    - And if you are writing library to use in other languages (like C), you also need to catch all exceptions at interface since they're likely unable to handle C++ exception.

We don't cover `std::uncaught_exceptions` and `std::nested_exception` in `<exception>`.
Only check it out when you need.

Error Handling

Assertion

assertion

- Assertion is a technique to **check expected behaviors** of functions or code segments **when debugging**.
  - For example, in Tensorflow code:
  - When the parameter is evaluated to `false`, program will be aborted.
    - `std::abort()` is the default behavior of `std::terminate()`, but the latter can change its behavior by `std::set_terminate_handler(…)`.
  - It's a macro defined in `<cassert>`; it only accepts one parameter, so multiple booleans should be connected with `&&`.
    - Remember? Macros parse by comma directly (e.g. `Func<int,double>(xx)`), so you may sometimes add an additional pair of paratheses, e.g. `assert((Func<int,double>(xx) && "Error: xxx"))`.

```
617       // Add the attributes to the function arguments.
618       assert(arg_attrs.size() == arg_types.size());
619       assert(result_attrs.size() == result_types.size());
620       result.attributes.append(builder.getNamedAttr(
621           getArgAttrsAttrName(result.name), builder.getArrayAttr(arg_attrs)));
622       result.attributes.append(builder.getNamedAttr(
623           getResAttrsAttrName(result.name), builder.getArrayAttr(result_attrs)));
```

assertion

- Sometimes you may also see `assert(false)`; this means current code path should never be reached.
- Particularly, this check is only done when macro `NDEBUG` is not defined (like Debug mode in VS); otherwise this macro **does nothing** (equivalent to `(void)0`).
  - For example, you shouldn't `assert(SomeImportantWork(xx))` though `SomeImportantWork` returns `bool`; when this macro is enabled, this function won't be executed.
    - You should only put condition checking that has no side effects no matter whether it's done.
- `assert` is done **in runtime**; if you want to determine in compile time, you can use **keyword** `static_assert(xx, msg)`.
  - E.g. `static_assert(sizeof(int) == 4, "SomeInfo")`; compile error if violated.
  - `msg` can be omitted since C++17.

assertion

- When to use assertion?
  - For any possible input, you shouldn’t use assertion to check; but instead by `std::expected` / exception to report to the caller.
  - For some inner methods (usually users cannot call it), some illegal input **should have been filtered**, then you may use assertion in case you casually violates assumption of these methods.
    - `assert` helps you to find it in debug mode!
  - Remember `[[assume(xx)]]` we have said? You may add an assertion before it to ensure that.
    - Notice that C++23 adds a `std::unreachable()` in `<utility>`, which has same functionality as `[[assume(false)]]`.

Contract*

```
int f(const int x)
  pre (x != 1)        // a precondition assertion
  post(r : r != 2)    // a postcondition assertion; r refers
{
  contract_assert (x != 3);   // an assertion statement
  return x;
}
```

- Assertion is really limited for safety checking.
- C++26 is likely to add *contracts*, which will enrich it a lot.
  - P2900; in fact contract is discussed since 2006 in C++ committee.
  - Three parts:
    - Pre-condition, Post-condition, Assertion
    - Before entering the function, pre-condition will be checked; after leaving the function, post-condition will be checked.
  - Users can use e.g. compiler options to control the behavior.
    - Ignore – not check for zero overhead;
    - Observe – call handler for violation, but continue to run. Compile-time violation will issue a warning.
    - Enforce – call handler for violation and then abort. Compile-time violation will issue compiler error.

一些提案甚至建议放弃对静态分析的支持。类似这样的提案有几十个变种，全都来得太晚，没一个能增进共识。

大量涌入的新奇提案（来自 Bloomberg 团队和其他团队，比如，[Berne 2019; Berne and Lakos 2019; Khlebnikov and Lakos 2019; Lakos 2019; Rosen et al. 2019]）和成百上千讨论这些提案的电子邮件阻碍了真正必需的讨论，即对工作文件中的设计现状进行问题修复。正如我曾不断警告的那样（比如 [Stroustrup 2019c]），这些企图重新设计契约的提案的结果是，在 Nico Josuttis 的提议下，契约被从 C++20 中移除 [Josuttis et al. 2019b]。我认为去年关于契约的讨论是一个典型的例子，谁都得不到任何东西，因为有人只想要按他们的方式来。新的研究组 SG21 能否为 C++23 或 C++26 交付某种能够被更广泛接受的东西，时间将会给出答案。

Error Handling

Debug helpers

Error Handling

- Debug helpers
  - source_location
  - stacktrace
  - debugging

source_location

- Usually, we may want to log error info with its location in source code, so we can quickly know where goes wrong.
- Before C++20, we may use macros __FILE__ and __LINE__, which will be substituted to source file name and line number.

```
static void CheckError(cudaError_t error, const char* file, int line)
{
    if (error == cudaSuccess)
        return;
    std::cerr << "Cuda Error at file " << file << ", line : " <<
        line << " :" << cudaGetErrorString(error);
    exit(EXIT_FAILURE);
}

#define CHECK_ERROR(error) (CheckError((error), __FILE__, __LINE__))
```

source_location

- Since C++20, `<source_location>` is added to solve it.
  - You can use default value directly; very convenient!

```
void LogError(std::string_view errorInfo, std::ostream& logFile = std::cerr,
    const std::source_location& location = std::source_location::current());
```

  - A source location has four methods:

Field access

`line`	return the line number represented by this object (public member function)
`column`	return the column number represented by this object (public member function)
`file_name`	return the file name represented by this object (public member function)
`function_name`	return the name of the function represented by this object, if any (public member function)

source_location

- For example:

```
void LogError(std::string_view errorInfo, std::ostream& logFile = std::cerr,
    const std::source_location& location = std::source_location::current())
{
    logFile << std::format("In file {} - function {} - line {}, Error :\n{}\n",
        location.file_name(), location.function_name(), location.line(), errorInfo);
    return;
}


int main()
{
    LogError("TestInfo");
    return 0;
}
```

```
In file /app/example.cpp - function int main() - line 17, Error :
TestInfo
```

Error Handling

- Debug helpers
  - source_location
  - stacktrace
  - debugging

stacktrace

- Since C++23, in `<stacktrace>`.
- Similar to `source_location`, you need `std::stacktrace::current` to get it; but you may print it directly!

```cpp
void LogError(std::string_view errorInfo, std::ostream& logFile = std::cerr,
    const std::source_location& location = std::source_location::current(),
    const std::stacktrace& trace = std::stacktrace::current())
{
    logFile << std::format("In file {} - function {} - line {}, Error :\n{}\n",
        location.file_name(), location.function_name(), location.line(), errorInfo);
    logFile << std::format("Current stack:\n{}\n", trace);
    return;
}

void Func(){ LogError("TestInfo"); }

int main()
{
    Func();
    return 0;
}
```

```
In file /app/example.cpp - function void Func() - line 17, Error :
TestInfo
Current stack:
   0# Func() at /app/example.cpp:17
   1# main at /app/example.cpp:21
   2#       at :0
   3# __libc_start_main at :0
   4# _start at :0
   5#
```

Full param: `current(skip = 0, max_depth = total_size(), allocator = default)` to preserve only items in `[skip, max_depth)`.

Note: some functions may be optimized out, thus there is no name or even no entry in `-O3`.

stacktrace

- `stacktrace` is allocated dynamically, so it can also designate an allocator (`std::basic_stacktrace<alloc>`; `stacktrace` uses the default allocator).
- It's a limited sequential container, with elements as `std::stacktrace_entry`;
  - But possibly it's not useful as a container, so see here for its methods if you're interested.
  - `stacktrace_entry` has `description()`, `source_line()` and `source_file()` to check information.
  - `stacktrace_entry` and `stacktrace` support hash, comparison and `std::to_string()` too.

Error Handling

- Debug helpers
  - source_location
  - stacktrace
  - debugging

debugging*

- Since C++26, in `<debugging>`.

Debugging support

Defined in header <debugging>

breakpoint (C++26)	pauses the running program when called (function)
breakpoint_if_debugging (C++26)	calls `std::breakpoint` if `std::is_debugger_present` returns `true` (function)
is_debugger_present (C++26)	checks whether a program is running under the control of a debugger (function)

- `breakpoint()` will set a hidden breakpoint; Debugger will identify it and stop there.
  - This is very useful when the condition is dramatically complex, e.g. `CheckThis() % 2 == 0 && CheckThat() && …`.
    - Debugger can pause on condition, but complex ones will be troublesome and writing them into code will be better.

Error Handling

Unit test

Unit test

- Test is one of the most important techniques in programming.
  - As complexity increases, it’s hard to ensure every part of code is correct; instead, we use tests to verify it.
  - TDD – Test-Driven Development.
  - Normally, pair programming is preferred; that is, one programmer implements the feature while the other implements tests and review code.
    - As students, we are possibly not able to do pair programming; however, you may utilize AI coder to help you.

About GPT*

- My attitude towards GPT or AI coder (like Copilot):
  - When learning new knowledge, like doing homework of this course, it's really **discouraged** to use AI to help you.
    - Stephen Krashan & Noam Chomsky: "Use of ChatGPT is a way of avoid learning".
  - If you're already adept in coding C++, then that's fine.
  - Personally I don't use AI coder frequently because it's kind of interruption to my thoughts; but I'd like to generate tests by it.
    - GPT is also misleading or even totally wrong, especially for newest C++ standard.

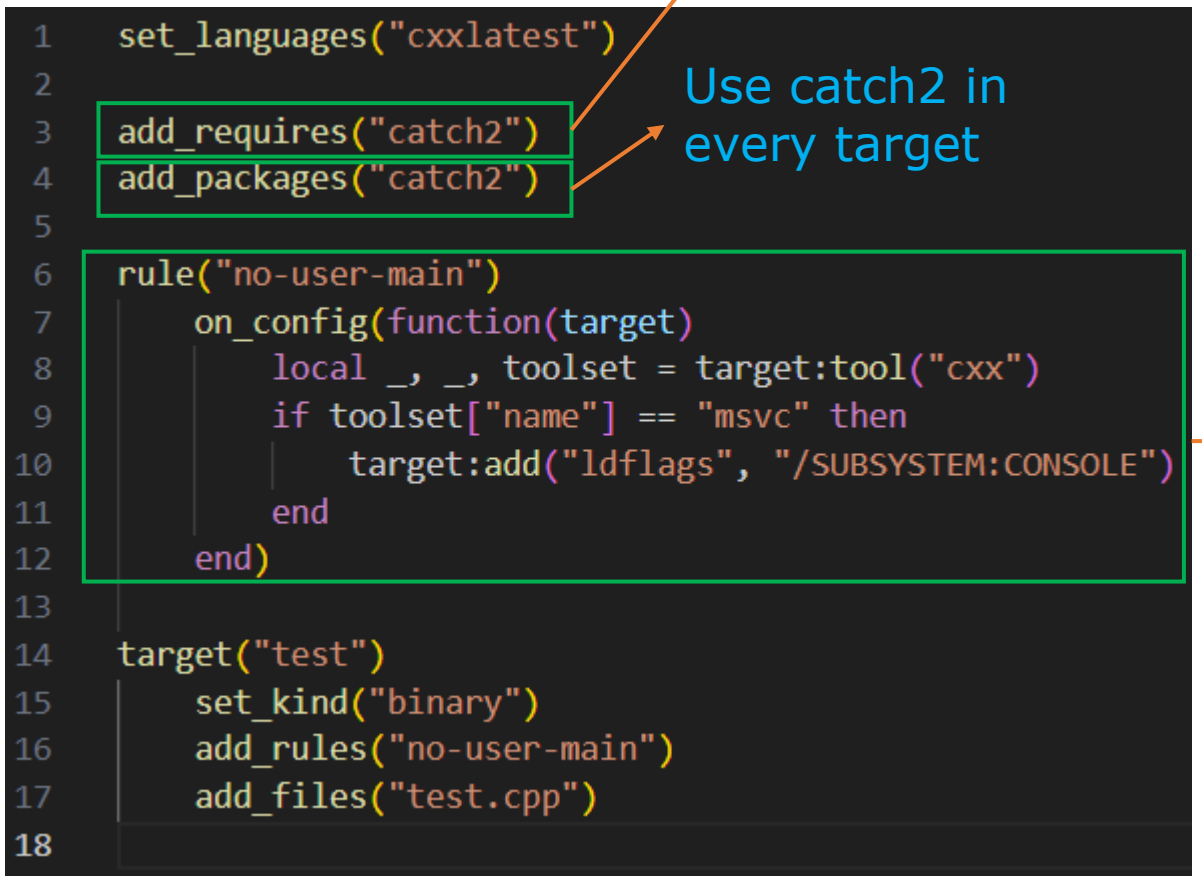
Unit test

- There are plenty kinds of tests;
  - Normally, we should code a unit test for each unit (usually every class / every source file).
  - After that, we need integration tests to combine units.
  - Finally, we may do system tests for the whole system features.

- We'll introduce a C++ unit test framework – **Catch2** in this course.
  - It's the second most popular framework currently.
  - There are plenty of similar frameworks, like Google Test, Boost Test, Visual C++ Testing Framework, etc. Learn them yourself when you need.

Catch2

- `xrepo install Catch2`.
- xmake.lua:




```lua
set_languages("cxxlatest")

add_requires("catch2")
add_packages("catch2")

rule("no-user-main")
    on_config(function(target)
        local _, _, toolset = target:tool("cxx")
        if toolset["name"] == "msvc" then
            target:add("ldflags", "/SUBSYSTEM:CONSOLE")
        end
    end)

target("test")
    set_kind("binary")
    add_rules("no-user-main")
    add_files("test.cpp")
```

Catch2

- test.cpp:

```cpp
#include <catch2/catch_test_macros.hpp>

TEST_CASE("Test Name")
{
    REQUIRE(1 + 2 == 3);
}
```

```
D:\111\University\Course\CS\程序设计实习与C++\PPT\Modern C++\Error Handling\code\Catch2>xmake build
[ 25%]: compiling.debug test.cpp
[ 50%]: linking.debug test.exe
[100%]: build ok, spent 1.953s

D:\111\University\Course\CS\程序设计实习与C++\PPT\Modern C++\Error Handling\code\Catch2>xmake run test
Randomness seeded to: 3256368759
===============================================================================
All tests passed (1 assertion in 1 test case)
```

- If changed to `1 + 2 != 3`:

```
-------------------------
Test Name
-------------------------
test.cpp(3)
.........................

test.cpp(5): FAILED:
  REQUIRE( 1 + 2 != 3 )
with expansion:
  3 != 3

=========================
test cases: 1 | 1 failed
assertions: 1 | 1 failed
```

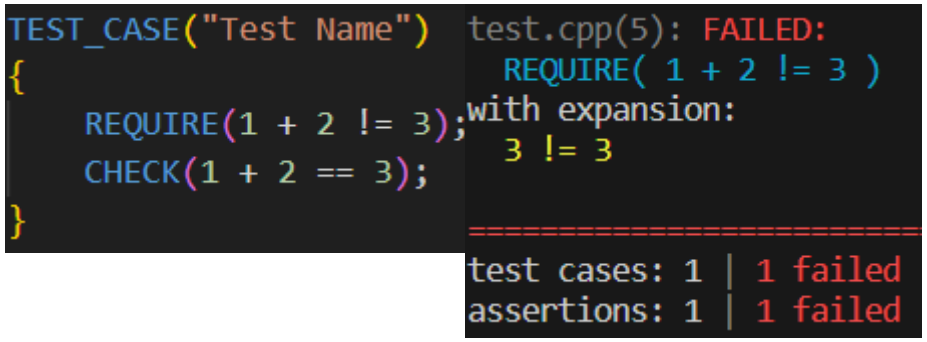
Catch2

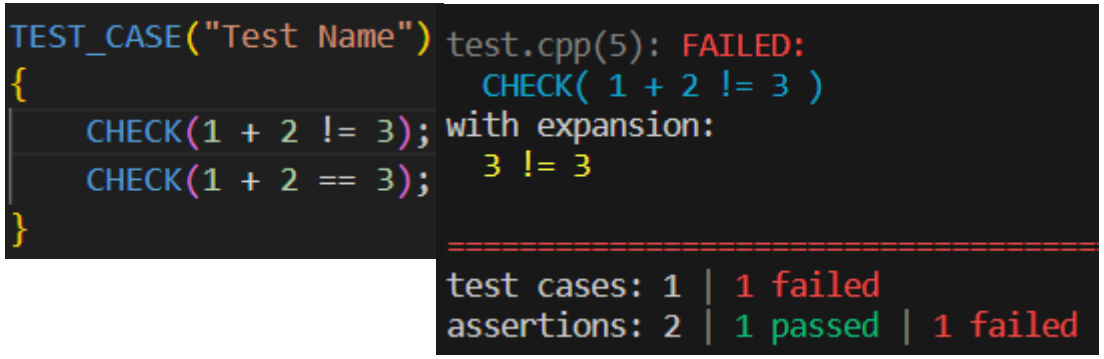
- You can also use it with command line: `xmake run test "Test Name"`
  - Which means run test “Test Name”.

```
D:\111\University\Course\CS\程序设计实习与C++\PPT\Modern C++\Error Handling\code\Catch2>xmake run test -h

Catch2 v3.4.0
usage:
  test.exe [<test name|pattern|tags> ... ] options

where options are:
  -?, -h, --help                          display usage information
  -s, --success                           include successful tests in
                                          output
  -b, --break                             break into debugger on failure
  -e, --nothrow                           skip exception tests
  -i, --invisibles                        show invisibles (tabs, newlines)
  -o, --out <filename>                    default output filename
  -r, --reporter <name[::key=value]*>     reporter to use (defaults to
                                          console)
  -n, --name <name>                       suite name
  -a, --abort                             abort at first failure
  -x, --abortx <no. failures>             abort after x failures
```

Catch2

- Now we introduce some details.
  - `REQUIRE / CHECK`: the former will stop the following tests in the current region while the latter won't.
    - `REQUIRE` should be used when previous things are preconditions of things later.





  - There are also `REQUIRE_FALSE / CHECK_FALSE` to assert `false`.
  - You can also check some expressions never throw by `REQUIRE_NOTHROW / CHECK_NOTHROW`.

Catch2

- Or even check throwing type by `REQUIRE_THROW_AS / CHECK_THROW_AS(exp, ExceptionType)`.

```
auto NoFile = [] {
    Lexer lexer{ TEST_FILE_DIR "/no-file.txt", nullFile };
};
REQUIRE_THROWS_AS(NoFile(), std::runtime_error);

auto EmptyFile = [] {
    Lexer lexer{ TEST_FILE_DIR "/input/empty.txt", nullFile };
};
REQUIRE_THROWS_AS(EmptyFile(), std::ios::failure);
```

- `TEST_CASE` can also use two arguments `(name, tag)`.
  - Tag can be used to group different `TEST_CASE` together.
  - `[]` is necessary.

```
TEST_CASE("Test1", "[math]")
{ // 勾股定理
    REQUIRE("a^2 + b^2 = c^2");
}

TEST_CASE("Test2", "[physics]")
{ // Einstein mass-energy equivalence
    REQUIRE("E = MC^2");
}

TEST_CASE("Test3", "[physics]")
{ // Schwarzschild radius
    REQUIRE("R = 2MG/(v^2)");
}
```

```
D:\111\University\Course\CS\程序设计实习与C++\PPT\Modern C++\Error Handling\code\Catch2>xmake run test [physics]
Filters: [physics]
Randomness seeded to: 392415022
===============================================================================
All tests passed (2 assertions in 2 test cases)
```

Catch2

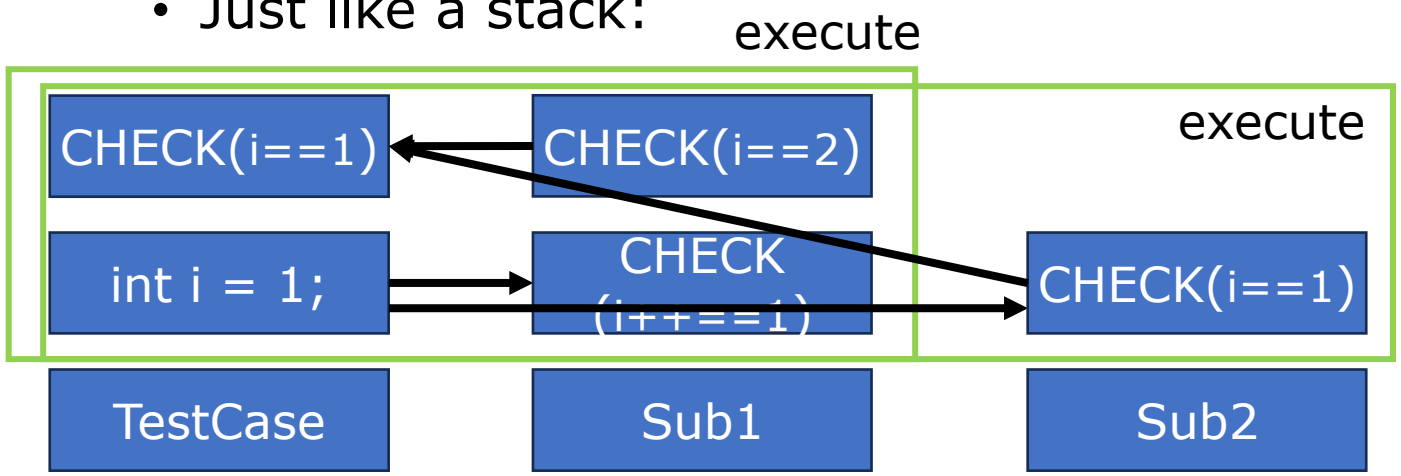
```
TEST_CASE("Test4", "[physics][math]")
{
    CHECK(false);
}
```

```
Filters: [physics]
Randomness seeded to: 2337118510

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
test.exe is a Catch2 v3.4.0 host appl
Run with -? for options

-------------------------------------
Test4
test.cpp(30): FAILED:
  CHECK( false )

=====================================
=================
test cases: 3 | 2 passed | 1 failed
assertions: 3 | 2 passed | 1 failed
```

- You can also attach multiple tags to a test case, e.g.
  - `[tag1],[tag2]` to filter by or, `[tag1][tag2]` to filter by and.
    - Here `[physics],[math]` will test all cases, and `[physics][math]` will only test “Test4”.
- Note: tags should begin with letters (i.e. a-z A-Z).
  - Otherwise it’s preserved for special filters.
    - `[.xxx]`：默认不运行，除非指定`xxx`标签。
    - `[!throws]`：即使测试成功，这个test也可能抛出异常。
    - `[!mayfail]`：即使断言失败，仍然继续运行。
    - `[!shouldfail]`：如果测试成功，则报告测试失败。
    - `[!nonportable]`：测试仅保证此平台的正确性。

Catch2

- Sometimes, a test case may be divided into multiple sections so that they may share same initializations.
  - Statements outside will be executed again.
  - Just like a stack:




```
TEST_CASE("Test5")
{
    int i = 1;
    SECTION("Sub1", "This is description")
    {
        CHECK(i++ == 1);
        CHECK(i == 2);
    }

    SECTION("Sub2", "Another description")
    {
        CHECK(i == 1);
    }
    CHECK(i == 1);
}
```

```
test.cpp(50): FAILED:
  CHECK( i == 1 )
with expansion:
  2 == 1

===============================================
test cases: 1 | 1 failed
assertions: 5 | 4 passed | 1 failed
```

  - `SECTION` can also be nested; the principle is still like stack.

Catch2

- Note1: Catch2 uses macro, so still remember to add paratheses for comma.
- Note2: Catch2 overloads `operator&&` and `operator||`, so using them **won't be short-circuit**.
  - Thus, you may need to split it to multiple `REQUIRE`.
- Note3: `SECTION` can be generated dynamically; you just need to ensure they have different names. For example:

```
TEST_CASE( "looped SECTION tests" ) {
    int a = 1;

    for( int b = 0; b < 10; ++b ) {
        SECTION("b is currently: " + std::to_string(b)) {
            CHECK( b > a );
        }
    }
}
```

Catch2

- You can also use log macros to print message.
  - When test fails, it will print all log until current scope.
  - 4 levels: `INFO(“xx”)/WARN(“xx”)/FAIL_CHECK(“xx”)/FAIL(“xx”)`;
    - INFO/WARN: just log some information.
    - FAIL_CHECK: log error and see this as a failed test (like `CHECK(false)`).
    - FAIL: same as FAIL_CHECK, and also terminate current region (like `REQUIRE(false)`)..
  - There are also two special macros:
    - `SUCCESS(“xx”)`: log success and see current as a success test.
    - `SKIP(“xx”)`: log skip the current scope.
  - These macros are essentially streams, so you can also `INFO(1 << “wow”)`.

```
test.cpp(52): FAILED:
  CHECK( 1 == 0 )
with message:
  Test!

test.cpp(56): FAILED:
  CHECK( 1 == 2 )
with messages:
  Test!
  Test2!

test.cpp(58): FAILED:
  CHECK( 1 == 3 )
with message:
  Test!
```

```
TEST_CASE("Test6")
{
    INFO("Test!");
    CHECK(1 == 0);

    {
        INFO("Test2!");
        CHECK(1 == 2);
    }
    CHECK(1 == 3);
}
```

Catch2

- You can also do a bunch of tests with different input:
  - Template test for different template parameter, defined in `<catch_template_test_macros.hpp>`.

```
TEMPLATE_TEST_CASE("TemplateTest", "[vector][template]",
                   int, std::string, (std::pair<int, float>))
{
    std::vector<TestType> vec;
    CHECK(std::is_arithmetic_v<TestType>);
}
```

    - You need to specify name by “Name - Type” (space is important), like “TemplateTest - int”; so tag is important here.

```
D:\111\University\Course\CS\程序设计实习与C++\PPT\Modern C++\Error Handling\code\Catch2>xmake run test "TemplateTest - int"
Filters: "TemplateTest - int"
Randomness seeded to: 774412008
===============================================================================
All tests passed (1 assertion in 1 test case)
```

Catch2

- Or generate lots of data:
  - Generator defined in <generators/catch_generators_all.hpp>
  - Generator can be seen as a virtual SECTION, ending until current scope ends; in each stack of execution flow, the generator is initialized as a single value.

```
TEST_CASE("GeneratorTest")
{
    auto i = GENERATE(1, 2, 3);
    SECTION("sub1")
    {
        auto j = GENERATE(2, 3, 4);
        CHECK(i * j <= 10);
    }

    CHECK(i != 0);
}
```

```
test.cpp(78): FAILED:
  CHECK( i * j <= 10 )
with expansion:
  12 <= 10

===============================================================================
test cases:  1 |  0 passed | 1 failed
assertions: 18 | 17 passed | 1 failed
```

SECTION("sub1") is same as:

```
SECTION("sub1")
{
    SECTION("unnamed1")
    {
        auto j = 2;
        CHECK(i * j <= 10);
    }

    SECTION("unnamed2")
    {
        auto j = 3;
        CHECK(i * j <= 10);
    }

    SECTION("unnamed3")
    {
        auto j = 4;
        CHECK(i * j <= 10);
    }
}
```

Catch2

- Generator can also be constructed from a container or iterator pair, by `Catch::Generators::from_range(…)`.
  - You cannot pass a C++20 range; use `std::ranges::to` to convert them.
  - Catch2 provides ranges-like generators, but since C++ has already provide it in standard library, we don't bother to talk about it here.
- Notice: you can also use `GENERATE_REF` if you want to use variables in the current scope.

```
TEST_CASE("GeneratorTest")
{
    auto i = GENERATE(1, 2, 3);
    SECTION("sub1")
    {
        auto j = GENERATE((Catch::Generators::from_range(
            std::views::iota(2, 5)
            | std::views::filter([](int i){ return i % 2 == 0; })
            | std::ranges::to<std::vector>())
        ));
        CHECK(i * j <= 10);
    }

    CHECK(i != 0);
}
```

Catch2

- Finally, you can also do benchmark:
  - Benchmarks often need warming up to make code segment as if in a real environment (otherwise cache cold miss etc. may influence performance).
  - You may also need to repeat benchmark to get an average result.
  - Benchmark in Catch2 will do these things automatically!

```
TEST_CASE("BenchmarkTest")
{
    BENCHMARK("Benchmark Name")
    {
        return Fib(3);
    }; // this ';' is necessary.

    BENCHMARK("Benchmark Name2")
    {
        auto result = Fib(7);
        return result;
    };
}
```

```
unsigned int Fib(unsigned int i){
    if(i <= 1)
        return 1;
    return Fib(i - 1) + Fib(i - 2);
}
```

<benchmark/catch_benchmark.hpp>

```
benchmark name                       samples       iterations    estimated
                                     mean          low mean      high mean
                                     std dev       low std dev   high std dev
-------------------------------------------------------------------------------
Benchmark Name                             100          3395     3.7345 ms
                                     13.5426 ns    13.2869 ns    14.1841 ns
                                     1.94134 ns    0.928491 ns   4.08312 ns

Benchmark Name2                            100           576     4.032 ms
                                     68.4392 ns    68.342 ns     68.6128 ns
                                     0.642246 ns   0.39291 ns    0.956696 ns
```

Final word

- Note1: if you really want a `main` function to do other things, you can use `Catch::Session` in `<catch_session.hpp>`.

Or if you want to provide command line arguments:

```cpp
int main()
{
    return Catch::Session{}.run();
}
```

```cpp
int main(int argc, char* argv[])
{
    return Catch::Session{}.run(argc, argv);
}
```

- Note2: These headers can be included by a single header `<catch_all.hpp>`.
- Note3: Catch2 also provides a random seed; you can get it by `Catch::getSeed()`.

Summary

- Error code extension
  - std::optional / std::expected
  - Monad operations
- Exception
  - Stack unwinding
  - try-catch block
  - std::exception
  - **Exception safety**
    - Copy-and-swap idiom
  - noexcept
    - Dtor should **never** throw.
  - When to use exception – make it rarely happen.
- Assertion
  - assert and static_assert
- Debug helpers
  - source_location and stacktrace
- Unit Test
  - Catch2

Next lecture…

- In the next lecture, we'll cover string and stream.
- This includes std::string and string view, and Unicode support in C++
- More importantly, std::format and std::print!
- A deep dive into stream…
- And finally regular expression!