Exception Handling and Exception Safety

CS100 Lecture 24

GKxx

December 26, 2023

Contents

Input failure

```
int num_of_people;
std::cin >> num_of_people;
```

What happens when the input is not an integer?

Input failure

```cpp
int num_of_people;
std::cin >> num_of_people;
```

What happens when the input is not an integer?

```cpp
if (!std::cin) {
  // handle input failure
}
```

strcpy

You are asked to write a `strcpy` function...

```
void strcpy(char *dest, const char *source) {
  while (*source)
    *dest++ = *source++;
  *dest = '\0';
}
```

strcpy

You are asked to write a `strcpy` function...

```
void strcpy(char *dest, const char *source) {
  while (*source)
    *dest++ = *source++;
  *dest = '\0';
}
```

In reality, things may go wrong:

- Null pointers? Or even worse - wild pointers?
- Buffer overflow?

Which is better?

1. Terminate the program on failure and report the error.

```
void strcpy(char *dest, const char *source) {
  if (!dest || !source) {
    std::cerr << "strcpy arguments invalid.\n";
    exit(1);
  }
  while (*source)
    *dest++ = *source++;
  *dest = '\0';
}
```

2. Return false on failure:

```
bool strcpy(char *dest, const char *source) {
  if (!dest || !source)
    return false;
  while (*source)
    *dest++ = *source++;
  *dest = '\0';
  return true;
}
```

Which is better?

3. Be silent to errors.

```
void strcpy(char *dest, const char *source) {
  if (dest && source) {
    while (*source)
      *dest++ = *source++;
    *dest = '\0';
  }
}
```

4. Use assertions.

```
void strcpy(char *dest, const char *source) {
  assert(dest != NULL);
  assert(source != NULL);
  while (*source)
    *dest++ = *source++;
  *dest = '\0';
}
```

A good blog on this topic: `https://blog.csdn.net/myan/article/details/1921`

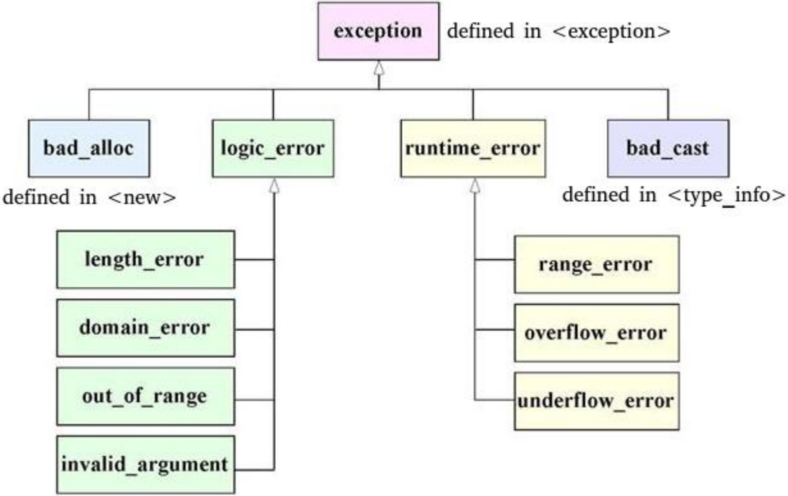
Contents

Throwing an exception

```cpp
class Dynarray {
  std::size_t m_length;
  int *m_storage;

public:
  int &at(std::size_t n) {
    if (n >= m_length)
      throw std::out_of_range{"Dynarray subscript out of range!"};
    return m_storage[n];
  }
};
```

Standard exceptions



Standard exceptions

- The normal new and new[] operators throw std::bad_alloc when running out of memory.
- dynamic_cast for references throws std::bad_cast when the cast fails.
    - dynamic_cast for pointers does not throw. It returns nullptr on failure.

Standard exceptions

- The normal `new` and `new[]` operators throw `std::bad_alloc` when running out of memory.
- `dynamic_cast` for references throws `std::bad_cast` when the cast fails.
  - `dynamic_cast` for pointers does not throw. It returns `nullptr` on failure.
- `std::system_error` is thrown in many cases, especially in functions that interface with OS facilities, e.g. the constructor of `std::thread`.
- `<chrono>` defines `std::nonexistent_local_time` and `std::ambiguous_local_time` representing some errors related to time settings.

Standard exceptions

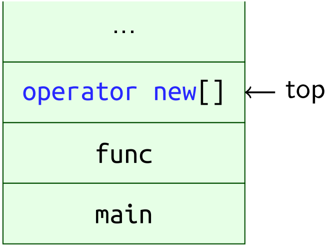
`operator[]` for STL containers does not check boundaries, but `at()` does.

```
std::vector<int> v;
v.at(0) = 42; // Throws std::out_of_range.
v[0] = 42;    // Does not throw, but undefined behavior
              // (and often severe runtime error).
```

We will see that exceptions `throw`n could be `catch`ed and handled.

Stack unwinding

```cpp
void func(int n) {
  std::string s;
  std::cin >> s;
  int *p = new int[n];
  // ...
}
int main() {
  int size = 100;
  func(size);
  // ...
}
```



Suppose `operator new[]` encounters shortage of memory...

Stack unwinding

```
void func(int n) {
  std::string s;
  std::cin >> s;
  int *p = new int[n];
  // ...
}
int main() {
  int size = 100;
  func(size);
  // ...
}
```

1. During the creation of p, std::bad_alloc is raised in operator new[].

Stack unwinding

```
void func(int n) {
  std::string s;
  std::cin >> s;
  int *p = new int[n];
  // ...
}
int main() {
  int size = 100;
  func(size);
  // ...
}
```

1. During the creation of `p`, `std::bad_alloc` is raised in `operator new[]`.
2. Control flow returns to `func`.

Stack unwinding

```
void func(int n) {
  std::string s;
  std::cin >> s;
  int *p = new int[n];
  // ...
}
int main() {
  int size = 100;
  func(size);
  // ...
}
```

1. During the creation of `p`, `std::bad_alloc` is raised in `operator new[]`.
2. Control flow returns to `func`.
3. `s` is destroyed.

Stack unwinding

```
void func(int n) {
  std::string s;
  std::cin >> s;
  int *p = new int[n];
  // ...
}
int main() {
  int size = 100;
  func(size);
  // ...
}
```

1. During the creation of p, std::bad_alloc is raised in operator new[].
2. Control flow returns to func.
3. s is destroyed.
4. n is destroyed.

Stack unwinding

```
void func(int n) {
  std::string s;
  std::cin >> s;
  int *p = new int[n];
  // ...
}
int main() {
  int size = 100;
  func(size);
  // ...
}
```

1. During the creation of `p`, `std::bad_alloc` is raised in `operator new[]`.
2. Control flow returns to `func`.
3. `s` is destroyed.
4. `n` is destroyed.
5. Control flow returns to `main`.

Stack unwinding

```cpp
void func(int n) {
  std::string s;
  std::cin >> s;
  int *p = new int[n];
  // ...
}
int main() {
  int size = 100;
  func(size);
  // ...
}
```

1. During the creation of `p`, `std::bad_alloc` is raised in `operator new[]`.
2. Control flow returns to `func`.
3. `s` is destroyed.
4. `n` is destroyed.
5. Control flow returns to `main`.
6. `size` is destroyed.

Stack unwinding

```
void func(int n) {
  std::string s;
  std::cin >> s;
  int *p = new int[n];
  // ...
}
int main() {
  int size = 100;
  func(size);
  // ...
}
```

1. During the creation of `p`, `std::bad_alloc` is raised in `operator new[]`.
2. Control flow returns to `func`.
3. `s` is destroyed.
4. `n` is destroyed.
5. Control flow returns to `main`.
6. `size` is destroyed.

Notice

Stack unwinding is only guaranteed to happen for **caught** exceptions. If an exception is not caught, whether the stack is unwound is **implementation-defined**.

Contents

Catch an exception

```cpp
void func(int n) {
  std::string s;
  std::cin >> s;
  int *p = new int[n];
  // ...
}
int main() {
  try {
    int size = 100;
    func(size);
  } catch (const std::bad_alloc &e) {
    // deal with shortage of memory here.
  }
  // ...
}
```

More Effective C++ Item 13: Catch exceptions by reference.

what()

The error message could be obtained via the 'what' member function, which is virtual, const and noexcept.

```
void fun() {
  throw std::runtime_error("I love watermelons.");
}
int main() {
  try {
    fun();
  } catch (const std::runtime_error &re) {
    std::cout << re.what() << std::endl;
  }
}
```

Output:

```
I love watermelons.
```

Catch an exception

```
void f(const std::vector<int> &v) {
  try {
    auto i = 42;
    auto copy = v;
    int x = copy.at(100);
    g(x);
  } catch (const std::bad_alloc &ba) {
    // deal with shortage of memory
  } catch (const std::out_of_range &oor) {
    // deal with illegal subscript '100'
  } catch (...) {
    // What else may happen (probably in 'g(x)')? We are not sure.
    throw; // Throw the exception again.
  }
  std::cout << "returns.\n";
}
```

Catch an exception

```cpp
void f(const std::vector<int> &v) {
  try {
    auto i = 42;
    auto copy = v;
⚠   int x = copy.at(100);      throws std::out_of_range
    g(x);
  } catch (const std::bad_alloc &ba) {
    // deal with shortage of memory
  } catch (const std::out_of_range &oor) {
    // deal with illegal subscript '100'
  } catch (...) {
    // What else may happen (probably in 'g(x)')? We are not sure.
    throw; // Throw the exception again.
  }
  std::cout << "returns\n";
}
```

Catch an exception

```
void f(const std::vector<int> &v) {
  try {
    auto i = 42;
    auto copy = v;  `copy' is destroyed
    int x = copy.at(100);
    g(x);
  } catch (const std::bad_alloc &ba) {
    // deal with shortage of memory
  } catch (const std::out_of_range &oor) {
    // deal with illegal subscript '100'
  } catch (...) {
    // What else may happen (probably in 'g(x)')? We are not sure.
    throw; // Throw the exception again.
  }
  std::cout << "returns\n";
}
```

Catch an exception

```
void f(const std::vector<int> &v) {
  try {
    auto i = 42;        `i' is destroyed
    auto copy = v;
    int x = copy.at(100);
    g(x);
  } catch (const std::bad_alloc &ba) {
    // deal with shortage of memory
  } catch (const std::out_of_range &oor) {
    // deal with illegal subscript '100'
  } catch (...) {
    // What else may happen (probably in 'g(x)')? We are not sure.
    throw; // Throw the exception again.
  }
  std::cout << "returns\n";
}
```

Catch an exception

```cpp
void f(const std::vector<int> &v) {
  try {
    auto i = 42;
    auto copy = v;
    int x = copy.at(100);
    g(x);
  } catch (const std::bad_alloc &ba) {  Not matched
    // deal with shortage of memory
  } catch (const std::out_of_range &oor) {
    // deal with illegal subscript '100'
  } catch (...) {
    // What else may happen (probably in 'g(x)')? We are not sure.
    throw; // Throw the exception again.
  }
  std::cout << "returns\n";
}
```

Catch an exception

```cpp
void f(const std::vector<int> &v) {
  try {
    auto i = 42;
    auto copy = v;
    int x = copy.at(100);
    g(x);
  } catch (const std::bad_alloc &ba) {
    // deal with shortage of memory
  } catch (const std::out_of_range &oor) {  Matched
    // deal with illegal subscript '100'
  } catch (...) {
    // What else may happen (probably in 'g(x)')? We are not sure.
    throw; // Throw the exception again.
  }
  std::cout << "returns\n";
}
```

Catch an exception

```cpp
void f(const std::vector<int> &v) {
  try {
    auto i = 42;
    auto copy = v;
    int x = copy.at(100);
    g(x);
  } catch (const std::bad_alloc &ba) {
    // deal with shortage of memory
  } catch (const std::out_of_range &oor) {
    // deal with illegal subscript '100'
  } catch (...) {
    // What else may happen (probably in 'g(x)')? We are not sure.
    throw; // Throw the exception again.
  }
  std::cout << "returns\n";
}
```

Catch an exception

```
void f(const std::vector<int> &v) {
  try {
    auto i = 42;
    auto copy = v;
    int x = copy.at(100);
    g(x);
  } catch (const std::bad_alloc &ba) {
    // deal with shortage of memory
  } catch (const std::out_of_range &oor) {
    // deal with illegal subscript '100'
  } catch (...) {
    // What else may happen (probably in 'g(x)')? We are not sure.
    throw; // Throw the exception again.
  }
  std::cout << "returns\n";  Control flow continues here
}
```

Catch by base class

`operator new[]` raises `std::bad_alloc` when out of memory.

- But if the array-new length is obviously invalid, an instance of `std::bad_array_new_length` is raised.

```
new int[-1]; // negative size
new int[3]{2, 3, 4, 6, 8}; // too many initializers
new int[LONG_MAX][100]; // too large
```

Catch by base class

`operator new[]` raises `std::bad_alloc` when out of memory.

- But if the array-new length is obviously invalid, an instance of `std::bad_array_new_length` is raised.

```
new int[-1]; // negative size
new int[3]{2, 3, 4, 6, 8}; // too many initializers
new int[LONG_MAX][100]; // too large
```

- `catch (const std::bad_alloc &)` also catches it, because of **inheritance**:

Catch by base class

```cpp
try {
  do_something();
} catch (const std::runtime_error &re) {
  // deal with runtime_error
} catch (const std::exception &e) {
  // deal with other kinds of exceptions
} catch (...) {
  // deal with other things
}
```

Catch by base class

```
try {
  do_something();
} catch (const std::runtime_error &re) {
  // deal with runtime_error
} catch (const std::exception &e) {
  // deal with other kinds of exceptions
} catch (...) {
  // deal with other things
}
```

Note: Other things (e.g. a string) can also be `thrown`.

```
throw "I don\'t want to talk to you.";
throw 42;
```

In this case, these things are caught by `catch (...)`.

Catch by base class

catch clauses are examined one-by-one.

```
try {
  do_something();
} catch (const std::exception &e) {
  std::cout << "exception\n";
} catch (const std::runtime_error &re) {
  std::cout << "runtime_error\n";
} catch (...) {
  // deal with other things
}
```

If an instance of `std::runtime_error` is thrown, it will be caught by "`catch (const std::exception &)`" instead of "`catch (const std::runtime_error &)`" in this case.

Stack unwinding

```
void fun() {
  int i = 42;
  std::vector<int> v;
⚠v.at(i) = 10;     throws std::out_of_range
}
int main() {
  try {
    std::string str("Hello");
    fun();
  } catch (...) {}
}
```

Stack unwinding

```
void fun() {
  int i = 42;
  std::vector<int> v;  `v' is destroyed
  v.at(i) = 10;
}
int main() {
  try {
    std::string str("Hello");
    fun();
  } catch (...) {}
}
```

Stack unwinding

```
void fun() {
  int i = 42;     `i' is destroyed
  std::vector<int> v;
  v.at(i) = 10;
}
int main() {
  try {
    std::string str("Hello");
    fun();
  } catch (...) {}
}
```

Stack unwinding

```
void fun() {
  int i = 42;
  std::vector<int> v;
  v.at(i) = 10;
}
int main() {
  try {
    std::string str("Hello");
    fun();    Control flow returns here
  } catch (...) {}
}
```

Stack unwinding

```
void fun() {
  int i = 42;
  std::vector<int> v;
  v.at(i) = 10;
}
int main() {
  try {
    std::string str("Hello");  `str' is destroyed
    fun();
  } catch (...) {}
}
```

Stack unwinding

```
void fun() {
  int i = 42;
  std::vector<int> v;
  v.at(i) = 10;
}
int main() {
  try {
    std::string str("Hello");
    fun();
  } catch (...) {}  The exception is caught.
}
```

Notes

- The `try` block and `catch` blocks are independent scopes. Objects declared in the `try` block cannot be used in `catch` blocks.
- When an exception occurs, local objects in the `try` block are destroyed before the exception is caught.
- Stack unwinding is only guaranteed to happen for **caught** exceptions.
- If an exception is thrown and not caught, '`std::terminate`' will be called to terminate the program. (defined in `<exception>`)

Function-try-block

A function-try-block is typically useful for a constructor.

```
class Dynarray {
public:
  Dynarray(std::size_t n)
      try : m_length(n), m_storage(new int[n]{}) {}
  catch (const std::bad_alloc &ba) {
    std::cerr << "No enough memory.\n";
    throw;
  }
};
```

- Exceptions raised both in constructor initializer list and function body can be caught.
- Non-static data members cannot be referred to in such catch blocks. (Why?)

Function-try-block

A function-try-block is typically useful for a constructor.

```
class Dynarray {
public:
  Dynarray(std::size_t n)
      try : m_length(n), m_storage(new int[n]{}) {}
  catch (const std::bad_alloc &ba) {
    std::cerr << "No enough memory.\n";
    throw;
  }
};
```

- Exceptions raised both in constructor initializer list and function body can be caught.
- Non-static data members cannot be referred to in such catch blocks. (Why?)
  - An exception thrown in the constructor indicates that the initialization has failed!
  - Once an exception is thrown, everything initialized in the try block are destroyed.

Contents

Exception safety guarantees

Exception-safe functions offer one of three guarantees:

- **Nothrow guarantee**: Promise never to throw exceptions.
- **Strong guarantee**: Promise that if an exception is thrown, the state of the program is **unchanged**, as if the function had not been called ("roll back").
- **Weak guarantee** (basic guarantee): Promise that if an exception is thrown, everything in the program remains in a valid state (though possibly changed).
  - No objects or data structures become corrupted.
  - All class invariants are satisfied. For example, a `Polynomial` should have at least one coefficient (the constant term). In `Dynarray`, `m_length` should represent the length of the memory block that `m_storage` points to.

Effective C++ Item 29: Strive for exception-safe code.

Exception safety guarantees

The level of an exception safety guarantee measures how hard it is to recover from an exception.

```
void foo(std::vector<int> &values) {
  try {
    values = something();
  } catch (const std::bad_alloc &ba) {
    // Can we assume that 'values' is still in a valid state? (weak guarantee)
    // Can we assume that 'values' remains unchanged? (strong guarantee)
  }
}
```

Exception safety guarantees

Effective C++ Item 29:

> *A software system is* ***either exception-safe or it's not****. There's no such thing as a partially exception-safe system. If a system has* ***even a single function*** *that's not exception-safe, the system as a whole is not exception-safe.*
>
> *A function can usually offer a guarantee no stronger than the* ***weakest*** *guarantee of the functions it calls.*

Which exception safety guarantee?

```
class Dynarray {
  int *m_storage;
  std::size_t m_length;

public:
  Dynarray &operator=(const Dynarray &other) {
    if (this != &other) {
      delete[] m_storage;
      m_storage = new int[other.m_length]; // May throw std::bad_alloc
      std::copy(other.m_storage, other.m_storage + other.m_length, m_storage);
      m_length = other.m_length;
    }
    return *this;
  }
};
```

Which exception safety guarantee?

```
class Dynarray {
  int *m_storage;
  std::size_t m_length;

public:
  Dynarray &operator=(const Dynarray &other) {
    if (this != &other) {
      delete[] m_storage;
      m_storage = new int[other.m_length]; // May throw std::bad_alloc
      std::copy(other.m_storage, other.m_storage + other.m_length, m_storage);
      m_length = other.m_length;
    }
    return *this;
  }
};
```

No guarantee at all! The data pointed to by `m_storage` has already been destroyed before the exception happens.

Which exception safety guarantee?

```
class Dynarray {
public:
  Dynarray &operator=(const Dynarray &other) {
    auto new_data = new int[other.m_length];
    std::copy(other.m_storage, other.m_storage + other.m_length, new_data);
    delete[] m_storage;
    m_storage = new_data;
    m_length = other.m_length;
    return *this;
  }
};
```

Which exception safety guarantee?

```
class Dynarray {
public:
  Dynarray &operator=(const Dynarray &other) {
    auto new_data = new int[other.m_length];
    std::copy(other.m_storage, other.m_storage + other.m_length, new_data);
    delete[] m_storage;
    m_storage = new_data;
    m_length = other.m_length;
    return *this;
  }
};
```

Strong guarantee. Nothing has been changed before `new[]` on the first line throws an exception.

Which exception safety guarantee?

```
class Dynarray {
public:
  Dynarray &operator=(const Dynarray &other) {
    m_length = other.m_length;
    auto new_data = new int[m_length];
    std::copy(other.m_storage, other.m_storage + m_length, new_data);
    delete[] m_storage;
    m_storage = new_data;
    return *this;
  }
};
```

Which exception safety guarantee?

```cpp
class Dynarray {
public:
  Dynarray &operator=(const Dynarray &other) {
    m_length = other.m_length;
    auto new_data = new int[m_length];
    std::copy(other.m_storage, other.m_storage + m_length, new_data);
    delete[] m_storage;
    m_storage = new_data;
    return *this;
  }
};
```

No guarantee. `m_length` is changed too early. If `new[]` throws, `m_length` is not equal to the length of the memory block that `m_storage` points to.

Which exception safety guarantee?

The "copy-and-swap" idiom, talked about in previous recitations.

```
class Dynarray {
public:
  void swap(Dynarray &other) noexcept {
    using std::swap;
    swap(m_length, other.m_length);
    swap(m_storage, other.m_storage);
  }
  Dynarray &operator=(const Dynarray &other) {
    Dynarray(other).swap(*this);
    return *this;
  }
};
```

Which exception safety guarantee?

The "copy-and-swap" idiom, talked about in previous recitations.

```
class Dynarray {
public:
  void swap(Dynarray &other) noexcept {
    using std::swap;
    swap(m_length, other.m_length);
    swap(m_storage, other.m_storage);
  }
  Dynarray &operator=(const Dynarray &other) {
    Dynarray(other).swap(*this);
    return *this;
  }
};
```

Strong guarantee. The only thing that may throw an exception is `Dynarray(other)` (which allocates memory through `new[]`).

Contents

noexcept vs throw()

Before C++11, a function may declare in advance what exception(s) it may throw.

```
void *operator new(std::size_t size) throw(std::bad_alloc); // May throw std::bad_alloc.
```

noexcept vs throw()

Before C++11, a function may declare in advance what exception(s) it may throw.

```
void *operator new(std::size_t size) throw(std::bad_alloc); // May throw std::bad_alloc.
```

To a function that offers nothrow guarantee: `throw()`

```
int add(int a, int b) throw() {
  return a + b;
}
```

noexcept vs throw()

People came to realize that it is **whether the function throws exceptions or not** that really matters.

- In most cases, knowing the specific exception type offers no more help.
- In most cases, all we can do is to catch it through `catch(...)`, report it or do some logging, and then throw it again through `throw;`.

noexcept vs throw()

People came to realize that it is **whether the function throws exceptions or not** that really matters.

- In most cases, knowing the specific exception type offers no more help.
- In most cases, all we can do is to catch it through `catch(...)`, report it or do some logging, and then throw it again through `throw;`.

Since C++11, declare `noexcept` for non-throwing functions.

```
class Dynarray {
public:
  void swap(Dynarray &other) noexcept {
    std::swap(m_storage, other.m_storage);
    std::swap(m_length, other.m_length);
  }
};
```

The `throw()` specifiers have been deprecated and removed in modern C++.

`noexcept`

The `noexcept` specifier makes it possible for more optimization.

- When an exception is thrown inside a `noexcept` function, whether the stack is unwound is *implementation-defined*.
  - Compilers need not keep the runtime stack in an unwindable state.
- Certain functions must be `noexcept` so that they can be called by standard library functions.

Move operations are often noexcept.

Recall that std::vector<T> will allocate a larger block of memory when the current memory capacity is not enough.

```
template <typename T>
class vector {
  T *m_storage;
  T *m_end_of_elem, *m_end_of_storage; // Possible implementation.
public:
  void push_back(const T &x) {
    if (size() == capacity())
      reallocate(capacity() == 0 ? 1 : capacity() * 2);
    construct_object_at(m_end_of_elem, x);
    ++m_end_of_elem;
  }
private:
  void reallocate(std::size_t new_capacity);
};
```

Move operations are often noexcept.

Before C++11, the elements are **copied** to the new memory block.

- Note: `std::vector<T>` does not use `new[]`, because it needs to separate **object creation** from **memory allocation**. The following code uses `allocate_memory`, `construt_object_at` and `destroy_and_deallocate` for demonstration only.

```
template <typename T>
class vector {
  void reallocate(std::size_t new_capacity) {
    auto new_storage = allocate_memory(new_capacity), p = new_storage;
    for (auto old_data = m_storage; old_data != m_end_of_elem; ++old_data)
      construct_object_at(p++, *old_data);
    destroy_and_deallocate(m_storage);
    m_storage = new_storage;
    m_end_of_elem = p;
    m_end_of_storage = m_storage + new_capacity;
  }
};
```

Move operations are often noexcept.

To offer **strong exception safety guarantee**, `reallocate` needs to "recall" the operations once an exception is encountered.

```
template <typename T>
class vector {
  void reallocate(std::size_t new_capacity) {
    auto new_storage = allocate_memory(new_capacity), p = new_storage;
    try {
      for (auto old_data = m_storage; old_data != m_end_of_elem; ++old_data)
        construct_object_at(p++, *old_data);
    } catch (...) {
      while (p != new_storage) destroy(--p);
      deallocate(new_storage); throw;
    }
    destroy_and_deallocate(m_storage);
    m_storage = new_storage; m_end_of_elem = p; m_end_of_storage = m_storage + new_capacity;
  }
};
```

Move operations are often noexcept.

Since C++11, a reasonable optimization is to **move** elements, instead of copying them.

```
template <typename T>
class vector {
  void reallocate(std::size_t new_capacity) {
    auto new_storage = allocate_memory(new_capacity), p = new_storage;
    try {
      for (auto old_data = m_storage; old_data != m_end_of_elem; ++old_data)
        construct_object_at(p++, std::move(*old_data));
    } catch (...) {
      // Wait ... The elements are moved! How can we recover them?
    }
    // ...
  }
};
```

Move operations are often noexcept.

Since C++11, a reasonable optimization is to **move** elements, instead of copying them. Unlike copy, **move** of an element is a modifying operation.

- The old elements are modified during this procedure.
- If a move throws an exception, there is no way of rolling back!

Due to this consideration, `std::vector<T>` uses the move operation of `T` **only when it is `noexcept`**.

```
for (auto old_data = m_storage; old_data != m_end_of_elem; ++old_data)
  construct_object_at(p++, std::move_if_noexcept(*old_data));
```

noexcept

noexcept is only a **logical guarantee**. A noexcpet function may still

- call functions that are not noexcept, or
- throw exceptions under certain circumstances.

Summary

- `throw` an exception
- `try`-`catch`: catch and handle exceptions
- Standard library exceptions: `std::exception`, `std::bad_alloc`, `std::logic_error`, `std::runtime_error`, `std::bad_cast`, ...
- Stack unwinding:
  - Destroys local objects in reverse order of initialization.
  - Only guaranteed to happen for **caught** exceptions.

Summary

- Exception safety guarantee:
  - Nothrow guarantee
  - Strong guarantee: “roll back”
  - Weak guarantee: Promises that everything is in a valid state.
- `noexcept`: specifies that a function offers nothrow guarantee.
- Move operations are often `noexcept`, because unlike copy, move is a modifying operation.