# Exception Handling and Exception Safety

GKxx

July 7, 2022

# Contents

# Input Failure

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

What happens when the input is not an integer?

# 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?
- Buffer overflow?

Detecting buffer overflow may not be easy.

# 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 << "Invalid
        arguments for
        strcpy.\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';
}
```

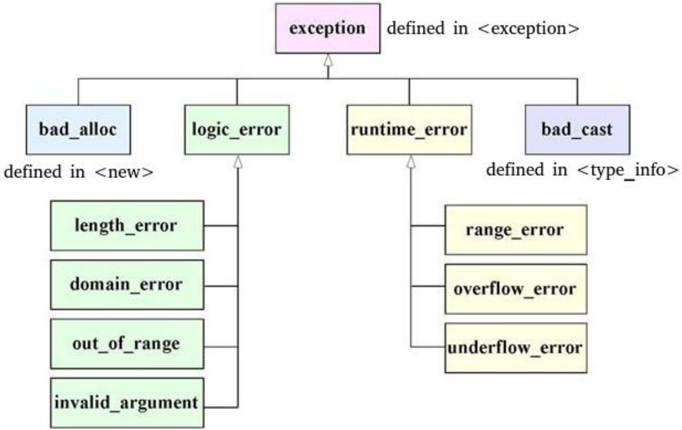
https://blog.csdn.net/myan/article/details/1921

# Contents

# Throwing an Exception

```
void strcpy(char *dest, const char *source) {
  if (!dest || !source)
    throw std::invalid_argument("Null pointers passed
        to strcpy.");
  while (*source)
    *dest++ = *source++;
  *dest = '\0';
}
```

# Standard Exceptions




- `logic_error`, `runtime_error` and their subclasses are defined in `<stdexcept>`.

# 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 casting 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 casting 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`.

# 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 probably causes a
    segmentation fault.
```

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

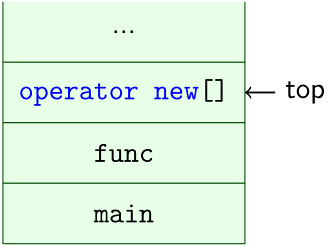
# Standard Exceptions

Let our `Array` do the same thing?

```
template <typename T>
class Array {
 public:
  const T &at(std::size_t n) const {
    if (n >= m_size)
      throw std::out_of_range("Array subscript out of
          range.");
    return m_data[n];
  }
  T &at(std::size_t n) {
    return const_cast<T &>(
      static_cast<const Array<T> *>(this)->at(n);
    ) // see Effective C++ Item 3
  }
  // ...
};
```

# Stack Unwinding

```
void func(int n) {
  int x = 42;
  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) {
  int x = 42;
⚠int *p = new int[n];
  // ...
}
int main() {
  int size = 100;
  func(size);
  // ...
}
```

1. `std::bad_alloc` is raised in `operator new[]`.

# Stack Unwinding



```
void func(int n) {
  int x = 42;
  int *p = new int[n];
  // ...
}
int main() {
  int size = 100;
  func(size);
  // ...
}
```

1. `std::bad_alloc` is raised in `operator new[]`.
2. Control flow returns to `func`.

# Stack Unwinding



```
void func(int n) {
  int x = 42;
  int *p = new int[n];
  // ...
}
int main() {
  int size = 100;
  func(size);
  // ...
}
```

1. `std::bad_alloc` is raised in `operator new[]`.
2. Control flow returns to `func`.
3. `x` is destroyed.

# Stack Unwinding



```
void func(int n) {
  int x = 42;
  int *p = new int[n];
  // ...
}
int main() {
  int size = 100;
  func(size);
  // ...
}
```

1. `std::bad_alloc` is raised in `operator new[]`.
2. Control flow returns to `func`.
3. `x` is destroyed.
4. `n` is destroyed.

## Stack Unwinding



```
void func(int n) {
  int x = 42;
  int *p = new int[n];
  // ...
}
int main() {
  int size = 100;
  func(size);
  // ...
}
```

1. `std::bad_alloc` is raised in `operator new[]`.
2. Control flow returns to `func`.
3. `x` is destroyed.
4. `n` is destroyed.
5. Control flow returns to `main`.

# Stack Unwinding



```
void func(int n) {
  int x = 42;
  int *p = new int[n];
  // ...
}
int main() {
  int size = 100;
  func(size);
  // ...
}
```

1. `std::bad_alloc` is raised in `operator new[]`.
2. Control flow returns to `func`.
3. `x` is destroyed.
4. `n` is destroyed.
5. Control flow returns to `main`.
6. `size` is destroyed.

# Stack Unwinding

```
void func(int n) {
  int x = 42;
  int *p = new int[n];
  // ...
}
int main() {
  int size = 100;
  func(size);
  // ...
}
```

1. `std::bad_alloc` is raised in `operator new[]`.
2. Control flow returns to `func`.
3. `x` 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



```
void func(int n) {
  int x = 42;
  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 {
    int i = 42;
    std::vector<int> 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? idk
    throw; // Throw the exception again.
  }
  std::cout << "returns.\n";
}
```

# Catch an Exception



Suppose `std::out_of_range` is raised.

```
void f(const std::vector<int> &v) {
  try {
    int i = 42;
    std::vector<int> 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? idk
    throw; // Throw the exception again.
  }
  std::cout << "returns\n";
}
```

# Catch an Exception



Suppose `std::out_of_range` is raised.

```
void f(const std::vector<int> &v) {
  try {
    int i = 42;
    std::vector<int> 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? idk
    throw; // Throw the exception again.
  }
  std::cout << "returns\n";
}
```

# Catch an Exception

Suppose `std::out_of_range` is raised.

```
void f(const std::vector<int> &v) {
  try {
    int i = 42;          'i' is destroyed
    std::vector<int> 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? idk
    throw; // Throw the exception again.
  }
  std::cout << "returns\n";
}
```

# Catch an Exception

Suppose std::out_of_range is raised.

```
void f(const std::vector<int> &v) {
  try {
    int i = 42;
    std::vector<int> 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? idk
    throw; // Throw the exception again.
  }
  std::cout << "returns\n";
}
```

# Catch an Exception



Suppose `std::out_of_range` is raised.

```
void f(const std::vector<int> &v) {
  try {
    int i = 42;
    std::vector<int> 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? idk
    throw; // Throw the exception again.
  }
  std::cout << "returns\n";
}
```

# Catch an Exception



Suppose `std::out_of_range` is raised.

```
void f(const std::vector<int> &v) {
  try {
    int i = 42;
    std::vector<int> 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? idk
    throw; // Throw the exception again.
  }
  std::cout << "returns\n";
}
```

# Catch an Exception



Suppose `std::out_of_range` is raised.

```
void f(const std::vector<int> &v) {
  try {
    int i = 42;
    std::vector<int> 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? idk
    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 `throw`n.

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

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 "`const std::exception &`" instead of "`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

```cpp
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>`)

# try-catch for Constructors

```
template <typename T>
class Array {
 public:
  Array(std::size_t n)
      try : m_size(n), m_data(new T[n]{}) {}
  catch (const std::bad_alloc &ba) {
    std::cerr << "No enough memory.\n";
    throw;
  }
};
```

Notes:

- 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?)

# Contents

# User-defined Exceptions



```
class Wrong_answer : public std::logic_error {
 public:
  Wrong_answer(std::size_t line_no)
      : std::logic_error("Wrong answer on line "
          + std::to_string(line_no)) {}
};
#define assert(X)                              \
  { if (!(X)) throw Wrong_answer(__LINE__); }
int main() {
  int a = rand(), b = rand();
  int ans = add(a, b);
  assert(ans == a + b);
  return 0;
}
```

# 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).
- **Weak guarantee** (basic guarantee): Promise that if an exception is thrown, everything in the program remains in a valid state.
  - No objects or data structures become corrupted.
  - All class invariants are satisfied.

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

# 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.*

# 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.*

# Contents

# noexcept vs throw()

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

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

# noexcept vs throw()



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

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

To declare that a function does not throw exceptions:

```
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.

# noexcept vs throw()



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

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

```
template <typename T>
void swap(Array<T> &a, Array<T> &b) noexcept {
  a.swap(b);
}
```

# noexcept vs throw()



People came to realize that it is **whether the function throws exceptions or not** that really matters.
Since C++11, declare `noexcept` for non-throwing functions.

```
template <typename T>
void swap(Array<T> &a, Array<T> &b) noexcept {
  a.swap(b);
}
```

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, the stack is *possibly* unwound.
  - 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 constructors and move assignment operators.

# `noexcept`

`noexcept` is not checked in compile-time. A `noexcpet` function may still

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

# Arguments to noexcept

`noexcept` may take one argument that is a constant expression and is convertible to `bool`.

```
// noexcept iff T is nothrow-copy-constructible.
template <typename T>
void fun() noexcept(
    std::is_nothrow_copy_constructible<T>::value) {
  // ...
}
```

`noexcept` is equivalent to `noexcept(true)`.

# The noexcept Operator

noexcept can also work as an operator, which returns a bool value indicating whether an expression throws exceptions.

```
template <typename T>
class Box {
  T thing;
 public:
  void swap(Box<T> &other)
      noexcept(noexcept(std::swap(thing, other.thing)))
  {
    std::swap(thing, other.thing);
  }
};
```

# Contents

# Which Exception-safety Guarantee?



```
class Array {
  int *m_data;
  std::size_t m_size;

 public:
  Array &operator=(const Array &other) {
    if (this != &other) {
      delete[] m_data;
      m_data = new int[other.m_size];
      std::copy(other.m_data,
                other.m_data + other.m_size, m_data);
      m_size = other.m_size;
    }
    return *this;
  }
};
```

# Which Exception-safety Guarantee?



```
class Array {
  int *m_data;
  std::size_t m_size;

 public:
  Array &operator=(const Array &other) {
    if (this != &other) {
      delete[] m_data;
      m_data = new int[other.m_size];
      std::copy(other.m_data,
                other.m_data + other.m_size, m_data);
      m_size = other.m_size;
    }
    return *this;
  }
};
```

**It does not offer even the basic guarantee.**

# Which Exception-safety Guarantee?

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

# Which Exception-safety Guarantee?

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

**Strong guarantee.**

# Which Exception-safety Guarantee?



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

# Which Exception-safety Guarantee?



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

No exception-safety guarantee.

# Which Exception-safety Guarantee?



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

# Which Exception-safety Guarantee?

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

Strong guarantee.

# Which Part may Throw?

```cpp
// For simplicity, assume T is default-constructible
    and copy-assignable.
template <typename T>
class Array {
  T *m_data;
  std::size_t m_size;

 public:
  Array(const Array &other)
      : m_data(new T[other.m_size]),
        m_size(other.m_size) {
    std::copy(other.m_data,
              other.m_data + other.m_size, m_data);
  }
};
```

# Which Exception-safety Guarantee?



```
template <typename T>
class Array {
 public:
  Array(const Array &other)
      : m_data(new T[other.m_size]),
        m_size(other.m_size) {
    std::copy(other.m_data,
              other.m_data + other.m_size, m_data);
  }
};
```

# Which Exception-safety Guarantee?



```
template <typename T>
class Array {
 public:
  Array(const Array &other)
      : m_data(new T[other.m_size]),
        m_size(other.m_size) {
    std::copy(other.m_data,
              other.m_data + other.m_size, m_data);
  }
};
```

**No guarantee.** If an exception occurs when copying, `m_size` and `m_data` will be destroyed, resulting in **memory leak**.

# Make it Exception-safe

```
template <typename T>
class Array {
 public:
  Array(const Array &other)
      : m_data(new T[other.m_size]),
        m_size(other.m_size) {
    try {
      std::copy(other.m_data,
                other.m_data + other.m_size, m_data);
    } catch (...) {
      delete[] m_data; // Avoid memory leak
      throw; // Let the caller know it!
    }
  }
};
```

# Which Exception-safety Guarantee?

```
template <typename T>
class Array {
 public:
  Array &operator=(const Array &other) {
    auto new_data = new T[other.m_size];
    std::copy(other.m_data,
                   other.m_data + other.m_size, new_data);
    delete[] m_data;
    m_data = new_data;
    m_size = other.m_size;
    return *this;
  }
};
```

# Which Exception-safety Guarantee?



```cpp
template <typename T>
class Array {
 public:
  Array &operator=(const Array &other) {
    auto new_data = new T[other.m_size];
    std::copy(other.m_data,
              other.m_data + other.m_size, new_data);
    delete[] m_data;
    m_data = new_data;
    m_size = other.m_size;
    return *this;
  }
};
```

No guarantee.

# Make it Exception-safe

```
template <typename T>
class Array {
 public:
  Array &operator=(const Array &other) {
    auto new_data = new T[other.m_size];
    try {
      std::copy(other.m_data,
                other.m_data + other.m_size, new_data);
    } catch (...) {
      delete[] new_data;
      throw;
    }
    delete[] m_data;
    m_data = new_data;
    m_size = other.m_size;
    return *this;
  }
};
```

# Which Exception-safety Guarantee?

```
template <typename T>
class Array {
 public:
  void swap(Array &other) noexcept {
    using std::swap;
    swap(m_size, other.m_size);
    swap(m_data, other.m_data);
  }
  Array &operator=(const Array &other) {
    Array(other).swap(*this);
    return *this;
  }
};
```

# Which Exception-safety Guarantee?



```cpp
template <typename T>
class Array {
 public:
  void swap(Array &other) noexcept {
    using std::swap;
    swap(m_size, other.m_size);
    swap(m_data, other.m_data);
  }
  Array &operator=(const Array &other) {
    Array(other).swap(*this);
    return *this;
  }
};
```

**Strong guarantee.**