Concepts of C++ Programming

Lecture 10: Exceptions and Advanced Memory Management

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2024/25

C++ Exceptions[130]

- Exceptions have similar semantics as in other languages

$\Rightarrow$ Transfer control and propagate information up the call stack

- Thrown by `throw`, `new`, and some standard library functions

- Exceptions can be handled in `try`–`catch` blocks
- Unhandled exceptions lead to termination

- When transferring control up the call stack, the runtime performs *stack unwinding*
- All objects with automatic storage duration are destructed

$\rightsquigarrow$ Correct behavior of RAII classes

[130] https://en.cppreference.com/w/cpp/language/exceptions

Throwing Exceptions[131]

- `throw` *expression*;
- Objects of any complete type can be thrown
- Exception object (heap-allocaetd) copy-initialized with expression
- Typically a subclass of `std::exception`

```
#include <exception>
void foo(unsigned i) {
  if (i == 42)
    throw 42;

  throw std::exception();
}
```

[131] https://en.cppreference.com/w/cpp/language/throw

Handling Exceptions[132]

- `try { ... } catch (`*declaration*`) { ... };`
- Exceptions occuring during `try`-block can be handled in `catch`-block
- Declaration type determines which type of exception is caught

```
#include <exception>
void bar() {
  try {
    foo(42);
  } catch (int i) { // handle exception of type int
  } catch (const std::exception& e) { // handle exception of type std::exception
  } catch (...) { // catch-all
  }
}
```

[132] https://en.cppreference.com/w/cpp/language/catch

Exceptions: Example

Quiz: What is problematic about this code?

```
#include <memory>
#include <print>
int foo(const int& x) { return x != 0 ? throw x : x; }
int bar(int x) { std::unique_ptr<int> ui(new int);
      *ui = x * 2; return foo(*ui); }
int main() {
  try { std::print("ok!␣{}\n", bar(21));
  } catch (int x) {}
}
```

A. Compile error: `throw` is a statement, not an expression.

B. Memory leak: Memory from `new` is leaked on exception.

C. Unhandled exception: the exception has type `const int&`.

D. Nothing: the program terminates with exit code zero.

Exceptions: Miscellaneous

- In a catch block, the current exception can be re-thrown
  - Syntax: `throw;`
  - E.g., to clean up resources and propagate exception further
- Functions can be marked as `noexcept`
  - Part of the function type
  - Indicates that the function will never throw an exception
  - Any exceptions that would propagate cause program termination
- Destructors, move constructors/assignment must not throw exceptions

Quiz: Which answer is correct?

```
#include <print>
struct A { A() { throw 1; } };
struct B {
    A a;
    B() try : a() {
    } catch (int x) {
        std::println("whoops?␣{}", x);
        throw; // rethrow exception
    }
};
int main() { try { B b; } catch (int x) { return x; } }
```

A. Compile error: Cannot use `try` outside function body.

B. The `throw;` is not necessary.

C. a is life in the `catch` block of the constructor.

D. No object of type `A` can be constructed, but objects of type `B` can be.

Exceptions: Performance and Code Size Considerations

- Exception handling (stack unwinding) is rather expensive
- Low overhead if no exceptions are thrown

⇒ In any case, exceptions should be used rarely

- The mere *possibility* of exceptions inhibits some optimizations
  - Increased control flow complexity, more state must be kept in stack memory
- For every possibly throwing call, corresponding cleanup code must be generated
- Unwind tables that map code location to cleanup landing pad can grow large

⇝ *Enabling* exceptions can have substantial code size impact

  - To disable exceptions: `-fno-exceptions`

Exceptions: Guidelines

- Use exceptions only in rare cases
- E.g., dynamic runtime errors (e.g., malformed data)

- Do not use exceptions for programmer errors
  - Use assertions for this
- Do not use exceptions for control flow
  - Use regular control flow operations for this

- Generally: exceptions should be **avoided** where possible
- When not using exceptions at all, disable them via a compiler flag

`operator new`

- `operator new` (`<new>`) can take arguments[133]
- Default, implicitly: `operator new (size)`
- Example: overload with extra arg `std::nothrow_t`

```
#include <new>
#include <array>
#include <print>
struct A { /* ... */ };
int main() {
  // Will throw std::bad_alloc
  auto* p1 = new std::array<int, 1000000000000>();
  // Will return nullptr on allocation failure
  auto* p2 = new(std::nothrow) std::array<int, 1000000000000>();
  if (!p2)
    std::println("allocation␣failed!");
}
```

[133] https://en.cppreference.com/w/cpp/memory/new/operator_new

Manually managing memory

- Sometimes, the default memory management operations are not enough
  - E.g., repeatedly calling `new` (explicit or implicit) is too expensive
  - E.g., for reusing already available memory

$\rightsquigarrow$ Placement new: construct object in already allocated storage

- Manually call constructor and destructor

Placement new

- `operator new(size, void* ptr)`
  - Returns `ptr` without doing any allocation
- Alignment must be ensured manually

```
#include <cstddef>
#include <new>
struct A { /* ... */ };
int main() {
  alignas(A) std::byte buffer[sizeof(A)];
  A* a = new(buffer) A();
  // ... do something with a
  a->~A(); // we must explicitly call the destructor
}
```

Placement new and Lifetime

- Placement new ends lifetime of overlapping objects; creates new object
- Lifetime is nested within the underlying storage

```
struct A { };
int main() {
  A* a1 = new A(); // lifetime of a1 begins, storage begins
  a1->~A(); // lifetime of a1 ends
  A* a2 = new (a1) A(); // lifetime of a2 begins
  delete a2; // lifetime of a2 ends, storage ends
}
```

Quiz: How to deallocate s1? What to write instead of XXX?

```
template <class T, size_t N>
class TAlloc {
  alignas(T) std::byte buffer[sizeof(T[N])];
  size_t cnt = 0;
public:
  T* make(T&& t) {
    void* vp = &buffer[sizeof(T)*cnt++];
    T* r = reinterpret_cast<T*>(vp);
    ::new(r) T(std::move(t));
    return r;
  }
};
int main() {
  TAlloc<std::string, 3> ta;
  auto* s1 = ta.make("Hello World!");
  // XXX
}
```

A. `delete(s1);`
B. `s1->~string();`
C. `s1->~basic_string();`
D. `ta.~TAlloc();`
E. Nothing, the strings are automatically freed at the end of `main`.

Placement new with unique_ptr

- std::unique_ptr<T, Deleter> – specify type of deleter
- Second parameter in constructor to specify deleter instance

- Default deleter calls delete
- For use with non-standard allocation, a custom deleter is required

- Code that uses custom allocators is typically rather complex
  $\Rightarrow$ unique_ptr is often not particularly useful in such contexts

Overloading `operator new`

- Classes can overload `operator new` and `operator delete`
- Can also provide overloads with extra arguments

- Rarely useful, e.g.:
  - Allocating extra storage after/before the object

union

- Class type that holds only one of its non-static members at a time
- Storage large enough to hold largest element
- All data members have the same address

- Writing to a union member *activates* it
- Reading an inactive union member is undefined behavior

```
union MyUnion { float f; long l; short a[2]; };
static_assert(sizeof(MyUnion) == sizeof(long));
int main() {
  MyUnion u; // f active, default-initialized
  u.f = 123.0; // f active
  u.a[1] = 12; // a active
  return u.a[1]; // ok
}
```

Union: Example

Quiz: What is the output of the program?

```
#include <print>
int main() {
  using Converter = union { float f; unsigned u; };
  std::println("{:08x}", Converter{32.5f}.u);
  return 0;
}
```

A. Compile error: Cannot have untyped union.

B. Compile error: Union initializer is ambiguous.

C. Undefined behavior: Program reads inactive union member.

D. The integer representation of `32.5f` (42020000).

`std::bit_cast`[134]

- For bitwise reinterpretation of object representations, use `std::bit_cast<TargetTy>()` from `<bit>`
  - Do not use `union` for this – C++ differs from C here
  - Do not use `reinterpret_cast`

[134] https://en.cppreference.com/w/cpp/numeric/bit_cast

Union with Non-Primitive Types

- `unions` can have non-primitive members
- `union` doesn't know which member is active...
- Lifetime needs to be managed explicitly outside of the union

- Typical use as part of a struct which tracks active element
- Can be used to implement more efficient variant
- Very difficult to get right

⇝ Prefer `std::variant`

Union with Non-Primitive Types: Example

```
union U {
  std::vector<int> v;
  std::string s;
  // needs explicit destructor -- can't do anything!
  // union doesn't know which member is active
  ~U() {}
};
int main() {
  U u{}; // constructs first element
  u.v.push_back(123);
  u.v.~vector<int>(); // lifetime of u.v ends
  new(&u.s) std::string("123"); // lifetime of u.s begins
  std::println("{}", u.s);
  u.s.~basic_string(); // lifetime of u.s ends
  // ~U() will be called, but is defined to do nothing
}
```

Implementing our own Vector

- At this point, we can implement our own vector

(see script)

Allocating Raw/Uninitialized Memory

- C `malloc`/`free` often work, but not always
- Problem: type might have increased alignment requirement

- `std::allocator<T>`[135] respects additional requirements
  - `allocate(elementCount)` – allocate an array suitable for n objects
  - `deallocate(ptr, elementCount)` – deallocate previously allocated memory

[135] https://en.cppreference.com/w/cpp/memory/allocator

Helper Functions for Handling Uninitialized Memory

- Provides more guarantees in case of an exception
- `std::uninitialized_move`
  – move range of elements into uninitialized memory
- `std::uninitialized_default_construct`
  – default-construct range of elements into uninitialized memory
- `std::destroy` – destruct range of elements

Exception Safety when Moving

- Move constructor/assignment might throw exceptions

Quiz: (Why) is this problematic?

A. Afterwards, vector might be in unrepairable state
B. Exception cannot be caught properly
C. New allocation will always be leaked
D. This is not a problem, just annoying

- `std::vector` guarantees exception safety
  - E.g., `push_back` guarantees to have no effect if any operations throws
- If move operations are not `noexcept`, elements will be copied instead

memcpy/memmove

- For primitive data types, constructing/destructing is not required
- `std::is_trivially_copyable_v<T>` – indicates whether byte-wise copying is possible
  - In fact, this is also possible for structs of trivially copyable types

- `std::memcpy(dest, src, count)` – copy bytes between non-overlapping regions
- `std::memmove(dest, src, count)` – copy bytes between regions
- In both cases, alignment of destination must be suitable

Custom Allocators

- Sometimes, the default allocator is not good enough
  - Many small allocations are expensive
  - All allocations have to be freed separately
  - Every allocation has memory overhead (e.g., tracking allocation size)
  - Requires synchronization in multi-threaded applications
  - Possibly bad locality
- Typical solution: bump pointer allocator
  - Allocate large chunk of memory once
  - Hand out slices for individual allocations
  - Free allocated memory when allocator is destroyed

Custom Allocators in C++

- Requirements specified by *Allocator*
  - In essence: `value_type`, `allocate`, `deallocate`
- Containers are allocator-aware and can use custom allocators

- Bump-ptr allocator in C++ standard library: `std::pmr::monotonic_buffer_resource`
  - Usable with `std::pmr::polymorphic_allocator` as allocator
  - Performance characteristics not that good (see inheritance later)
- For performance with many small allocations, custom allocators are often required

Exceptions and Advanced Memory Management – Summary

- C++ Exceptions allow for unordinary control flow transfers
- Almost everything can be thrown and caught
- Exception unwinding calls destructors of objects with automatic storage duration
- Objects can be constructed in allocated memory with placement new
- Required when memory allocation and object construction are separated
- `union`s provide an untagged overlapping storage
- Writing exception-safe code is difficult
- Custom allocators can substantially improve performance in some applications

Exceptions and Advanced Memory Management – Questions

- Why do some people see C++ exceptions as problematic?
- What are upsides and downsides of C++ exceptions?
- Why is writing exception-safe code difficult?
- What happens when an exception is thrown in a `noexcept` function?
- Why should move constructors/assignment be marked as `noexcept`?
- What requirements must be met for placement new?
- Why is using `union` much more difficult than in C?
- What are benefits of bump pointer allocators?