Effective C++ Programming

NIE-EPC (v. 2021):

UNDEFINED BEHAVIOR, OBJECT LIFETIME, STATIC AND DYNAMIC STORAGE ALLOCATION, INITIALIZATION AND DESTRUCTION

Undefined behavior

- *Recall:*
  - C++ standards define observable behavior of a C++ program on the abstract machine.
  - C++ implementations provide the same observable behavior on a particular computer system.
- Is the observable behavior always defined?
  - **No**, it is defined only if C++ standard rules/requirements are satisfied.
- What happens if they are violated?
  - Then, C++ standards **do not define program/code behavior**.
- This case is referred to as "*undefined behavior*" (*UB*).
- *Examples:*
  - signed integer overflow, dereferencing invalid pointer (pointer that does not point to an existing object of a given type), accessing out-of-bound array element, creating an object in not properly aligned storage, data race, ...

Undefined behavior (*cont.*)

- Once some standard rule is violated, the C++ standard no longer define behavior on the abstract machine.
- In practice, this violation mostly happens at runtime, but it can also happen at compile time.
  - In the first case, the behavior is undefined for the executed program.
  - In the second case, the behavior is undefined even for compile-time C++ implementation tools (compiler, linker,…).
- *Example program source file:*

```
#include <iostream>
#include <limits>

int main() {
  int i = std::numeric_limits<int>::max();
  i++;  // causes signed integer overflow => undefined behavior
  std::cout << i;
}
```

- *Question:* When does the overflow happen?

Undefined behavior (*cont.*)

```
int i = std::numeric_limits<int>::max();
i++;  // causes signed integer overflow => undefined behavior
```

- When does the overflow happen?
  1. With disabled optimizations, the calculation will likely be performed at runtime ⇒ the overflow will happen at runtime ⇒ the program behavior is undefined at runtime once the overflow takes place.
  2. With enabled optimizations, the calculation will likely be performed at compile time ⇒ the overflow will happen at compile time ⇒ the behavior is undefined already at compile time.
- Ad 1. Once the overflow occurs, the behavior of the program is no longer defined by the C++ standard.
  - It may crash, it may behave as expected, it may behave unexpectedly,…
- Ad 2. Once the overflow occurs, the behavior of the implementation tools is no longer defined by the C++ standard.
  - It may throw a compilation error or warning, it may crash, it may generate some machine code with some expected or unexpected behavior,…

Undefined behavior (*cont.*)

- *Example:* GCC/x86_64/Linux implementation:
  - In both cases (enabled/disabled optimization) a compiler generated program machine code without any error/warning message.
  - Relevant part of generated machine code for enabled optimizations:

```
mov   esi, -2147483648
mov   edi, OFFSET FLAT:_ZSt4cout
call  std::basic_ostream<char, std::char_traits<char> >::operator<<(int)
```

  - ⇒ The overflow occurred at compile time.
  - Relevant part of generated machine code for disabled optimizations:

```
mov   DWORD PTR [rbp-4], 2147483647
add   DWORD PTR [rbp-4], 1
mov   eax, DWORD PTR [rbp-4]
mov   esi, eax
mov   edi, OFFSET FLAT:_ZSt4cout
call  std::basic_ostream<char, std::char_traits<char> >::operator<<(int)
```

  - ⇒ The overflow occurs at runtime when add instruction is executed.

Undefined behavior (*cont.*)

```
int i = std::numeric_limits<int>::max();
i++;  // causes signed integer overflow => undefined behavior
```

- With this code, overflow happens always (unconditionally) and may happen both at compile time or at runtime.
- What if we change this code to?

```
std::cin >> i;
i++;  // causes overflow only if i equals max value of int
```

- Then, the overflow:
  - cannot happen at compile time (the calculation cannot be performed at compile time when the input value is not known);
  - happens only if 2147483647 is read from the standard input.
- *Conclusion:* The behavior of the program is now:
  - undefined only if 2147483647 is read from the standard input,
  - (well) defined otherwise (if user inputs 1, `i` will be incremented to 2).
- *Note:* This implies that, generally, undefined behavior cannot be detected by static code analysis (*more details later*).

Undefined behavior (*cont.*)

- The C++ standard does not define behavior for this program once the overflow occurs:

```
#include <iostream>
#include <limits>

int main() {
  int i = std::numeric_limits<int>::max();
  i++;  // causes signed integer overflow => undefined behavior
  std::cout << i;
}
```

- Yet, a C++ implementation generated a program binary executable file with some machine code:

```
mov   esi, -2147483648
mov   edi, OFFSET FLAT:_ZSt4cout
call  std::basic_ostream<char...
```

```
mov   DWORD PTR [rbp-4], 2147483647
add   DWORD PTR [rbp-4], 1
mov   eax, DWORD PTR [rbp-4]
mov   esi, eax
mov   edi, OFFSET FLAT:_ZSt4cout
call  std::basic_ostream<char...
```

- In both cases, execution renders same behavior:

```
$ ./a.out
-2147483648
```

Undefined behavior (*cont.*)

- Once some condition for undefined behavior is met, from the perspective of the C++ standard, **anything can happen** (*undefined* = *anything is allowed*).
- This “anything” includes also the possibility of the observed behavior…

ANARCHAOS

```
$ ./a.out
-2147483648
```

- …just it is not defined by the C++ standard.
- Instead, it is defined by a given implementation (in our case, by *GCC/x86_64/Linux*).
- *Implication:*
  - When the behavior is implementation-defined, there is **no portability**.
  - With another C++ implementation / on other system / with other built setup / …, the **observed behavior may be completely different**.

Undefined behavior (*cont.*)

- (Partial) summary:

1. When the **rules of the C++ standard are not violated**:
   - *C++ standard* defines behavior observable on the abstract machine.
   - *C++ implementation* must provide the same behavior observable on a particular computer system.
   - This must hold for all C++ implementations on all systems (*portability*).
2. When the **rules of the C++ standard are violated**:
   - *C++ standard* does not define any behavior on the abstract machine.
   - *C++ implementation* can do whatever it wants; typically, it provides some behavior.
   - This behavior is implementation-specific; generally, it is different for different implementation/systems (*no portability*).

- *Conclusion:*
  - To have correctly-running portable programs, one should avoid undefined behavior as much as possible.

Perils of undefined behavior

- When a condition for undefined behavior is met, the actually-observed behavior (defined by a given C++ implementation) may correspond with some "**expected behavior**".

```
int i = std::numeric_limits<int>::max();
i++;
```

```
$ ./a.out
-2147483648
```

- Is this result expected?
  - Mathematically, no — adding two positive numbers cannot give negative result.
  - However, this outcome might be expected under the assumption of *two's complement* binary representation of integer numbers.
  - With other representations (such as *one's complement*, where negative zero exists), the result might be different.
- "Expected behavior" is **subjective** — different people may have different expectations.
- Frequent argumentation that "*the code is correct since it behaves as I expect*" is very dangerous and wrong once UB is involved.

Perils of undefined behavior (*cont.*)

- In our example, the implementation-specific behavior was always the same (hardcoded in the program machine code):

```
mov   esi, -2147483648
mov   edi, OFFSET FLAT:_ZSt4cout
call  std::basic_ostream<char...
```

```
mov   DWORD PTR [rbp-4], 2147483647
add   DWORD PTR [rbp-4], 1
mov   eax, DWORD PTR [rbp-4]
mov   esi, eax
mov   edi, OFFSET FLAT:_ZSt4cout
call  std::basic_ostream<char...
```

```
$ ./a.out
-2147483648
```

- Once this machine code is generated, each execution of the program is guaranteed to have the same behavior.
- However, **generally, this does not hold**.
- *Example:*
  - Undefined behavior caused by *data race* (unsynchronized read-write access to the same memory location from multiple threads).
  - Each time data race occurs, it may have different unpredictable impact on observable program behavior.

Perils of undefined behavior (*cont.*)

- *Example:*

```
int i = 0;  // global variable shared by threads
void inc_i() { i++; }
```

- When function `inc_i` is executed N times by each of T threads, one might **expect** that the value of `i` would be incremented by $N \times T$ in total.
- This “expected” behavior is undefined:
  - According to the C++ standard, this is *data race*, which causes undefined behavior.
- *Note:* To get rid of data race / undefined behavior, `i` would need to have atomic data type, such as `std::atomic<int>`, instead of `int`.
- However, in practice, this expected behavior may be observed in most cases and, only occasionally, it may be different.

Perils of undefined behavior (*cont.*)

- Undefined behavior when the program mostly behaves as expected:
  - Extremely dangerous for production environments and mission-critical applications (*server applications, databases, banking, embedded systems in planes, cars, space crafts,*…).
- Program can pass all tests and run without problems for long time, until once…:
  - In best cases, occasional unexpected behavior causes program to crash.
  - In worst cases, it continues to run with incorrect state.
- Once a problem is identified, it might be very hard to find its cause:
  - We would need to replicate this unexpected behavior during testing / debugging.
  - Since it happens only occasionally, it may be almost impossible.

Detection undefined behavior

- Can undefined behavior be detected?
  - We have seen, that this is generally impossible at compile time (with static code analysis).
- Is it possible at runtime?
  - In theory, maybe.
  - **In practice, no.**
- Why?
  - Even if we could detect violation of C++ standard rules at runtime, it would **impose a large runtime and memory overhead into C++ programs**.
  - The purpose of programming in C++ is mostly **performance and efficiency**.
- $\Rightarrow$ Generally, in programming, we can either have safety or performance. It's not possible to have both at once.
- **C++ goes for performance.**

Detection undefined behavior (*cont.*)

- *Example:* (library) source file (*left*), translation unit (*right*):

```cpp
// deref_int.cpp:
int deref_int(int* ptr) {
  return *ptr;
}
```

```cpp
int deref_int(int* ptr) {
  return *ptr;
}
```

- According to the C++ standard, dereferencing of a pointer (`*ptr` here) is valid only if this pointer actually points to an existing object of a given type (`int` here).
- If this requirement is not satisfied, the behavior is undefined once the dereferencing takes place.
- ⇒ To detect undefined behavior with this code, a C++ implementation would need to **test the "validity" of a pointer**.
- *How?*

Detection undefined behavior (*cont.*)

- The most efficient representation of a pointer is just the **value of the address**.
- This representation allows generating the following optimal machine code (*GCC/x86_64/Linux*):

```
int deref_int(int* ptr) {
  return *ptr;
}
```

```
deref_int(int*):
  mov eax, DWORD PTR [rdi]
  ret
```

- A compiler has no idea what will be passed as a function argument.
- ⇒ Detection of UB would require some mechanism of how to find out whether a pointer actually points to an existing object.
- Is such a mechanism even possible? Likely not.
- Even if it was, it would have tremendous negative impact on performance:
  - When some object would be destroyed, all pointers that pointed to it would need to be informed about this destruction.
  - In the function, there would need to be a test for whether the pointer is valid or not.

Non-standard C++ extensions

- What happens in case of undefined behavior is mostly implementation-specific.
- Implementations can do anything they want and, typically, they do not care much about what happens:

```
int deref_int(int* ptr) {
  return *ptr;
}
```

```
deref_int(int*):
  mov eax, DWORD PTR [rdi]
  ret
```

- *Observation:*
  - GCC does not care whether the passed pointer actually points to an existing object of type `int`.
  - This is a function user (caller) responsibility to guarantee that.
  - Machine code simply reads a 32-bit value from the passed address.
  - In case of undefined behavior, if this address is accessible, the function will just return some value.
  - Otherwise, it will likely cause the running program (process) to crash (with something like *segfault*).

Non-standard C++ extensions (*cont.*)

- In some (rare) cases, C++ implementations explicitly specify what happens even when C++ standard rules are violated.
- These cases form non-standard C++ extensions.
- *Example:*

```
union U {
  int i;
  float f;
};
```

- *Union* is a *class-like* type.
- In contrast to *classes* and *structs*, union's non-static member variables **share the same storage**:

```
struct S {  // same with class
  int i;
  float f;
};
```

```
std::cout << sizeof(S) << sizeof(U);  // prints '84' (x86_64/Linux)
```

Non-standard C++ extensions (*cont.*)

- C++ standard rules:
  - Only one union's member variable is so-called "*active*", namely the one most recently written into.
  - Only the active member variable may be read.
  - ⇒ Reading the inactive member variable = undefined behavior.

```
union U {
  int i;
  float f;
};

int main() {
  U u;
  u.f = 1.0f;
  return u.i;  // undefined behavior according to C++ standards
}
```

- However, GCC explicitly allows this code and specifies a particular behavior for it:
  - "*The relevant bytes of the representation of the object are treated as an object of the type used for the access.*"
  - ⇒ `u.i` has the value of an object of type `int` that has the same binary representation as the `float` object with value `1.0f`.

Non-standard C++ extensions (*cont.*)

- *GCC/x86_64/Linux:*
  - Floating-point values are represented in IEEE 754 format.
  - `float` is a single-precision IEEE 754 floating-point data type.
  - Value 1 has binary representation 0x3f800000 in hex with this data type.

```
int main() {
  U u;
  u.f = 1.0f;
  return u.i;
}
```

```
main:
  mov eax, 0x3F800000
  ret
```

- The behavior of the resulting program is:
  - undefined by the C++ standard,
  - but guaranteed by the used GCC implementation.
- ⇒ This source code is generally not portable.

- *Similar case:*

```
int main() {
  float f = 1.0f;
  return *reinterpret_cast<int*>(&f);
}
```

```
main:
  mov eax, 0x3F800000
  ret
```

Object lifetime

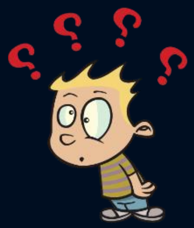
- **Object lifetime** — time interval during which the object exist at runtime (when the program is run).
- *Simplified rules:*
- Lifetime begins with object initialization, which constructs the object.
- In case of class types:
  - object initialization involves calling a class constructor,
  - object lifetime ends when its destructor is called.
- Before an object can be constructed/initialized, a storage must be allocated for it.
  - Storage = place where a binary representation of the object will be stored.
  - *Recall* — this storage must be properly aligned and sized.

Storage allocations

- How is object storage allocated?
- Two separate cases:

1. Object existence is known when the program is compiled:
   - For storage allocation for this object, a compiler is responsible.
   - This case is referred to as a ***"static allocation"***.
   - Meaning of *static* here = *resolvable-at-compile-time*.
   - *Effect* — allocation is hard-code in the program (it's machine code).
2. Object existence is not known until runtime:
   - Storage allocation must be *explicitly* initiated by a programmer.
   - This case is referred to as a ***"dynamic allocation"***.
   - Meaning of *dynamic* = *resolvable-not-until-runtime*.
   - *Effect* — allocation needs to be resolved at runtime by some external mechanisms (memory allocation functions).

Ambiguity of “static”



- “*Static*” vs “*static*” ambiguity:
  - Static allocations are not related to the `static` specifier/keyword:

```
void f() {
  int i = 1;
  static int j = 2;
  ...
}
```

- *Non-static* function-local variable `i`:
  - It’s lifetime is restricted to the function body.
  - ⇒ Each time `f` is called the lifetime of `i` starts (with its initialization) and ends (when the function is left).
- *Static* function-local variable `j`:
  - It’s lifetime basically covers the whole program run.
  - Typically, it starts when `f` is called first time (at the latest) and ends when the program is terminated.
- Storage for both `i` and `j` variables are statically allocated — allocation is “hard-coded” in the program at compile time.

Ambiguity of "static" (*cont.*)

- The word "*static*" can have many different meaning in C++.
- We have seen:
  - *Static entities* (functions, variables) with the meaning of being local to translation units with respect to linking (having *internal linkage*).
  - *Static class members* (functions, variables) with the meaning of belonging to the class itself instead of to its instances.
  - *Static allocations* with the meaning of being resolvable at compile time.
- The last meaning of the word "static" is used more generally.
- *Example* — static vs dynamic polymorphism:
  - *Polymorphism* = writing a unified code that can operate on objects of different types.
  - *Dynamic polymorphism* is provided by virtual member functions.
    - Function calls are resolved at runtime according to the actual (*dynamic*) object type.
  - *Static polymorphism* is provided by templates.
    - Templates are resolved at compile time (they do not exist at runtime).

Static allocations — *example*

- Translation unit (*left*) and generated machine code (*right*):

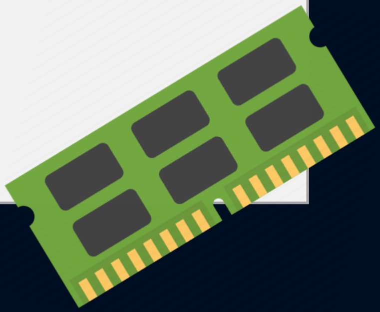
```
void g(int*, int*);

void f() {
  int i = 1;
  static int j = 2;
  g(&i, &j);
}
```

```
f():
  push  rax
  mov   dword ptr [rsp + 4], 1
  lea   rdi, [rsp + 4]
  mov   esi, offset f()::j
  call  g(int*, int*)
  pop   rax
  ret

f()::j:
  .long   2
```

- Storage for `i` and `j` variables is statically allocated:
  - Allocations in both cases are hard-coded in the program file / generated machine code (at compile time).
- *Note:* observable behavior or `f` involves pointers to `i` and `j`.
  - Function `f` passes these pointers as arguments to function `g`.
  - ⇒ A compiler is "forced" to allocate storage for `i` and `j` in memory.
  - According to ABI, pointers are passed through `rdi` and `rsi` registers.
  - On the contrary, pointers `&i` and `&j` are not stored in memory.

Static allocations — *example (cont.)*

```
void g(int*, int*);

void f() {
  int i = 1;
  static int j = 2;
  g(&i, &j);
}
```



- Where in memory are `i` and `j` variables stored?
- C++ standards do not specify this; they "*do not care*".
- C++ standards only prescribe so-called "*storage duration*".
  - `i` has "*automatic storage duration*" — storage is allocated when **the code block** is entered and deallocated when it is exited.
  - `j` has "*static storage duration*" — storage is allocated when **the program** is started and deallocated when it is exited.
- Storage in memory (address space) is specified by a given C++ implementation.
- Implementations need to conform to a system ABI.

Static allocations — *example (cont.)*

```
void g(int*, int*);

void f() {
  int i = 1;
  static int j = 2;
  g(&i, &j);
}
```

```
f():
  push  rax
  mov   dword ptr [rsp + 4], 1
  lea   rdi, [rsp + 4]
  mov   esi, offset f()::j
  call  g(int*, int*)
  pop   rax
  ret

f()::j:
  .long   2
```

- Used implementation: *Clang/x86_64/Linux*:
  - Storage for non-static function-local variable `i` is allocated on the stack.
    - Allocation is effectively performed by "reserving" stack space by lowering the stack pointer register `rsp` (this is what the push instruction does).
    - Deallocation is performed by raising the stack pointer back to the original value (by using the pop instruction).
  - Storage for static function-local variable `j` is allocated on the so-called program *data segment*.
    - Memory for data segment is allocated when the program is executed and deallocated when it is exited.

Allocation and initialization

- Once there is a storage allocated for some object, its lifetime begins with its initialization.
- The object lifetime ends with its destruction, which takes place before its storage is deallocated.
- There are many forms of initialization, which, generally, have different effect for objects of different types.
- "Rough" distinction of two different cases.

1. *Class types:*
   - Initialization involves a constructor call.
   - Destruction involves a destructor call.
2. *Non-class types* (such as "*fundamental*" language-intrinsic types):
   - Initialization requires setting a binary representation to a desired value (determined by the initialization expression).
   - Destruction does not require any action.

Allocation and initialization (*cont.*)

```
void g(int*, int*);

void f() {
  int i = 1;
  static int j = 2;
  g(&i, &j);
}
```

```
f():
  push  rax
  mov   dword ptr [rsp + 4], 1
  lea   rdi, [rsp + 4]
  mov   esi, offset f()::j
  call  g(int*, int*)
  pop   rax
  ret

f()::j:
  .long   2
```

- In our case, `int` is a fundamental non-class type.
- `i` is initialized by `= 1` expression:
  - It is initialized each time f is executed after its storage has been allocated.
  - Initialization = setting its binary representation to “have” value 1.
  - Its lifetime ends when the function is left (*no action required*).
- `j` is initialized by `= 2` expression:
  - Initialization = setting its binary representation to “have” value 2.
  - This value is hard-code in the program file and loaded to the data segment when the program is executed.

Allocation and initialization (*cont.*)

- For a *class type*:
  - allocation/deallocation works the very same way,
  - initialization involves constructor, destruction involves destructor.

```
struct X {
  X(int);
  ~X();
};

void g(X*) noexcept;

void f() {
  X x(1);
  g(&x);
}
```

```
f():
  push  rbx
  sub   rsp, 16
  lea   rbx, [rsp + 8]   ; address of the storage
  mov   rdi, rbx         ; passed by rdi to constructor
  mov   esi, 1           ; initialization argument passed by rsi
  call  X::X(int)
  mov   rdi, rbx         ; address of x passed to g by rdi
  call  g(X*)
  mov   rdi, rbx         ; to destructor as well
  call  X::~X()
  add   rsp, 16
  pop   rbx
  ret
```

- *Notes:*
  - A static function-local variable would be initialized only the first time f is called and destructed when the program exits.
  - Destructor for x would need to be called even if g threw an exception.
    - ⇒ In case of *not*-noexcept g, machine code would be more complicated.

Static vs dynamic allocations

- Static allocations — always take place when the program is run $\Rightarrow$ can be hard-coded in the program machine code.
- In some cases, allocations cannot be resolved until runtime.
- *Examples*:
  - Conditional allocations dependent on some runtime-evaluable condition.
  - Allocations of storage for multiple objects, where their number is not known until runtime.
  - *Etc…*
- Such allocations need to be resolved at runtime, i.e., dynamically.
- *Example:*

```
void g(int*, int*);

void f(bool b, long n) {
  int* pi = b ? new int(1) : nullptr;
  int* pa = new int[n]{};
  g(pi, pa);
}
```

Dynamic allocations

- "Dynamically-allocated objects" — the very same concept:
  1) First, a storage for an object needs to be allocated.
  2) Second, that object is in this storage initialized.
- Ad 1) How is a storage allocated in such a case?
  - Again, C++ standard do not specify this.
  - They only define so-called *allocation functions*.
  - These functions are called `operator new` and return a block of "*uninitialized*" memory of a desired size of bytes.
    - *Uninitialized* = there is no object in this memory yet.
  - How do these functions work is implementation-defined.
- Typical *implementations*:
  - Internally allocate memory on the heap (a system-provided mechanism for dynamic memory allocations).
  - ⇒ `operator new` internally calls some heap allocation function, such as `malloc`.

`operator new` vs `new` *expression*

- Allocation function `operator new` only provides storage for dynamically-allocated objects.
- Then, objects need to be in that storage constructed / initialized.
- Usually, both these steps are required (by programmers) to be performed together.
- $\Rightarrow$ They are "bound" in a mechanism called `new` *expression*.
- new expression is the most common form of new.
- It is responsible for both:
  1) allocation storage for an object,
  2) its initialization in this storage.
- For allocation, it internally uses `operator new`.

operator new vs new *expression* (*cont.*)

- Example with a fundamental type…

```
void g(int*);

void f() {
  int* pi = new int(1);  // new expression
  g(pi);
}
```

```
f():
  push  rax
  mov   edi, 4
  call  operator new(unsigned long)
  mov   dword ptr [rax], 1
  mov   rdi, rax
  pop   rax
  jmp   g(int*)
```

- and with a class type:

```
void g(X*);

void f() {
  X* px = new X(1);  // new expression
  g(px);
}
```

```
f():
  ...
  mov   edi, 1
  call  operator new(unsigned long)
  mov   rbx, rax
  mov   rdi, rax
  mov   esi, 1
  call  X::X(int)
  ...
```

- *Observation:*
  - new expression internally allocates storage with operator new,
  - and then initializes the object in this storage.

Dynamic allocations (*cont.*)

- With *statically-allocated objects*:
  - objects are automatically destructed,
  - storage is automatically deallocated.
- The same does not hold for *dynamically-allocated objects*:
  - object is not destructed (no destructor call),
  - storage is not deallocated (no deallocation function call).
- Objects "manually" allocated and initialized with expression `new` need to be "manually" destructed and deallocated with its counterpart — expression `delete`.

```
struct X {
  X(int);
  ~X();
};

void g(X*)
  noexcept;

void f() {
  X x(1);
  g(&x);
}
```

```
f():
  push  rbx
  sub   rsp, 16
  lea   rbx, [rsp + 8]
  mov   rdi, rbx
  mov   esi, 1
  call  X::X(int)
  mov   rdi, rbx
  call  g(X*)
  mov   rdi, rbx
  call  X::~X()
  add   rsp, 16
  pop   rbx
  ret
```

```
void g(X*);

void f() {
  X* px
    = new X(1);
  g(px);
}
```

```
f():
  ...
  call  X::X(int)
  mov   rdi, rbx
  add   rsp, 8
  pop   rbx
  pop   r14
  jmp   g(X*)
```

`operator delete` vs `delete` expression

- new expression:
  1) allocates storage for an object,
  2) initializes object in this storage.
- `delete` expression:
  a) destructs object,
  b) deallocates its storage.
- `operator new`:
  - C++ allocation function internally called by new expression in ad 1).
  - Typically implemented by calling some heap allocation function (such as `malloc`).
- `operator delete`:
  - C++ deallocation function internally called by `delete` expression in ad b).
  - Typically implemented by calling a heap deallocation function (`free`).

operator delete vs delete expr. (cont.)

```
void g(X*);

void f() {
  X* px = new X(1);
  g(px);
  delete px;  // delete expression
}
```

```
f():
  ...
  mov   edi, 1
  call  operator new(unsigned long)
  mov   rbx, rax
  mov   rdi, rax
  mov   esi, 1
  call  X::X(int)
  mov   rdi, rbx
  call  g(X*)
  mov   rdi, rbx
  call  X::~X()
  mov   rdi, rbx
  add   rsp, 8
  pop   rbx
  pop   r14
  jmp   operator delete(void*)
```

- *Observation* — `delete` expression:
  - destructs object by calling its destructor,
  - deallocates its storage by calling C++ deallocation function `operator delete`.
- *Notes:*
  - (De)allocation functions `operator new` and `operator delete` are provided by the C++ standard library.
  - ⇒ In practice, they are provided by library implementations.
  - *Example* — `operator new` in *libc*++ invokes `malloc`: https://github.com/llvm/llvm-project/blob/main/libcxx/src/new.cpp.