Lecture 4
Secure Software Engineering Group
Philipp Dominik Schubert
CONTENTS

1. Operator overloading
2. Memory
3. Dynamic memory allocation
4. Real-world examples
   1. Copy constructor / assign
   2. Move constructor / assign
5. Notes on dynamic memory allocation
Operators

- =  
- +  
- -  
- *  
- /  
- ==  
- []  
- <  
- >  
- <=  
- And lots of others
Operator overloading

- Operators have a certain meaning
  
  ```
  int sum = 4 + 5;
  ```
  
  - + is the mathematical plus that sums up numbers

- But operators can be overloaded
  
  - An overloaded operator may have more than one meaning
  - Meaning depends on context (type(s) it is applied to)
  - Overloading operators is a powerful instrument
    - It allows you to write code / implement algorithms exactly like shown in maths textbooks

- Have you ever used an overloaded operator?
  
  - Yes, remember the character output stream `std::cout` and `std::string`
Operator overloading

- Rather natural use of + for strings
  
  ```cpp
  std::string a = "AAAA";
  std::string b = "BBB";
  std::string result = a + b;
  ```

- Overloaded operators should be thoroughly designed
  - Should behave as expected
  - Do not implement
    
    ```cpp
    std::string a = "AAAA";
    std::string b = "BBB";
    std::string result = a + b;
    to delete the contents of a and b and fill result with "I am a very funny guy."
    ```
Operator overloading

- What operators can be overloaded?
  - If you design a data type only overload operators …
    - that are useful
    - whose meaning is clear
  - Do not overload all operators or as much as possible
  - Custom operators?
    - E.g. `operator**`
      - Python's power operator
      - Arbitrary defined operators would be possible
      - But are not intuitive
Simple example: operator<<

- Definition inside type definition

```cpp
#include <iostream>

struct Type {
    Type(int i, double d) : i(i), d(d) {}
    int i;
    double d;
    friend std::ostream&
        operator<<(std::ostream& os, const Type &t) {
            return os << t.i << " and " << t.d;
        }
};

int main() {
    Type t(1, 2.222);
    std::cout << t << '\n';
    return 0;
}
```

- Definition outside type definition

```cpp
#include <iostream>

struct Type {
    Type(int i, double d) : i(i), d(d) {}
    int i;
    double d;
};

std::ostream& operator<<(std::ostream& os, const Type &t) {
    return os << t.i << " and " << t.d;
}

int main() {
    Type t(1, 2.222);
    std::cout << t << '\n';
    return 0;
}
```
Operator overloading

- Remember our Vec3 type

```cpp
class Vec3 {
private:
    double x;
    double y;
    double z;
public:
    Vec3() : x(0), y(0), z(0) { }
    Vec3(double x, double y, double z) : x(x), y(y), z(z) { }
    size_t size() { return 3; }
    double euclidean_length() { return sqrt(x*x+y*y+z*z); }
friend std::ostream& operator<<(std::ostream& os, const Vec3& v) {,
    return os << v.x << 'n' << v.y << 'n' << v.z;
};
```

- Why is `operator<<` declared as friend?
  - Obviously it is not a member function
  - It receives a Vec3 as a parameter
  - If a function / operator is declared as friend it can access a type’s private data members!

```cpp
Vec3 v1(4, 5, 6);
std::cout << v1 << 'n';
```
### Operator overloading

- Let’s overload `operator+` and `operator[]` for convenience.

```cpp
class Vec3 {
private:
    double x;
    double y;
    double z;
public:
    Vec3() : x(0), y(0), z(0) {}
    Vec3(double x, double y, double z) : x(x), y(y), z(z) {}
    size_t size() { return 3; }
    double euclidean_length() { return sqrt(x*x+y*y+z*z); }
    double& operator[] (size_t idx) { switch (idx) {
        case 0: return x; break;
        case 1: return y; break;
        case 2: return z; break;
        default: /* handle error */;
    }
}
friend Vec3 operator+ (Vec3 lhs, const Vec3& rhs){
    for (size_t i = 0; i < lhs.size(); ++i)
        lhs[i] += rhs[i];
    return lhs;
}
friend std::ostream& operator<< (std::ostream& os,
    const Vec3& v) {
    return os << v.x << " " << v.y << " " << v.z;
};
```
Operator overloading

- Now we can use

```cpp
int main() {
    Vec3 v1(1, 2, 3);
    Vec3 v2(4, 5, 6);
    v2[1] = 50;
    Vec3 v3 = v1 + v2;
    std::cout << v3 << '
';
    return 0;
}
```

- Output
  - 5 52 9
Operator overloading

- You are now able to write these yourself

friend Vec3 operator- (Vec3 lhs, const Vec3& rhs);
friend double operator* (const Vec3& lhs, const Vec3& rhs);
friend Vec3 operator* (Vec3 lhs, double rhs);
friend Vec3 operator% (Vec3 lhs, const Vec3& rhs);

// caution, may not be intuitive in this example!
friend bool operator< (const Vec3& lhs, const Vec3& rhs);
friend bool operator> (const Vec3& lhs, const Vec3& rhs);
friend bool operator<=(const Vec3& lhs, const Vec3& rhs);
friend bool operator>=(const Vec3& lhs, const Vec3& rhs);
friend bool operator==(const Vec3& lhs, const Vec3& rhs);
friend bool operator!=(const Vec3& lhs, const Vec3& rhs);

- You only have to implement operator< and operator== for comparisons
- Others can be expressed using < and ==

- Caution: It is not a smart idea to check two doubles for equality!
How about custom operators?

- Again, not possible in the C++ language
- **But**, it is possible using some ugly hack
  - We will do a fun lecture later on 😊
    - I will show you some of those things
Before we continue on dynamic memory: this—a special pointer

- **this** pointer

“The keyword this is a **prvalue expression** whose value is the address of the **implicit object parameter** (object on which the non-static member function is being called). It can appear in the following contexts:

1. Within the body of any non-static **member function**, including **member initializer list**
2. within the **declaration** of a non-static member function anywhere after the (optional) cv-qualifier sequence, including **dynamic exception specification**(deprecated), **noexcept specification**(C++11), and the trailing return type(since C++11)
3. within **default member initializer** (since C++11)”

- **this** lets you access the address of the object on which a (non-static) member function is being called on
  → address of the receiver object
this—a special pointer: an example

```cpp
#include <iostream>

struct X {
  void print_my_address() {
    std::cout << "the address of this object is: " << this << '\n';
  }
};

int main() {
  X a;
  a.print_my_address();
  X b;
  b.print_my_address();
  return 0;
}
```
Memory layout

- Memory layout in Linux systems (and C/C++ programs)
  - Kernel
    - Contains command-line and environment variables (OS data)
  - Stack
    - Contains function parameters and functions return address
    - Local variables
  - Heap
    - Allows allocation of huge chunks of memory
  - Data
    - Contains initialized and uninitialized (global) variables
  - Text
    - Read-only, contains program text (machine instructions)
- Memory is just a linear piece of addressable locations
- High addresses
- Low addresses
Memory layout

- Please consider this website
  http://www.geeksforgeeks.org/memory-layout-of-c-program/
- This page explains in detail why we have such a memory layout
- I highly encourage you to read that article
Why dynamic memory allocation?

- Consider local (stack) memory (of fixed size)
  
  ```c
  int buffer[10];
  ```

- Problem
  - How to store more than 10 elements?
  - What if you do not know the number of elements up-front?
  - How do you store a million elements?
    - Stack size is usually limited by the operating system
      - Why? → Well, think about deallocation
    - What if data should be used in more than one scope?
  
- Dynamically allocated heap memory solves these problems
  - Simply take (allocate) what you need
  - Allocate chunks of memory as large as you like (size of RAM chips)
Dynamic memory allocation

- Allocate what your RAM chips provide
  - Simply allocate what you need
  - But caution
    - You have to do the clean-up yourself
      - No garbage collector (unlike Java)!
    - Free memory after usage
      - Memory is yours until you explicitly free it:
        - There is no out of scope!
        - Do not lose the memory handle!
- The keyword for allocation is new
- The keyword for deallocation is delete
Dynamic memory allocation

- Remember pointers
  ```
  int i = 42; // integer variable i
  int *i_ptr = &i; // pointer i_ptr points to i
  ```
- Pointers will now become really useful
  - They are inevitable even
  - Operator `new` allocates the amount of memory you need
  - But `new` cannot provide a name for the allocated memory
    - It simply returns a pointer/address to “your” memory
  - Caution
    - Do not lose size information!
      - Otherwise you risk undefined reads and writes!
    - Do not forget to delete the memory and do not delete twice!
      - Otherwise you leak memory or program has undefined behavior/program crash!
More on undefined behavior

- Caution the following talk may causes nightmares
- “Undefined Behavior is awesome!”, Piotr Padlewski
  - https://www.youtube.com/watch?v=ehyHyAla5so
Allocate and delete memory

- The keyword for allocation is `new`
  - In fact it is an operator (that can even be overloaded)
  - Standard signatures are
    ```cpp
    void* operator new ( std::size_t count ); // for objects
    void* operator new[]( std::size_t count ); // for arrays
    ```
  - Notice `new` is returning a `void` pointer
  - If memory cannot be allocated `new` throws an `std::bad_alloc` exception (next lecture)
- The keyword for deallocation is `delete`
  - In fact it is an operator (that can even be overloaded)
  - Standard signatures are
    ```cpp
    void operator delete ( void* ptr ); // for objects
    void operator delete[]( void* ptr ); // for arrays
    ```
  - Every `new` needs a `delete` ⇒ otherwise your program has a leak
Allocate and delete memory

void* operator new ( std::size_t count ); // for objects
void* operator new[]( std::size_t count ); // for arrays
void operator delete ( void* ptr ); // for objects
void operator delete[]( void* ptr ); // for arrays

- Why is there a distinction between objects and arrays?
  - Just a syntactic oddity
  - Has no use at all
  - Even worse
    - Do not confuse between them
    - Do not allocate with `new` and free with `delete[]` (and vice versa)
```cpp
#include <iostream>

int main() {
  int *i = new int(13);
  *i = 42;
  std::cout << *i << "\n";
  delete i;
  delete i;
  return 0;
}
```
Valgrind was developed by Julian Seward
- British compiler construction specialist

Valgrind is a tool-suite that allows for detection of memory errors
- It runs the program under analysis multiple times
- It stops the program many times during its execution
- In those breaks it analyzes registers, stack, … and collects all these information
- With help of these information it can determine whether a program has some memory issues
- Caution: the program under analysis is executed ~100 times slower than usually
- Analyzing big projects needs time
- It is still worth while

There are not many ways for detecting memory issues
- Double delete ➔ program crashes
- Missing delete ➔ program leaks ➔ consumes more and more memory, until crash
Clang’s AddressSanitizer [https://clang.llvm.org/docs/AddressSanitizer.html](https://clang.llvm.org/docs/AddressSanitizer.html)

- Fast memory error detector
- Uses compiler instrumentation and a run-time library (slowdown ~ 2x)
- Can detect
  - Out-of-bound accesses to heap, stack, and globals
  - Use-after-free
  - Use-after-return
  - User-after-scope
  - Double-free, invalid free
  - Memory leaks (experimental)
- Compile and link program to be instrumented

```bash
$ clang++ -Wall -Wextra -std=c++17 -O1 -g -fsanitize=address -fno-omit-frame-pointer program.cpp -o program
```
Detecting leaks

```cpp
#include <iostream>

int main() {
    int *i = new int(13);
    *i = 42;
    std::cout << *i << 'n';
    delete i;
    return 0;
}
```

```
philipp@pdschbrt:~/ownCloud/cppp/tmp$ valgrind --leak-check=full --track-origins=yes ./double-free
==30151== Memcheck, a memory error detector
==30151== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==30151== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==30151== Command: ./double-free
==30151== 42
==30151== HEAP SUMMARY:
==30151== in use at exit: 72,704 bytes in 1 blocks
==30151== total heap usage: 3 allocs, 2 frees, 73,732 bytes allocated
==30151== LEAK SUMMARY:
==30151== definitely lost: 0 bytes in 0 blocks
==30151== indirectly lost: 0 bytes in 0 blocks
==30151== possibly lost: 0 bytes in 0 blocks
==30151== still reachable: 72,704 bytes in 1 blocks
==30151== suppressed: 0 bytes in 0 blocks
==30151== Reachable blocks (those to which a pointer was found) are not shown.
==30151== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==30151== For counts of detected and suppressed errors, rerun with: -v
==30151== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
philipp@pdschbrt:~/ownCloud/cppp/tmp$
```
Detecting leaks

```cpp
#include <iostream>

int main() {
    int *i = new int(13);
    *i = 42;
    std::cout << *i << '\n';
    return 0;
}
```

```sh
$ clang++ -Wall -Wextra -std=c++17 -g no-free.cpp -o no-free
```

```n
philipp@pdschbrt:/ownCloud/cppp/tmp$ ./double-free
==2979== Memcheck, a memory error detector
==2979== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==2979== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==2979== Command: ./double-free
==2979== 42
==2979== HEAP SUMMARY:
==2979== in use at exit: 72,708 bytes in 2 blocks
==2979== total heap usage: 3 allocs, 1 frees, 73,732 bytes allocated
==2979== 4 bytes in 1 blocks are definitely lost in loss record 1 of 2
==2979== at 0x4C2E0F: operator new(unsigned long) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==2979== by 0x40081B: main (double-free.cpp:4)
==2979== LEAK SUMMARY:
==2979== definitely lost: 4 bytes in 1 blocks
==2979== indirectly lost: 0 bytes in 0 blocks
==2979== possibly lost: 0 bytes in 0 blocks
==2979== still reachable: 72,784 bytes in 1 blocks
==2979== suppressed: 0 bytes in 0 blocks
==2979== Reachable blocks (those to which a pointer was found) are not shown.
==2979== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==2979== For counts of detected and suppressed errors, rerun with: -v
==2979== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)

==29791==ERROR: LeakSanitizer: detected memory leak
==29791==ERROR: AddressSanitizer: 4 byte(s) in 1 object(s) allocated from:
  #0 0x4f0718 in operator new(unsigned long) /home/philipp/GIT-Repos/llvm-project-11/compiler-rt/lib/asan/asan_new_dele
e.cpp:99
  #1 0x4f3f18 in main /home/philipp/ownCloud/cppp/tmp/double-free.cpp:4:12
  #2 0x7f29a9c883f in __libc_start_main /build/glibc-6ev40/glibc-2.23/csu/../csu/libc-start.c:291

SUMMARY: AddressSanitizer: 4 byte(s) leaked in 1 allocation(s).

philipp@pdschbrt:/ownCloud/cppp/tmp$
```

```sh
$ clang++ -Wall -Wextra -std=c++17 -O1 -g -fsanitize=address -fno-omit-frame-pointer no-free.cpp -o no-free
```
Dynamically allocated arrays

```c
int main() {
    int *array = new int[10];
    for (int i = 0; i < 10; ++i) {
        array[i] = 13;
        // this is pointer arithmetic, same meaning as `array[i]`
        *(array+i) += 2;
    }
    // do useful things with array
    delete[] array; // we have to use the array delete
    return 0;
}
```

- Problem with dynamic arrays
  - Programmers have to store size information themselves
    - High risk for index out-of-bounds
      - Undefined behavior or segmentation fault
Dynamically allocated 2D arrays (matrices)

```c
int main() {
    int rows = 3;
    int cols = 4;
    int **matrix = new int*[rows];
    for (int row = 0; row < rows; ++row)
        matrix[row] = new int[cols];
    for (int row = 0; row < rows; ++row)
        for (int column = 0; column < cols; ++column)
            matrix[row][column] = 42;
    // do useful stuff with matrix
    for (int row = 0; row < rows; ++row)
        delete[] matrix[row];
    delete[] matrix;
    matrix = nullptr;
    return 0;
}
```
Allocate a user-defined type

```cpp
struct Vertex {
    Vertex() : x(0), y(0), z(0) { }
    Vertex(double x, double y, double z) : x(x), y(y), z(z) { }
    double x, y, z;
    friend std::ostream& operator<<(std::ostream& os, const Vertex& v) {
        return os << v.x << " " << v.y << " " << v.z;
    }
};

int main() {
    Vertex *v1 = new Vertex;
    Vertex *v2 = new Vertex(1, 2, 3);
    Vertex v3(3, 2, 1);
    v2->x = 42; // -> shorthand for(*v2).x (dereference and access data)
    std::cout << *v2 << '\n';
    delete v1; v1 = nullptr;
    delete v2; v2 = nullptr;
    return 0; }
```
Copy & move using dynamic memory
Copy constructor & copy assign gone wrong

```cpp
#include <iostream>

struct DynInt {
    DynInt(int i) : i_ptr(new int(i)) {}
    ~DynInt() { delete i_ptr; } // we have to clean up
    DynInt(const DynInt &di) = default;
    DynInt& operator=(const DynInt &di) = default;
    friend std::ostream& operator<< (std::ostream& os, const DynInt &di) {
        return os << *di.i_ptr;
    }
    int *i_ptr;
};

int main() {
    DynInt di1(42);
    // ups! we copy the pointer i_ptr, but not what it points-to!
    DynInt di2 = di1;
    *di2.i_ptr = 100;
    std::cout << di1 << '\n';
    std::cout << di2 << '\n';
    // we call dtor for di1 and di2 ⇒ we call dtor twice for the same heap object
    return 0;
}
```
Copy constructor & copy assign gone wrong

```cpp
#include <iostream>

struct DynInt {
    DynInt(int i) : i_ptr(new int(i)) {}  // we have to clean up
    ~DynInt() { delete i_ptr; }

    DynInt(const DynInt& di) = default;
    DynInt& operator=(const DynInt& di) = default;

    friend std::ostream& operator<<(std::ostream& os, const DynInt& di) {
        return os << *di.i_ptr;
    }

    int* i_ptr;
};

int main() {
    DynInt di1(42);
    // ups! we copy the pointer i_ptr, but not what it points to!
    DynInt di2 = di1;
    *di2.i_ptr = 100;
    std::cout << di1 << '
';
    std::cout << di2 << '
';
    // we call dtor for di1 and di2  we call dtor twice for the same heap object
    return 0;
}
```

**Error in `.dyn-int`: double free or corruption (fasttop): 0x0000000001b9dc20 **

```
philipp@pdschbprt:/ownCloud/cppp/tmp$ clang++ -std=c++17 -Wall -Wextra dyn-int.cpp -o dyn-int
philipp@pdschbprt:/ownCloud/cppp/tmp$ ./dyn-int
100
100
*** Error in `.dyn-int': double free or corruption (fasttop): 0x0000000001b9dc20 ***
```

© Heinz Nixdorf Institut / Fraunhofer IEM
Copy constructor & copy assign: How to fix it?

```cpp
#include <iostream>

struct DynInt {
    DynInt(int i) : i_ptr(new int(i)) {}
    ~DynInt() { delete i_ptr; }
    DynInt(const DynInt &di) : i_ptr(new int(*di.i_ptr)) {} // create a deep copy!
    DynInt &operator=(const DynInt &di) {
        if (this != &di) { *i_ptr = *di.i_ptr; }
        return *this;
    }

    friend std::ostream &operator<<(std::ostream &os, const DynInt &di) {
        return os << *di.i_ptr;
    }

    int *i_ptr;
};

int main() {
    DynInt di1(42);
    DynInt di2 = di1; // call copy constructor
    *di2.i_ptr = 100;
    std::cout << di1 << '\n';
    std::cout << di2 << '\n';
    return 0;
}
```

© Heinz Nixdorf Institut / Fraunhofer IEM
Move constructor & move assign gone wrong

```cpp
#include <iostream>

struct DynInt {
    DynInt(int i) : i_ptr(new int(i)) {}
    ~DynInt() { delete i_ptr; }
    DynInt(const DynInt &di) : i_ptr(new int(*di.i_ptr)) {} // create a deep copy!
    DynInt &operator=(const DynInt &di) {
        if (this != &di) { *i_ptr = *di.i_ptr; }
        return *this;
    }
    DynInt(DynInt &&di) = default;
    DynInt& operator=(DynInt &&di) = default;
    friend std::ostream &operator<<(std::ostream &os, const DynInt &di) {
        return os << *di.i_ptr;
    }
    int *i_ptr;
};

int main() {
    DynInt di1(42);
    DynInt di2 = std::move(di1); // call move constructor
    *di2.i_ptr = 100;
    std::cout << di2 << '\n';
    return 0;
}
```
move constructor & move assign gone wrong

```
philipp@pdschbrt:~/ownCloud/cppp/tmp$ clang++ -std=c++17 -Wall -Wextra dyn-int.cpp -o dyn-int
philipp@pdschbrt:~/ownCloud/cppp/tmp$ ./dyn-int
100
*** Error in `./dyn-int': double free or corruption (fasttop): 0x00000000010c5c20 ***
= = = = = = Backtrace: = = = = = =
/lib/x86_64-linux-gnu/libc.so.6(+0x777f5) [0x7efc1c0757f5]
/lib/x86_64-linux-gnu/libc.so.6(+0x8038a) [0x7efc1c07e38a]
/lib/x86_64-linux-gnu/libc.so.6(cfree+0x4c) [0x7efc1c08258c]
./dyn-int[0x400a9d]
./dyn-int[0x4009c1]
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xf0) [0x7efc1c01e840]
./dyn-int[0x400889]
= = = = = = Memory map: = = = = = =
00400000-00401000 r-xp 00000000 08:02 29923143 /home/philipp/ownCloud/cppp/tmp/dyn-int
00600000-00601000 r--p 00000000 08:02 29923143 /home/philipp/ownCloud/cppp/tmp/dyn-int
00601000-00602000 rw-p 00001000 08:02 29923143 /home/philipp/ownCloud/cppp/tmp/dyn-int
010b4000-010e6000 rw-p 00000000 00:00 0 [heap]
7efc14000000-7efc14021000 rw-p 00000000 00:00 0
7efc14021000-7efc18000000 --p 00000000 00:00 0
7efc1bffe000-7efc1c1be000 r-xp 00000000 08:02 21627152 /lib/x86_64-linux-gnu/libc-2.23.so
7efc1c1be000-7efc1c3be000 --p 001c0000 08:02 21627152 /lib/x86_64-linux-gnu/libc-2.23.so
7efc1c3be000-7efc1c3c2000 r--p 001c0000 08:02 21627152 /lib/x86_64-linux-gnu/libc-2.23.so
7efc1c3c2000-7efc1c3c4000 rw-p 001c4000 08:02 21627152 /lib/x86_64-linux-gnu/libc-2.23.so
```
Move constructor & move assign: How to fix it?

```cpp
#include <iostream>

struct DynInt {
    DynInt(int i) : i_ptr(new int(i)) {}
    ~DynInt() { delete i_ptr; }
    DynInt(const DynInt &di) : i_ptr(new int(*di.i_ptr)) {}
    DynInt &operator=(const DynInt &di) {
        if (this != &di) { *i_ptr = *di.i_ptr; }
        return *this;
    }
    DynInt(DynInt &&di) : i_ptr(di.i_ptr) { di.i_ptr = nullptr; }
    DynInt &operator=(DynInt &&di) {
        if (this != &di) {
            delete i_ptr;
            i_ptr = di.i_ptr;
            di.i_ptr = nullptr;
        }
        return *this;
    }

    friend std::ostream &operator<<(std::ostream &os, const DynInt &di) { return os << *di.i_ptr; }

    int *i_ptr;
};

int main() {
    DynInt di1(42);
    DynInt di2 = std::move(di1); // call move constructor
    *di2.i_ptr = 100;
    std::cout << di2 << '\n';
    return 0;
}
```
Relax now

- You know how to handle special member functions in context of dynamic memory allocation
  - It will not get anymore complicated than that
A note on special member functions using built-in and STL types

- STL types provide many useful constructors
- STL types know how they have to be …
  - destructed
  - copied
  - copied assigned
  - moved
  - move assigned

- If your are dealing with types containing only primitive (built-in) or STL data members = default is fine
- If you are dealing with dynamic memory yourself, you now know how to deal with special member functions such as copy, move, and the assignment operators
Notes on dynamic memory allocation

- Raw `new` and `delete` need to be used rarely
  - “Too” error prone
  - Usually no need for raw `new` and `delete`
  - There are exceptions of course

- If you need to allocate an object
  - Just do so using stack memory
    - and return by value (RVO and `std::move()` take care of performance)
    - `std::move()` the variable between scopes, if you want it to live longer then its scope
  - If the object is too large to be stored on the stack, then use dynamic memory allocation

```cpp
#include <iostream>
#include <utility>
using namespace std;

int main() {
    int outer_scope;
    {
        int inner_scope = 42;
        outer_scope = std::move(inner_scope);
    }
    std::cout << outer_scope << '
';
    return 0;
}
```
Notes on dynamic memory allocation

- If you need to allocate a fixed-size array of objects
  - Just do so using stack memory
    - `int data[10];`
    - `std::array<int, 10> more_data; // use: #include <array>
      - Safe wrapper for fixed-size stack allocated arrays
      - Carries size information

- If you need to dynamically allocate an array of objects
  - Use `std::vector`
    - It was created for this purpose
    - Safe wrapper for dynamically allocated arrays
    - Carries size information
Notes on dynamic memory allocation

- If you need raw `new` and `delete` nevertheless
  - Try to use smart pointers rather than raw pointers
    - Smart pointers take care of deallocation → `delete`
    - Smart pointers do the clean-up themselves
      - You cannot leak anymore
      - “The poor man’s garbage collector”
    - Implement reference counting
  - We will see smart pointers in one of the next lectures
Recap

- Operator overloading
- A program's memory layout
- Dynamic memory allocation
- Dynamic memory allocation for arrays
- Dynamic memory allocation for objects
- Valgrind and Clang’s Sanitizers
- Copy constructor
- Move constructor
- Notes on dynamic memory allocation
Thank you for your attention
Questions?