C++ Programming

Lecture 4
Software Engineering Group

Philipp D. Schubert
1. Operator overloading
2. Memory
3. Dynamic memory allocation
4. Copy constructor / assign (a real-world example)
5. Move constructor / assign (a real-world example)
6. Notes on dynamic memory allocation
Operators

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

- Operators have a certain meaning
  
  ```cpp
  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 text books

- 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
  
  ```
  string a = "AAAA";
  string b = "BBB";
  string result = a + b;
  ```

- Overloaded operators should be thoroughly designed
  
  - Should behave as expected
  
  - Do not implement
    
    ```
    string a = "AAAA";
    string b = "BBB";
    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?
  - Please refer to http://en.cppreference.com/w/cpp/language/operators
  - If you write 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

Syntax
Overloaded operators are functions with special function names:

```
operator op (1)
operator type (2)
operator new operator new [ ] (2)
operator delete operator delete [ ] (4)
operator "" "" suffix-identifier (5) (since C++11)
```

`op` any of the following 38 operators: `+` `-` `/` `%` `&` `&=` `|` `|=` `^` `^=` `<<` `>>` `<<=` `>>=` `*` `*=` `/=` `%=` `|=` `^=` `&=` `|&` `&|` `|&=` `&|=` `&|&` `&|&=` `&|&=` `&|=` `&|==` `&|= &` `&|= |`

1) overloaded operator;
2) user-defined conversion function;
3) allocation function;
4) deallocation function;
5) user-defined literal.

Overloaded operators
When an operator appears in an expression, and at least one of its operands has a class type or an enumeration type, then overload resolution is used to determine the user-defined function to be called among all the functions whose signatures match the following:

<table>
<thead>
<tr>
<th>Expression</th>
<th>As member function</th>
<th>As non-member function</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>@a</td>
<td>(a).operator( )</td>
<td>operator(a)</td>
<td>std::cin calls std::cin.operator( )</td>
</tr>
<tr>
<td>a@a</td>
<td>(a).operator( b )</td>
<td>operator( b, a)</td>
<td>std::cout &lt;&lt; 42 calls std::cout.operator(42)</td>
</tr>
<tr>
<td>a&gt;&gt;b</td>
<td>(a).operator( b )</td>
<td>operator( a, b)</td>
<td>std::cout &lt;&lt; 42 calls std::cout.operator(42)</td>
</tr>
<tr>
<td>a-=</td>
<td>operator( a, b)</td>
<td>cannot be non-member</td>
<td>std::string s; s = &quot;abc&quot;; calls s.operator(&quot;abc&quot;)</td>
</tr>
<tr>
<td>a(b...)</td>
<td>(a).operator(b...)</td>
<td>cannot be non-member</td>
<td>std::random_device r; auto m = r(); calls r.operator( )</td>
</tr>
<tr>
<td>@b</td>
<td>(b).operator()</td>
<td>operator()</td>
<td>std::cout &lt;&lt; 42 calls std::cout.operator(42)</td>
</tr>
<tr>
<td>a&gt;&gt;b</td>
<td>(a).operator( b )</td>
<td>operator( a, b)</td>
<td>std::cout &lt;&lt; 42 calls std::cout.operator(42)</td>
</tr>
<tr>
<td>a+=</td>
<td>operator( a, b)</td>
<td>cannot be non-member</td>
<td>auto p = std::max_unique_ptr&lt;&gt;(); p-&gt;bar() calls p.operator( )</td>
</tr>
<tr>
<td>a&lt;&lt;</td>
<td>operator( a, b)</td>
<td>cannot be non-member</td>
<td>std::vector&lt;int&gt; i; i = 1; calls i.operator( )</td>
</tr>
</tbody>
</table>

Note: for overloading user-defined conversion functions, user-defined literals, allocation and deallocation see their respective articles.

Overloaded operators (but not the built-in operators) can be called using function notation:

```
std::cout operator( (a, b));
```
Simple example: \texttt{operator<<}

- Definition inside type definition

```cpp
#include <iostream>
using namespace std;
struct Type {
    Type(int i, double d) : i(i), d(d) {}
    int i;
    double d;
    friend ostream&
        operator<<(ostream& os, const Type &t) {
            return os << t.i << " and " << t.d;
        }
};

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

- Definition outside type definition

```cpp
#include <iostream>
using namespace std;
struct Type {
    Type(int i, double d) : i(i), d(d) {}
    int i;
    double d;
}

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

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

- 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 variables!

```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) { }
    constexpr size_t size() { return 3; }
    double euclidean_length() { return sqrt(x*x+y*y+z*z); }
    friend ostream& operator<<(ostream& os, const Vec3& v) {
        return os << v.x << '\n' << v.y << '\n' << v.z;
    }
};

Vec3 v1(4,5,6);
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) { }
    constexpr 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: return NAN; break;
        }
    }
};
```

A const function member or operator is one, that does not modify the data members!

We need a `const` version of `operator[]`

```cpp
Vec3 v;
v[1] = 12;
const Vec3 w;
double x = w[1]; // calls const version
```

```cpp
const double& operator[](size_t idx) const {
    switch (idx) {
        case 0: return x; break;
        case 1: return y; break;
        case 2: return z; break;
        default: return NAN; break;
    }
}
```

```cpp
friend Vec3 operator+ (Vec3 lhs, const Vec3& rhs) {
    for (size_t i = 0; i < lhs.size(); ++i)
        lhs[i] += rhs[i];
    return lhs;
}
```

```cpp
friend ostream& operator<< (ostream&amp; os,
    const Vec3&amp; 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;
    cout << v3 << '
';
    return 0;
}
```

- Output
  - 5 52 9
Operator overloading

- You are now able to write these yourself

```cpp
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);
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 really have to implement `operator<` and `operator==` for comparisons
- Others can be expressed using `<` and `==`

```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);
    constexpr size_t size() const;
    double euclidean_length() const;
    double& operator[] (size_t idx);
    const double& operator[] (size_t idx) const;
    friend Vec3 operator+ (Vec3 lhs, const Vec3& rhs);
    friend ostream& operator<< (ostream& os, const Vec3& v);
};
```

- Caution: why is it 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** pointer lets you access the address of the object on which a (non-static) member function is being called on  →  address of the receiver object
  - May be useful for the current exercise sheet
  - All details on **this**, later on
Memory layout

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

- High addresses
- Low addresses

Diagram:
- Kernel
- Stack
- Heap
- Data
- Text
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 this page
Why dynamic memory allocation?

- Consider local 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 data is usually limited by the operating system
      
      - Why? → Think about deallocation
    
    - What if data should be used in more than one scope?
  
  - Dynamically allocated heap data solves the problem
    
    - Simply take (allocate) what you need
    
    - Allocate chunks of memory as large as you like (size of RAM chips)
Dynamic memory allocation

- Ability to 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 the memory after usage
      - The 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
  ```c
  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 & 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
    
    ```
    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 a operator (that can even be overloaded)
  - Standard signatures are
    
    ```
    void operator delete ( void* ptr ); // for objects
    void operator delete[]( void* ptr ); // for arrays
    ```
  - Every **new** needs a **delete** ➔ otherwise your program contains a leak / error
Allocate and delete memory

```cpp
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)
Double free

- Task allocate one single `int`

```cpp
int main() {
    int *dyn_int = new int(13);
    *dyn_int = 42;
    cout << *dyn_int << '
';
    // we have a leak here
    // how to fix?
    delete dyn_int;
    dyn_int = nullptr;
    delete dyn_int;
    return 0;
}
```

```
pdschbrt@apparatus:/Dropbox/upb/C++Programming1617/code/code_04$ ./double_del
*** Error in `./double_del': double free or corruption (fasttop): 0x0000000000000000 ***
Abgebrochen (Speicherabzug geschrieben)
pdschbrt@apparatus:/Dropbox/upb/C++Programming1617/code/code_04$
```
Valgrind, a tool to detect memory misuse [https://valgrind.org/docs/manual/quick-start.html](https://valgrind.org/docs/manual/quick-start.html)

- 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 & more memory, until crash
Clang’s AddressSanitizer [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-bounds access 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
  - $ clang++ -Wall -std=c++17 -O1 -g -fsanitize=address -fno-omit-frame-pointer MyProg.cpp -o MyProg
Valgrind, a tool to detect memory misuse

```c
int main() {
    int *dyn_int = new int;
    *dyn_int = 42;
    cout << *dyn_int << '\n';
    // we have a leak here
    return 0;
}
```
Valgrind, a tool to detect leaks

```c++
int main() {
    int *dyn_int = new int;
    *dyn_int = 42;
    cout << *dyn_int << '\n';
    delete dyn_int;
    dyn_int = nullptr;
    return 0;
}
```

```
Memcheck, a memory error detector
Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
Command: ./alloc
42
HEAP SUMMARY:
in use at exit: 0 bytes in 0 blocks
total heap usage: 1 allocs, 1 frees, 4 bytes allocated
All heap blocks were freed -- no leaks are possible
For counts of detected and suppressed errors, rerun with: -v
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```
Allocate 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
    - Otherwise possibility for index out of bounds
      - Undefined behavior or segmentation fault
Allocate 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 ostream& operator<<(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);
    // -> is shorthand for(*v2).x
    // (dereference and access content)
    v2->x = 42;
    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>
using namespace std;

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

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

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

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

    int* i_ptr;
};

int main() {
    DynInt di1(42);
    DynInt di2 = di1;  // call copy assignment operator
    *di2.i_ptr = 100;  // ups! we copy the pointer
                          // but not what it points to!
    cout << di1 << endl;
    cout << di2 << endl;
    return 0;  // even more ups, we call dtor for di1 and di2
               // we call dtor twice for the same heap
}
```
Copy constructor & copy assign: How to fix it?

```cpp
#include <iostream>
using namespace std;
struct DynInt {
    DynInt(int i) : i_ptr(new int(i)) {}
    ~DynInt() { delete i_ptr; }  // we have to clean up
    DynInt(const DynInt &di) : i_ptr(new int(*di.i_ptr)) {};} // create a real copy!
    DynInt& operator=(const DynInt &di) = default;  // this works, since it calls
        // copy constructor
    friend ostream& operator<<(ostream& os, const DynInt &di) { return os << *di.i_ptr; }
    int *i_ptr;
};

int main() {
    DynInt di1(42);
    DynInt di2 = di1;  // call copy assignment operator
    *di2.i_ptr = 100;  // does this work?
    cout << di1 << '\n';
    cout << di2 << '\n';
    return 0;  // does this work?
}
```
Move constructor & move assign gone wrong

#include <iostream>
using namespace std;
struct DynInt {
    DynInt(int i) : i_ptr(new int(i)) {}
    ~DynInt() { delete i_ptr; } // we have to clean up
    DynInt(const DynInt &di) : i_ptr(new int(*di.i_ptr)) {}; // create a real copy!
    DynInt& operator=(const DynVecUser& dvu) = default; // this now works, since it calls copy constructor

    DynInt(DynInt &&di) = default;
    DynInt& operator=(DynInt &&di) = default;
    friend ostream& operator<<(ostream& os, const DynInt &di) { return os << *di.i_ptr; }
    int *i_ptr;
};

int main() {
    DynInt di1(42);
    DynInt di2 = move(di1); // call move assignment operator use: #include <utility>
    *di2.i_ptr = 100; // does this work?
    cout << di2 << '\n'; // does this work?
    return 0;
}
#include <iostream>

using namespace std;

struct DynInt {
    DynInt(int i) : i_ptr(new int(i)) {} // we have to clean up
    ~DynInt() { delete i_ptr; }
    DynInt(const DynInt& di) : i_ptr(new int(*di.i_ptr)) { }; // create a real copy!
    DynInt(DynInt&& di) = default; // move constructor
    DynInt& operator=(const DynVecUser& dvu) = default; // this now works, since it calls copy constructor
    DynInt& operator=(DynInt&& di) = default;

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

    int* i_ptr;
};

int main() {
    DynInt di1(42);
    DynInt di2 = move(di1); // call move assignment operator
    *di2.i_ptr = 100; // does this work?
    cout << di2 << 'n';
    return 0; // does this work?
}
Move constructor & move assign: How to fix it?

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

struct DynInt {
    DynInt(int i) : i_ptr(new int(i)) {}
    ~DynInt() { delete i_ptr; }  // we have to clean up
    DynInt(const DynInt &di) : i_ptr(new int(*di.i_ptr)) {}  // create a real copy!
    DynInt& operator= (const DynVecUser& dvu) = default;  // this works, since it calls
    // copy constructor
    DynInt(DynInt &&di) { i_ptr = di.i_ptr; di.i_ptr = nullptr; }  // steal the data, and set old pointer to null!
    DynInt& operator= (DynInt &&di) = default;  // this now works, since it calls move constructor
    friend ostream& operator<< (ostream& os, const DynInt &di) { return os << *di.i_ptr; }
};

int main() {
    DynInt di1(42);
    DynInt di2 = move(di1);  // call move assignment operator use: #include <utility>
    *di2.i_ptr = 100;  // does this work?
    cout << di2 << '
';
    return 0;  // does this work?
}
```
Relax now

- You know how to handle special member functions in context of dynamic memory allocation
- It will not get anymore complicated than that

[Figure taken from https://bplusmovieblog.files.wordpress.com/2014/10/the-matrix-2825.png?w=590]
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 works fine
- If you are dealing with dynamic memory yourself, you now know how to deal with special member functions such as copy and move
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 one object
  - Just do so using stack memory
    - and return by value (RVO and `move()` take care of performance)
    - `move()` the variable between scopes, if you want them to live longer then one scope
  - If the object is too large for 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 = move(inner_scope);
    }
    cout << outer_scope << '\n';
    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

- If you need to dynamically allocate an array of objects
  - Use `std::vector`
    - It was created for this purpose
    - Acts as a 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”
      - Implemented by reference counting
  - We will see them 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?