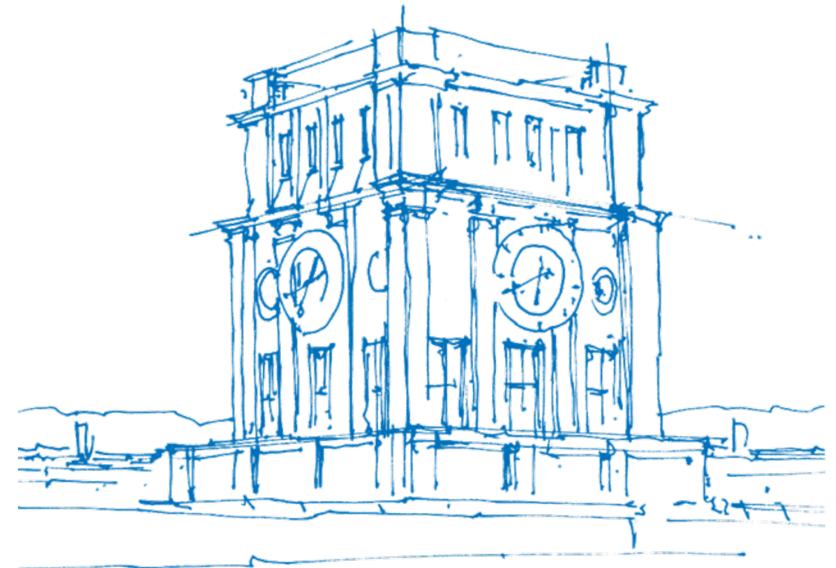
Database Systems on Modern CPU Architectures

Introduction to Modern C++

Moritz Sichert
Technische Universität München
Department of Informatics
Chair of Data Science and Engineering



Overview

Prerequisites:

- You have basic knowledge of C or C++
  - Header files, `.c`/`.cc`/`.cpp` files
  - Preprocessor, compiler, linker
  - Pointers
  - Builtin types (`int`, `unsigned`, etc.)
- You know a programming language (pref. Java)

Contents:

- Overview of C++ language features with examples
- A few important parts of the standard library:
  - Containers
  - Memory management
  - Threads
  - Mutexes
  - Atomics

Hello World

```
#include <iostream>

int main(int argc, char* argv[]) {
    std::cout << "how many numbers should I print? ";
    unsigned numbers = 0;
    std::cin >> numbers;
    for (int i = 1; i <= numbers; ++i) {
        std::cout << i << std::endl;
    }
    return 0;
}
```

Functions

- Parameters can be values or references
- Can be overloaded

```
int f1(int i) { std::cout << "f1(int) " << i; return i + 1; }
float f1(float f) { std::cout << "f1(float) " << f; return f + 1; }
void f2(int& i) { std::cout << "f2 " << i; }

f1(123); // prints "f1(int) 123", returns 124
f1(1.23); // prints "f1(float) 1.23", returns 2.23
f2(456); // compiler error
int i = 456;
f2(i); // prints "f2 456"
```

Pointers and References

```
void increase_value(int value) { ++value; }
void increase_reference(int& value) { ++value; }
// compiler error:
void increase_const_ref(const int& value) { ++value; }
int value = 123;
int* value_pointer = &value;
int& value_reference = value;
increase_value(value);
// value == *value_pointer == value_reference == 123
increase_reference(value);
increase_reference(*value_pointer);
increase_reference(value_reference);
// value == *value_pointer == value_reference == 126
```

Differences between Pointers and References

- Pointers can be null (`nullptr`)
- References cannot be changed to reference another object

```
int value_a = 123;
int value_b = 456;
int* pointer = nullptr;
int& reference; // compiler error
int& reference = value_a;
// using *pointer is a runtime error ("undefined behavior")
pointer = &value_a;
// value_a == *pointer == reference == 123
pointer = &value_b;
// *pointer == value_b == 123
reference = value_b;
// value_b == reference == value_a == 456 (!!!)
```

Classes

```
class ASimpleClass {
private:
    void private_function() { std::cout << "private" << std::endl; }
protected:
    int protected_int;
public:
    int public_int;
    void public_function() { private_function(); protected_int = 123; }
};
ASimpleClass my_object;
my_object.private_function(); // compiler error
my_object.protected_int = 123; // compiler error
my_object.public_int = 123;
my_object.public_function(); // prints "private"
```

Constructors / Destructors

```
class ASimpleClass {
public:
    ASimpleClass() { std::cout << "default constructor"; }
    ASimpleClass(int) { std::cout << "int constructor"; }
    ~ASimpleClass() { std::cout << "destructor"; }
    ASimpleClass(const ASimpleClass&) { std::cout << "copy"; }
    ASimpleClass& operator=(const ASimpleClass&) { std::cout << "=copy"; }
};
void do_nothing_value(ASimpleClass) {}
void do_nothing_reference(ASimpleClass&) {}
ASimpleClass my_object; // prints "default constructor"
do_nothing_value(my_object); // prints "copy destructor"
do_nothing_reference(my_object); // prints nothing
ASimpleClass other_object(123); // prints "int constructor"
other_object = my_object // prints "=copy"
// prints "destructor destructor"
```

Operator Overloading

```
class OperatorClass {
public:
    int operator+(int a) { std::cout << '+'; return a; }
    void operator<<(const OperatorClass&) { std::cout << "<<"; }
};
bool operator==(const OperatorClass&, const OperatorClass&) {
    std::cout << "==";
    return false;
}
OperatorClass o;
int a = o + 123; // prints "+", a == 123
o << o; // prints "<<"
o << 123; // compiler error
if (o == o) { /* never reached, prints "==" */ }
```

Inheritance

```
class Base {
public:
    void test() { std::cout << "Base test()"; }
    virtual void virtual_test() { std::cout << "Base virtual_test()"; }
};
class Derived : public Base {
public:
    void test() { std::cout << "Derived test()"; }
    void virtual_test() override { std::cout << "Derived virtual_test()"; }
};
Derived derived;
Base& base = derived;
base.test(); // prints "Base test()"
base.virtual_test(); // prints "Derived virtual_test()"
derived.test(); // prints "Derived test()"
```

Templates

- “Generics” of C++
- Allows for type-safe generic programming
- Is implemented by essentially copying templated code for every type it is used with (as opposed to type-erasure in Java, for example)

```
template <typename T> T increment(T value) { return ++value; }
increment<int>(123); // uses a copy of increment where T=int, returns 124
increment<float>(1.23); // same, but for float, returns 2.23
increment(456); // template arguments can also be automatically deduced in some
                // cases, this is equivalent to increment<int>(456)
template <typename T> class Container { T& front(); /* [...] */ };
Container<int> int_container;
Container<float> float_container = int_container; // compiler error
```

Converting Types ("casting")

Types of casts:

- `static_cast`: Conversion that may change object representation
- `reinterpret_cast`: Conversion without changing object representation, useful mainly for pointers
- `dynamic_cast`: Safely converting references (or pointers) to base class to references to derived class
- `const_cast`: Making const references or pointers non-const

static_cast

```cpp
int i = 123;
float f = i; // implicit conversion
double d = static_cast<double>(f);
class IntLikeClass {
public:
    IntLikeClass(int) { std::cout << "constructor"; }
    operator int() { std::cout << "conversion"; return 456; }
};
void print_int(int i) { std::cout << "print_int " << i; }
IntLikeClass object = static_cast<IntLikeClass>(i); // prints "constructor"
i = static_cast<int>(object); // prints "conversion", i == 456
print_int(object); // prints "conversion print_int 456"
```

reinterpret_cast

- Conversion between pointers
- Allowed conversions:
  1. From any pointer to `char*` or sufficiently large integer (e.g. `uintptr_t`)
  2. From such an integer back to a pointer
  3. From `char*` to any other pointer
- Type aliasing in particular is *not* allowed

```
float f = 12.34;
float* f_p = &f;
char* char_pointer = reinterpret_cast<char*>(f_p); // OK (point 1)
uintptr_t uint_pointer = reinterpret_cast<uintptr_t>(f_p); // OK (point 1)
float* f_p2 = reinterpret_cast<float*>(uint_pointer); // OK (point 2)
int* int_pointer = reinterpret_cast<int*>(f_p); // Not allowed
char* data = get_data();
int* int_pointer3 = reinterpret_cast<char*>(data); // OK (point 3)
```

`reinterpret_cast` and Type Aliasing

- Accessing an object through a pointer of a different type is not allowed (except `char*`)
- Enables type-based optimizations

This is *not* allowed in C++:

```
float y = 1.23; int i;
// Snippet from Quake III source code:
i = * ( long * ) &y; // evil floating point bit level hacking
i = 0x5f3759df - ( i >> 1 ); // what the fuck?
y = * ( float * ) &i;
```

Do this instead:

```
float y = 1.23; int i;
std::memcpy(&i, &y, sizeof(float));
i = 0x5f3759df - ( i >> 1 ); // what the fuck?
std::memcpy(&y, &i, sizeof(float));
```

dynamic_cast

```
class Base {};
class Derived1 : public Base {};
class Derived2 : public Base {};
Derived1 d1;
Base& b = d1; // static_castor implicit conversion
Base* b_ptr = &b;
dynamic_cast<Derived1&>(b); // OK
dynamic_cast<Derived2&>(b); // throws std::bad_cast
dynamic_cast<Derived1*>(b_ptr); // OK
dynamic_cast<Derived2*>(b_ptr); // OK, returns nullptr
```

Type Inference

```
auto i = 123; // i is an int
auto j = i; // j is an int
auto& k = j; // k is an int reference
auto d = i * 2.0; // d is a double
template <typename Arg1, typename Arg2> class ATemplatedClass {
public:
    class nested_type {};
};
ATemplatedClass<FirstTemplateArgument, SecondTemplateArgument> get_object();
auto o = get_object();
decltype(o)::nested_type n1;
decltype(get_object())::nested_type n2;
decltype(1 + 1.0) value = 2; // value is a double with value 2.0
```

Lambda Functions

```
template <typename F> void call_func(const F& func) { func(123); }
call_func([](int i) { std::cout << i; }); // prints "123"
int a = 456;
call_func([a](int i) { std::cout << (a + i); }); // prints "579"
call_func([&a](int i) { a = i; }); // a is 123 now

template <typename F> /*???*/ return_func(const F& func) { return func(123); }
// Solution for C++11:
template <typename F>
auto return_func(const F& func) -> decltype(func(123)) { return func(123); }
// Solution for C++14:
template <typename F>
auto return_func(const F& func) { return func(123); }
```

R-value References

- Objects can be copied with the copy constructor or copy assignment operator
- But: ownership of an object cannot be transferred
- Work-around: Allocate the object on the heap and use the pointer to it to transfer ownership
- Solution: R-value references that enable move constructors and move assignment operators

```
void take_reference(int&) { std::cout << "lvalue ref"; }
void take_reference(const int&) { std::cout << "const lvalue ref"; }
void take_reference(int&&) { std::cout << "rvalue ref"; }
int get_number() { return 789; }
int a = 123;
take_reference(a); // prints "lvalue ref"
take_reference(std::move(a)); // prints "rvalue ref"
const int b = 456;
take_reference(b); // prints "const lvalue ref"
take_reference(123); // prints "rvalue ref"
take_reference(get_number()); // prints "rvalue ref"
```

Move-constructors and Move-assignment

```
class ExpensiveObject {
private:
    int* resources;
public:
    ExpensiveObject(const ExpensiveObject& other) {
        resources = make_copy_of_resources(other.resources);
    }
    ExpensiveObject(ExpensiveObject&& other) {
        resources = other.resources;
        other.resources = nullptr;
    }
};
```

Containers

- `std::vector<T>`: dynamically sized array
- `std::list<T>`: doubly linked list
- `std::map<K, V>`: search tree
- `std::unordered_map<K, V>`: hash table
- Standard library containers use iterators
- Iterators can be used in a “range-for” loop

```
std::vector<int> v{1, 2, 3};
v.push_back(10);
for (int i : v) {
    std::cout << i << ' ';
}
// prints 1 2 3 10
```

Memory Management

- In C++ new memory can be allocated on the heap with `new` and freed with `delete`
- But this has the same disadvantages as `malloc`/`free` in C, i.e. forgetting to free memory or freeing it more than once
- Use `std::unique_ptr<T>` or `std::shared_ptr<T>` instead
- `std::unique_ptr<T>`: Represents a pointer which holds a `T` object that is owned exactly once, memory is freed when the `unique_ptr` is destroyed
- `std::shared_ptr<T>`: Represents a pointer which holds a `T` object that can be owned multiple times at different places, memory is freed when the last owning `shared_ptr` is destroyed

```
int* int_pointer = new int; // allocate memory for one int
delete int_pointer;
int* int_array = new int[100]; // allocate memory for 100 ints
delete[] int_array;
```

`std::unique_ptr<T>`

- In header `<memory>`
- `std::unique_ptr<T>`: Represents a pointer which holds a `T` object that is owned exactly once
- memory is freed when the `unique_ptr` is destroyed
- Cannot be copied, ownership is transferred when moved
- Useful helper function `std::make_unique<T>()`

```
auto int_pointer = std::make_unique<int>();
// int_pointer is a std::unique_pointer<int>
*int_pointer = 123;
auto other_int_pointer = int_pointer; // compile error
auto other_int_pointer = std::move(int_pointer); // OK, move constructor
// now, int_pointer == nullptr and *other_int_pointer == 123
if (int_pointer) { /* never reached */ }
other_int_pointer.reset(nullptr); // memory is freed
```

std::shared_ptr<T>

- In header `<memory>`
- `std::unique_ptr<T>`: Represents a pointer which holds a `T` object that can be owned multiple times at different places
- memory is freed when the last owning `shared_ptr` is destroyed
- When copied, increases reference count, when moved, transfers ownership
- Useful helper function `std::make_shared<T>()`

```
auto int_pointer = std::make_shared<int>();
// int_pointer is a std::shared_pointer<int>
*int_pointer = 123;
auto other_int_pointer = int_pointer; // OK, reference count increased
// now, *int_pointer == *other_int_pointer == 123
int_pointer.reset(nullptr); // memory is NOT freed yet
other_int_pointer.reset(nullptr); // now memory is freed
```

Threads

- Every platform has different threading APIs
- C++11 introduced `std::thread` in the header `<thread>`
- Can be nicely combined with lambda functions
- `std::thread` objects cannot be copied or moved

```
std::vector<std::thread> threads;
for (int i = 0; i < 10; ++i) {
    threads.emplace_back([i] {
        std::cout << (i + 1);
    });
}
// prints the numbers 1 to 10 in any order
// all threads must be joined before destructor is called
for (auto& t : threads) {
    t.join();
}
```

Mutexes

- With multi-threading, mutexes are needed
- Again, different APIs for different platforms
- C++11 introduced `std::mutex` in `<mutex>` with `lock()` and `unlock()`
- since C++17 there is also `std::shared_mutex` in `<shared_mutex>` that additionally has `lock_shared()` and `unlock_shared()`
- As for memory allocation, using the locking methods manually can easily lead to inconsistencies or even crashes
- "`std::unique_ptr` for mutexes" is called `std::unique_lock`
- "`std::shared_ptr` for shared mutexes" is called `std::shared_lock`
- Just like threads, mutex objects cannot be copied or moved

Mutex Example

```
std::mutex m;
std::vector<std::thread> threads;
int number = 0;
for (int i = 0; i < 10; ++i) {
    threads.emplace_back([i, &m, &number] {
        std::unique_lock lock(m);
        ++number;
    });
}
// at the end number is always 10
for (auto& t : threads) {
    t.join();
}
```

Atomics

- Mutexes are relatively slow
- Many CPUs can execute some instructions atomically
- C++11 introduces `std::atomic<T>` in `<atomic>` as a high-level wrapper

```
std::vector<std::thread> threads;
std::atomic<int> number;
number = 0; // atomic assignment
for (int i = 0; i < 10; ++i) {
    threads.emplace_back([i, &number] {
        ++number; // atomic increment
    });
}
// at the end number is always 10
for (auto& t : threads) {
    t.join();
}
```

More Information

- Best reference for C++ language and standard library: `https://en.cppreference.com/`
- C++ standard document (unofficial draft): `http://www.open-std.org/jtc1/sc22/wg21/`
- Feel free to ask any question about C++ (even if it is not directly related to a programming assignment) in Mattermost