Zusammenfassung C++

Konzepte der C++ Programmierung (Technische Universität München)

cpp

Jakob Lambert-Hartmann

February 17, 2023

Contents

1 Basics

1.1 Compilation

preprocessing replaces `#include` statements

compilation *compilation*(*.hpp* and *.cpp*) $\mapsto$ *.o*: This only parses the files

linking definitions are linked

input of compiler is called translation unit output is object code

1.2 Type safety

- not type safe, cast objects without compilation errors.
- strongly typed, every object requires a type

1.3 Value categories

lvalues : object has identity(used on left/right side of assignment). `&var` is a lvalue

rvalues : temporary objects `&&var` is a rvalue

prvalue : pure rvalue

xvalue : lvalue converted to rvalue by `std::move()`

1.4 Increment and decrement

`++a` changes value and returns object
`a++` creates copy of object, changes value and returns copy
Prefix better performance.

1.5 Casts

static_cast converts value

const_cast, reinterpres_cast like C-style, don't use

dynamic_cast for polymorphy

1.6 Enum classes

```
enum class TrafficLight{red, green, yellow};
TraficLight t1 = TrafficLight::yellow;
```

1.7 Exceptions

Thrown with `throw`(without `new`), handled with `try-catch`. Use `int function() noexcept;` when function never throws.
Functions not allowed to throw:

- destructors
- move constructors and assignment operators

2 Functions

functions are not objects. But type T can be function object if:

- `T` is object and has `operator()`

2.1 std::function

std::function can wrap around a function

2.2 lambda expressions

```
[capture_list] (param_list) constexpr -> return_type {body}
```

capture list : variables imported either by value(`[i]`) or by reference(`[&i]`). Import all variables with default capture:

- `[&]` by reference
- `[=]` by copy

e.g.: `[=, &i]`: all by val, i by ref

param_list : parameters, can be omitted when empty

return type : optional

return type : constexpr optional, implicitly constexpr when possible, but ensured with keyword

Type of lambda expression undefined, meaning no two lambdas same. To return it use `std::function`

2.2.1 templated lambdas

Lambdas can be templated and have auto parameters. `auto` also viable in other functions $\mapsto$ templated function

```
auto twice = [] (auto x) {return x + x;};
[]<typename T>(T x, T y){return x + y;};
int function(auto i){...}
```

3 Classes

3.1 access modes

public : access everyone

protected : access class and subclasses

private : access class itself

3.2 friends

friend declarations give access to private+protected

```
class A {
        friend class B;       //class B is friend of A
        friend void foo(A&); // non member function foo ist friend of A
};
```

3.3 forward declaration

Needed for circular dependencies:

```
class A;
class B{ A a };
class A{ B b };
```

3.4 const functions

Const functions can not change state of object (except members with `mutable`)

3.5 explicit constructors

```
struct Foo {explicit Foo(int i);};
void printFoo(Foo a);
printFoo(123) // ERROR: no implicit conversion to struct Foo
```

3.6 ref qualifiers

```
struct Bar{ void foo() & { ... } void foo() && { ... } };
```

Value category overload(rvalue, lvalue). First function `foo` called on lvalues, second called on rvalues(`Bar{}.foo()` calls second)

3.7 Inheritance

Inheritance mode

public : public members are public and protected is protected

protected : public and protected members are only accessible from friend of derived class and its derived classes

private : public and preotected are only accessible from friends of derived class

virtual : for multiple inheritances (when multiple definitions provided by base classes)

```
class Base{ int a; };
Class Derived0 : public Base{ int b; }
```

3.8 functions keywords

virtual : functions must be implemented in derived class

final : prevents overriding function

3.9 caveats of polymorphy

3.9.1 constructors and virtual functions

during construction virtual functions behave like regular functions. `Base(){foo();}` calls `Base::foo` even if virtual.

3.9.2 virtual destructors

Derived class deleted as base class pointers $\mapsto$ undefined behavior. Thus destructors of the base class should be:

- protected and non virtual
- public and virtual

3.9.3 slicing

assigning derived class to base class loses members defined in derived class (also when using references) use `dynamic_cast<new_type> (expression)` to avoid slicing problem

3.10 vtables

polymorphism has overhead. In stack there is a vtable pointer. This pointer is a reference table for each derived class, which is pointing to the correct function implementations.
This costs

runtime : each function call has to follow the pointer to the vtable and then follow the pointer to the actual function

memory : polymorphic objects have a larger size as it has to store pointers to the vtable and the vtable itself

3.11 abstract classes and pure virtual functions

A pure virtual function (`virtual int function() = 0;`) must be implemented by the derived classes
Abstract classes cannot be instantiated, but they can be used as base class and its possible to have references to abstract classes
destructor can be marked as pure virtual when class needs to be abstract, but there is no suitable function that can be declared pure virtual.

4 Resources Acquisition is Initialization

Concept to bind resource lifetime to object lifetime by

- constructor acquires resource (malloc)

- destructor releases resource (free)
- typically delete copy operations and implement move operations

For instance `std::ofstream` closes file automatically.

4.1 Copy

By default usually shallow copy. If not deleted declared if used(member wise copy). Copy constructor implicitly declared when not defined by user. If one of following, copy constructor/assignment `= deleted`

- class has non static data that cannot be copy constructed/assigned
- class has base class which cannot be copy constructed/assigned
- class has base class with deleted or inaccessible destructor
- class has user defined move constructor/assignment

4.1.1 Copy elision

compilers have to omit copy/move constructions:

- in a return statement when the operand is a prvalue of a class of the return type

```
T f(){ return T();}
f(); //only one call to constructor
```

- in initialization of object when expr is class of variable type

```
T x = T{T{f()}}; // only one call to constructor
```

code that relies on side effect of copy/move constructors is not portable

4.2 Move

Copy sometimes not wanted or unnecessary overhead. Move to steal resources $\mapsto$ improved performance.
Move constructor(defined `A(A&& other)`) invoked with:

- `A a{std::move(b)}`
- `A a = std::move(b)`

Compiler will declare public move constructor if

- there are no user declared copy constructors
- there are no user declared copy assignment operators

- there are no user declared move assignment operators
- there are no user declared destructors

Compiler will declare public move assignment if all the following are true

- there are no user declared copy assignment operators
- there are no user declared copy constructors
- there are no user declared move constructors
- there are no user declared destructors

4.3 Rule of Three

If a class requires one then probably all three

- user defined destructor
- user defined copy constructor
- user defined copy assignment

4.4 Rule of Five

When class wants move semantics it has to define:

- move/copy constructor
- move/copy assignment
- destructor

If copy not wanted, then define as `= delete`

4.5 Rule of Zero

classes not dealing with ownership should not have custom:

- move/copy constructor
- move/copy assignment
- destructor

5 Ownership

Resource owned by one object. Ownership can only be transferred by moving.

5.1 owning smart pointer

- `std::unique_ptr` take ownership. Delete when object out of scope
- movable, ¬ copyable
- creation: `std::make_unique<type>(arg0,..., argN)`
- dereferencing: `*,[],->`
- converts to bool to check if empty
- `get()` ↦ raw pointer. `release()` ↦ `get()` + releases ownership

5.2 shared pointer

- `std::shared_prt` resource ownership shared. Only released when every object out of scope
- movable & copyable
- overhead, only use when necessary
- use `std::weak_ptr` to break cycles

5.3 guidelines

- moving always preferred over copying
- passing by reference if ownership not transferred
- passing by rvalue reference if ownership is transferred
- passing by value if should be copied

6 Templates

no code generated until instantiated.

explicit instantiation : Instantiation of specific specialization

```
template class tmpl_name <arglist>; // class
template ret_type name <arglist>(paramlist); // function
```

- can be compiled into lib
- can be defined in source file
- no unforseen types

implicit instantiation : Use template (only takes used parts)

- definition in header file
- template user needs to compile
- usable for any type
- only needed code compiled

6.1 default template parameters

```
template <typename T = std::byte>
```

6.2 template specialization

General template + specialized template for specific types

```
template <typename T> clas Container {
 /* implementation generic */};

template <> clas Container<float> {
 /* implementation for float */};
```

Also possible for partial specialization(only for classes)

```
template <typename A, typename B> clas Container {
 /* implementation generic */};

template <typename C> clas Container<float, C> {
 /* implementation for float */};
```

6.3 deduction guides

```
template <typename T1, typename T2>
struct Pair {
        T1 first; T2 Second;
        Pair(const T1& x, const T2& y);
};

// deduction guide
template <typename T1, typename T2>
Pair(T1, T2) -> Pair<T1, T2>;
```

6.4 Metaprogramming

templates with `constexpr` for compile time programming

```
template <unsigned N>
constexpr unsigned fibonacci(){
        if constexpr (N >= 2)
                return fibonacci<N-1>() + fibonacci<N-2>();
        else
                return N;
}
```

6.5 Type traits

6.5.1 substitution failure is not an error (SFINAE)

```
int negate(int i) {return -i;}

template <typename T>
typename T::value_type negate(const T& t) {return -T(t); }
```

enable if

```
template <typename T, typename std::enableif<std::is_arithmetic_v<T>, bool>::typ
```

6.6 Concepts

```
template <typename T>
requires std::integral<T>
T gcd(T a, T b){...}
```

7 Compile time programming

`static_assert` check at compile time, compile error when evaluates to false

7.1 Parameter packs

```
template <typename Head, typename... Tail>
void printElements (const Head& head, const Tail&... tail){
        std::cout << head;
        if(sizeof...(tail) > 0) std::cout << ",";
        printElements(tail...);
}
```

unary left : `... op pack`

unary right : `op pack ...`

binary left : `init op ... op pack`

binary right : `pack op ... op init`

7.2 Constant Expression

constexpr variables : must be

- literal type (`std::string` and `std::vector` possible since C++20)
- immediately initialized with constexpr

constexpr functions : possible as member functions, must

- have literal return type
- no `goto`, var definition of non-literal type

```
int f(int x) {return x * x;}
constexpr int g(int x) {return x * x;}
int main(){
        const int x = f(x); //not constexpr
        const int y = g(x); //constexpr
        const int z{1}; // constexpr
        constexpr int xx{x}; // error f(x) not constexpr
        constexpr int yy{y}; // ok
        constexpr int zz{z}; // ok
}
```

7.2.1 constexpr if

```
template <typename T>
auto getValue (T t){
        if constexpr (std::is_pointer<T>)
                return *t; // no compile error because only run when t pointer
        else
                return t;
}
```

7.2.2 consteval

`constexpr` can be called at runtime, `consteval` guaranteed compile time.