Concepts of C++ Programming

winter term 2023/24

PD Dr. Tobias Lasser

Computational Imaging and Inverse Problems (CIIP)
Technical University of Munich

Tweedback today

The Tweedback session ID today is zb4b, the URL is:

`https://tweedback.de/zb4b`

- I will try prioritizing upvoted questions first!

Contents

Return types

- we already know how to specify a return type:

```
int foo(); // foo returns an int
```

- C++ also allows a trailing return type:

```
auto foo() -> int; // foo returns an int
```

  - the keyword `auto` is fixed, the actual return type follows after the parameter list and the symbols `->`

- this really becomes useful with templates (see later), where the return type depends on the arguments:

```
template <typename T, typename U>
auto sum(const T& x, const U& y) -> decltype(x+y);
```

  - but you might also just like the style better

Returning multiple values

- returning more than one value from a function is not supported directly by the syntax
- but we can use structured bindings and `std::pair` or `std::tuple` to do so:

```cpp
std::pair<int, std::string> foo() {
    return std::make_pair(17, "C++");
}

std::tuple<int, int, float> bar() {
    return std::make_tuple(1, 2, 3.0);
}

int main() {
    auto [i, s] = foo(); // i is int with i == 17,
                         // s is std::string with s == "C++"
    auto [a, b, c] = bar(); // a, b are int, c is float,
                            // a == 1, b == 2, c == 3.0
}
```

Structured bindings

- structured bindings allow you to initialize multiple entities by elements / members of an object (such as `std::pair` or `std::tuple`)
- they work nicely with the standard library, for example with associative containers like `std::map`:

```
std::map<std::string, int> myMap; // map with strings as keys
// ... fill the map ...

// iterate over the container using range-for loop
for (const auto& [key, value] : myMap)
    std::println("{}: {}", key, value);
```

Structured bindings (cont.)

- you can also bind `struct` members or `std::array` entries:

```
struct myStruct { int a{1}; int b{2}; };
auto [x, y] = myStruct{}; // x, y are int, x == 1, y == 2

std::array<int, 3> myArray{47, 11, 9};
auto [i, j, k] = myArray; // i == 47, j == 11, k == 9
```

- structured bindings can have qualifiers (references, `const`):

```
myStruct ms;
auto& [u, v] = ms;          // u, v now reference ms.a, ms.b
ms.a = 11;
std::println("{}", u);     // prints 11
u = 22;
std::println("{}", ms.a); // prints 22
```

- you can even provide a `std::tuple`-like API for your own data types to enable structured bindings (see reference)

Returning multiple values revisited

- specifying the return type for multiple values can be annoying:

```
std::tuple<int, int, float> bar() {
    return std::make_tuple(1, 2, 3.0);
}
```

- with auto we can let the compiler deduce the return type automatically (even without trailing return type):

```
auto bar() {
    return std::make_tuple(1, 2, 3.0);
}
```

- this is convenient for the author of the function, but not necessarily great for the user of the function (that might not even be able to see the function body!)

Contents

Parameter passing revisited

We can pass parameters to functions:

- by value:

```
void foo(int value);
```

- by reference:

```
void foo(int& value);
```

- by const reference:

```
void foo(const int& value);
```

How to choose?

Parameter passing

Refer to the C++ Core Guidelines:
`https://isocpp.github.io/CppCoreGuidelines/`

	Cheap or impossible to copy (e.g., int, unique_ptr)	**Cheap to move** (e.g., vector<T>, string) or **Moderate cost to move** (e.g., array<vector>, BigPOD) or **Don't know** (e.g., unfamiliar type, template)	**Expensive to move** (e.g., BigPOD[], array<BigPOD>)
Out	X f()	X f()	
In/Out	f(X&)	f(X&)	f(X&)
In	f(X)	f(const X&)	f(const X&)
In & retain "copy"	f(X)	f(const X&)	f(const X&)

"Cheap" ≈ a handful of hot int copies
"Moderate cost" ≈ memcpy hot/contiguous ~1KB and no allocation

** or return unique_ptr<X>/make_shared_<X> at the cost of a dynamic allocation*

Parameter passing (cont.)

Summarized guidelines:

- "in" parameters: pass by value (for cheaply-copied types) or pass by const reference

```
void f1(const std::string& s); // OK, pass by const reference
void f2(std::string s);        // potentially expensive
void f3(int x);                // OK, cheap
void f4(const int& x);         // not good, unnecessary overhead
```

- "in-out" parameters: pass by reference (if you cannot avoid it)

```
void update(Record& r);  // assume that update writes to r
```

- "out" parameters: return them, either as single return type or as `std::pair`, `std::tuple`

Contents

Overloading

Overloaded functions:

- we can declare different functions having the same name but different argument types:

```
void f(int);    // a function called f, taking an int
void f(double); // another function f, taking a double
```

- on a function call, the compiler automatically resolves the overloads in the current scope and calls the best match
    - if there is no best match, it's a compile error

Overload resolution criteria

The following criteria are tried in order:

1. exact match (no or only trivial conversions, e.g. `T` to `const T`)
2. match using promotions (e.g. `bool` to `int`, `char` to `int`, or `float` to `double`)
3. match using standard conversions (e.g. `int` to `double`, `double` to `int`, or `int` to `unsigned int`)
4. match using user-defined conversions (see later)
5. match using ellipsis `...` (see later)

Overloading: examples

```
void print(int);
void print(double);
void print(long);
void print(char);

void f(char c, int i, short s, float f) {
    print(c);   // exact match: print(char)
    print(i);   // exact match: print(int)
    print(s);   // integral promotion: print(int)
    print(f);   // float to double promotion: print(double)

    print('a'); // exact match: print(char)
    print(49);  // exact match: print(int)
    print(0);   // exact match: print(int)
    print(0L);  // exact match: print(long)
}
```

Overloading: examples (cont.)

```
using complex = std::complex<double>;

int pow(int, int);
double pow(double, double);
complex pow(double, complex);
complex pow(complex, int);
complex pow(complex, complex);

void h(complex z) {
    auto i  = pow(2, 2);     // invokes pow(int, int)
    auto d  = pow(2.0, 2.0); // invokes pow(double, double)
    auto z2 = pow(2, z);     // invokes pow(double, complex)
    auto z3 = pow(z, 2);     // invokes pow(complex, int)
    auto z4 = pow(z, z);     // invokes pow(complex, complex)
    auto e  = pow(2.0, 2);   // ERROR: ambiguous
}
```

Overloading and the return type

Caution: return types are **not** considered in overload resolution!

- this is good, so function calls are context-independent:

```cpp
float sqrt(float);
double sqrt(double);

void f(float fla, double da) {
    float fl = sqrt(da);  // invokes sqrt(double)
    auto d = sqrt(da);    // invokes sqrt(double)
    fl = sqrt(fla);       // invokes sqrt(float)
    d = sqrt(fla);        // invokes sqrt(float)
}
```

Contents

Functors

Functions are not objects in C++

- they cannot be passed as parameters
- they cannot have state

However, a type `T` can be a function object (or functor), if:

- `T` is an object
- `T` defines `operator()`

Function objects can be used like functions.

Functor example

Functor storing a state:

```cpp
struct Adder {
    int value{1};

    int operator() (int param) {
        return param + value;
    }
};

int main() {
    Adder myAdder;
    myAdder.value = 5;
    myAdder(1);          // returns 6
    myAdder(4);          // returns 9
    myAdder.value = 7;
    myAdder(1);          // returns 8
}
```

std::function

std::function is a wrapper for all callable targets

- defined in `<functional>` header
- stores, copies, and invokes the wrapped target
- caution: can incur a slight overhead in both performance and memory

```
#include <functional>
int addFunc(int a) { return a + 3; }

int main() {
    std::function adder{addFunc};
    int a{adder(5)};   // a == 8

    // alternatively specifying the function type:
    std::function<int(int)> adder2{addFunc};
}
```

- function type is declared as `return_type(argument_list)`
- deduction guides usually makes this unnecessary

std::function example

```cpp
#include <functional>
#include <iostream>

void printNum(int i) { std::println("{}", i); }

struct PrintNum {
    void operator() (int i) { std::println("{}", i); }
};

int main() {
    // store a function
    std::function f_printNum{printNum};
    f_printNum(-47);

    // store the functor
    std::function f_PrintNum{PrintNum{}};
    f_PrintNum(11);

    // fix the function parameter using std::bind
    std::function<void()> f_leet{std::bind(printNum, 31337)};
    f_leet();
}
```

Contents

Lambda expressions

Lambda expressions are a simplified notation for anonymous function objects

- they are an expression and can be used anywhere expressions can be used:

```
std::find_if(container.begin(), container.end(),
    [](int val) { return 1 < val && val < 10; }
);
```

- the function object created by the lambda expression is called closure
- the closure can hold copies or references of captured variables

Lambda expressions: the syntax

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

- `capture_list` specifies the variables of the environment to be captured in the closure
- `param_list` are the function parameters
- `return_type` specifies the return type; it is optional! If not specified, the return type is deduced from the return statements in the body

- the `capture_list` can be empty: `[]() {}`
- if the `param_list` is empty, it can be omitted
  - the shortest lambda expression is thus: `[]{}`

Lambda expressions: examples

```
bool isMultipleOf3(int v) { return v % 3 == 0; }

void f() {
    std::vector<int> numbers{ /* ... */ };

    // version with functions:
    std::remove_if(numbers.begin(), numbers.end(), isMultipleOf3);

    // version with lambdas:
    std::remove_if(numbers.begin(), numbers.end(),
        [](int v) { return v % 3 == 0; }
    );
}
```

Storing lambda expressions

Lambda expressions can be stored in variables:

- using auto:

```
auto lambda = [](int a, int b) { return a + b; };

std::println("{}", lambda(2, 3)); // outputs 5
```

- using std::function:

```
std::function func = [](int a, int b) { return a + b; };

std::println("{}", func(3, 4)); // outputs 7
```

- the signature of the lambda can be stated explicitly:

```
std::function<int(int, int)> func2 =
    [](int a, int b) { return a + b; };

std::println("{}", func2(4, 5)); // outputs 9
```

The type of a lambda expression

- the type of a lambda expression is not defined
- no two lambdas have the same type (even if they are identical otherwise):

```
auto myFunc(bool first) { // ERROR: ambiguous return type
    if (first)
        return []() { return 42; };
    else
        return []() { return 42; };
}
```

- for each lambda, the compiler generates a unique closure class with a constructor and `operator() const`
  - in fact, we could do this ourselves, it's just more effort
  - nevertheless, lambdas turned out to be a game changer

Lambda expression return types

- just like for functions, the return type can be deduced from the return statement:

```cpp
void f() {
    auto x = [] { std::println("Hello"); }; // void return type

    auto y = [] { return 42; }; // int return type

    auto z = [] { if (true) return 1; else return 2.0; };
                  // ERROR: inconsistent types

    auto z2 = [] -> int { if (true) return 1; else return 2.0; };
                  // OK: explicit return type
}
```

Lambda captures

- lambda captures are part of the state of a closure
  - can refer to automatic variables in the surrounding scopes
  - can refer to the `this` pointer in the surrounding scope (see later)
- capture can be done by copy: creates a copy of the captured variable in the closure
- capture can be done by reference: creates a reference to the captured variable in the closure
- captures can be used in the lambda expressions like regular variables or references

Lambda captures: examples

```cpp
void f() {
    int i{0};

    auto lambda1 = [i]() { std::cout << i; }; // i by copy
    auto lambda2 = [i]() { ++i; }; // ERROR: i is read-only!

    auto lambda3 = [&i]() { ++i; }; // OK: i by reference
    lambda3();
    std::cout << i; // outputs 1
}
```

Caution: beware dangling references:

```cpp
auto g() {
    int j{0};
    return [&j]() { ++j; }; // beware: reference to j will dangle!
}
```

Lambda captures: examples (cont.)

Capture by copy vs. by reference:

```
void f() {
    int i{42};

    auto lambda1 = [i]() { return i + 42; };
    auto lambda2 = [&i]() { return i + 42; };

    i = 0;

    int a = lambda1();   // a == 84
    int b = lambda2();   // b == 42
}
```

Lambda default captures

- capture defaults allow you to capture all variables in the surrounding scope
  - by copy: [=]
  - by reference: [&]
- if defaults are used, only diverging capture types can be specified afterwards:

```cpp
void f() {
    int i{0}, j{42};

    auto lambda1 = [&, i]() {};  // j by reference, i by copy
    auto lambda2 = [=, &i]() {}; // j by copy, i by reference

    auto lambda3 = [&, &i]() {}; // ERROR: non-diverging capture
    auto lambda4 = [=, i]() {};  // ERROR: non-diverging capture
}
```

Lambda default captures (cont.)

- default captures can be dangerous
- in particular default by reference `[&]` may have unwanted side effects

- in general: avoid default captures!

Contents

Handling error conditions

What to do in error conditions?

- terminate the program: `if (somethingWrong) exit(1);`
    - quite drastic, very problematic in libraries
- return an error value and let the caller decide
    - often hard to define an error value, e.g. `int getInt();`
- return a legal value and leave program in error state
    - e.g. the C standard library sets global variable `errno`:

```
double d = sqrt(-1.0);  // value of d is meaningless
```

    - need to test `errno` basically everywhere, also issues with concurrency and global `errno` variable
- call an error-handler: `if (wrong) errorHandler();`
    - but how can the handler handle the problem?

Exceptions

The C++ “solution” is exceptions:

- exceptions transfer control and information up the call stack
- see if the caller(s) can handle the exceptional condition
- exceptions are raised via `throw` expressions
  - `dynamic_cast`, `new` and some standard library functions can also raise exceptions
- exceptions are handled in `try`-`catch` blocks
  - handling them is optional
  - however, if an exception is not caught, the program is terminated
  - errors during handling an exception also lead to program termination

Throwing exceptions

- objects of any type may be thrown as exception objects
- syntax: `throw expression;`
- copy initializes the exception object from `expression` and throws it
- the standard library offers `std::exception` and some derived exception types such as `std::invalid_argument` or `std::out_of_range`
  - `std::exception` contains an explanatory string, it can be queried using `what()`
  - your own exceptions should usually derive from the standard library classes

```
#include <stdexcept>

void foo(int i) {
    if (i == 42)
        throw 42;

    throw std::logic_error("What is the meaning of life?");
}
```

Handling exceptions

- when an exception is thrown, C++ performs stack unwinding
  - ensures proper clean up of objects with automatic storage duration
  - there is some slight run-time overhead, so exceptions are sometimes avoided in real-time applications
- the stack is unwound until a `try`-`catch` block is found
  - exceptions that occur in the `try` block can be handled in the `catch` block
  - the parameter of the `catch` block determines the type of exception that causes the block to be entered

Exception handling example

```cpp
#include <exception>
#include <iostream>

void bar() {
    try {
        foo(42);
    } catch (int i) {
        /* handle the exception somehow */
    } catch (const std::exception& e) {
        std::cerr << e.what() << "\n";
        /* handle the exception somehow */
    }
}
```

Noexcept functions

- some functions do not throw (or should not throw)
- you can mark functions as `noexcept`:

```
int myFunction() noexcept; // may not throw an exception
```

- valuable information for the programmer (no need to handle exceptions) and the compiler (optimizations)
- however, the compiler cannot fully check for the compliance of `noexcept` functions
    - if a `noexcept` function throws anyway, the program is terminated immediately
    - this can happen unexpectedly, e.g. using a `std::vector` and not having enough memory (leading to a `std::bad_alloc` exception)

Exception guidelines

- exceptions should be used rarely
  - main use case: establishing class invariants, for example in RAII (see later)
- exceptions should not be used for control flow!

- some functions must not throw exceptions
  - destructors
  - move constructors and assignment operators
  - see reference documentation for details

C++23 and `std::expected`

- C++23 introduces `std::expected`, another way to handle errors
  - this is very similar to Rust
- `std::expected<T, E>` stores either a value of type `T` or an error of type `E`
  - idea: function return value always contains either the valid result, or an error object
  - `has_value()` checks if the expected value is there
  - `value()` or dereferencing `*` accesses the expected value
  - `error()` returns the unexpected value
  - monadic operations are also supported, such as `and_then()` or `or_else()`
- performance guarantee: no dynamic memory allocation takes place

C++23 and std::expected

usage example (omitting the necessary includes to fit slide):

```cpp
enum class parse_error { invalid_input };

auto parse_number(std::string_view str) -> std::expected<double, parse_error> {
    if (str.empty())
        return std::unexpected(parse_error::invalid_input);
    // do actual parsing ...
    return 0.0;
}

int main() {
    auto process = [](std::string_view str) {
        if (const auto num = parse_number(str); num.has_value())
            std::println("value: {}", *num);
        else if (num.error() == parse_error::invalid_input)
            std::println("error: invalid input");
    };

    for (auto s : {"42", ""})
        process(s);
}
```

Error handling in C++

- both error handling mechanisms, exception handling and `std::expected`, have their value
  - exception handling is well suited for the rare failures you cannot do much about at the called function (e.g. out of memory)
  - `std::expected` is well suited for more regular errors, such as errors in parsing
- also used very often still: returning error codes - not very practical, and requires a lot of discipline in querying error values

Summary

1. Concepts of procedural programming
   - Functions and return values
   - Functions and parameters
   - Function overloading
   - Function objects
   - Lambda expressions
   - Error handling