Prüfung

Files

| Obo | Fasta |
|---|---|
| [Term]<br>id: GO:2001308<br>name: gliotoxin metabolic process<br>namespace: biological_process<br>def: "The chemical reactions and pathways involving the epipolythiodioxopiperazine gliotoxin, a poisonous substance produced by some species of fungi." [PMID:16333108, PMID:17574915, PMID:18272357]<br>synonym: "gliotoxin metabolism" EXACT [GOC:obol]<br>is_a: GO:0006518 ! peptide metabolic process<br>is_a: GO:0006790 ! sulfur compound metabolic process<br>is_a: GO:0043385 ! mycotoxin metabolic process<br>is_a: GO:0046483 ! heterocycle metabolic process<br>is_a: GO:1901360 ! organic cyclic compound metabolic process<br>created_by: pr<br>creation_date: 2012-03-15T03:42:10Z | >sp\|P0DTC3\|AP3A_SARS2 ORF3a protein OS=Severe acute respiratory syndrome coronavirus 2 OX=2697049 GN=3a PE=3 SV=1<br>MDLFMRIFTIGTVTLKQGEIK<br>DATPSDFVRATATIPIQASLPFG<br>WLIVGVALLAVFQSAS<br>KIITLKKRWQLALSKGVHFVC<br>NLLLLFVTVYSHLLLVAAGLE<br>APFLYLYALVYFLQSINF<br>VRIIMRLWLCWKCRSKNPLLY<br>DANYFLCWHTNCYDYCIPYN<br>SVTSSIVITSGDGTTSPIS<br>EHDYQIGGYTEKWESGVKDC<br>VVLHSYFTSDYYQLYSTQLST<br>DTGVEHVTFFIYNKIVDEP<br>EEHVQIHTIDGSSGVVNPVME<br>PIYDEPTTTTSVPL |

Code Formatter: > clang-format filename

Makefile: filename im Makefile ändern, >make compile

Vorlesungen:

• special function `main`
• function declaration
• function definition gives the body of the function
• builtin functions

```
returnType funcName ( parameters ) {
        function body
        return(returnType)
}
```

Namespaces
• can be used to isolate code and variables
• image a global variable x or a function test
• better to have your variables and functions in their own namespace

```
#include <iostream>
// import all methods from iostream std
namespace
// not recommended for header files!!
using namespace std;
// first name space
namespace mapp {
int x = 20;
void func1() {
cout << "Inside mapp::func1 x is: " << x <<
endl;
}
```

Special function features: inline functions
• should be only used if really speed is a problem
• body of `inline` functions is copied into the caller scope
• speed improvements as number of jumps between functions is reduced
• but size increase as same code is copied into multiple places
• useful for very short functions
• `inline` is a proposal for the compiler
• don’t use `#define` macros, better use inline

Pointers and References
• Questions: how to return more than one value/item?
• Answer 1: in principle you can’t – but you can write functions which are called with references as arguments.
• Answer 2: you can return a data structure which keeps more
than one value, such as array, vector, list, map etc.
• Calls:
– call by value - argument variable can’t be changed
– call by reference - argument variable can be changed
 ⁎ using pointers
 ⁎ using references (C++ only)
• Call by references:
– to change the input variable (not so recommended)
– for efficiency reasons (no copy of large data structures)

Remember if x is a pointer:
• `*x` is the value and x is the memory address
• but assignment is like this: `*x = value`
Remember if x is a normal variable:
• x is the value and &x is the memory address (pointer)

A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable. [Tutorialspoint — C++ References, 2019]

const function arguments
• sometimes you call functions per reference for efficiency reasons (large data structures normally) but not to modify the arguments
• to make clear that a variable even given as reference did not change its values you can use the `const` keyword
• any change to the variable will produce a compile error
→ use const if you don’t want to change the variable just like to copy the address of the object into the function

Some recommendations
Prefere references over pointers but:
→ Just use references in parameter lists.
→ Only use them for large data structures or if you like to modifiy the variable.
→ When passing structs or classes (use const if read-only).
→ When passing an argument by reference, always use a const reference unless you need to change the value of the argument.

When not to use pass by reference:
→ When passing fundamental types that don't need to be modified (use pass by value).

Function overloading:
In some programming languages, function overloading or method overloading is the ability to create multiple functions of the same name with different implementations. Calls to an overloaded function will run a specific implementation of that function appropriate to the context of the call, allowing one function call to perform different tasks depending on context. [Wikipedia — Function overloading, 2019]

Lambda captures
You can capture by both reference and value, which you can specify using & and = respectively:
+ [epsilon] - capture epsilon by value
+ [&epsilon] - capture by reference
+ [&] - captures all variables used in the lambda by reference
+ [=] - captures all variables used in the lambda by value
+ [&, epsilon] - captures variables like with [&], but epsilon by value
+ [=, &epsilon] - captures variables like with [=], but epsilon by reference

Which container???
We have map, unorderd map, multimap, unordered multimap, set, array, vector ...
Hint: Until having problems with memory and time use vector and map as data containers, you can ignore the others in 99% of the cases. Other containers can be imitated, for instance multimaps can imitated as map with vectors!

Structures and Classes
• A `struct` is as user defined data type combining data items of different kinds.
• You can see a structure as a C++ class with only public variables but no methods.

Class:
A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class. [Tutorialspoint — C++ Classes and Objects, 2019]

• a `class` with just some public properties is like a `struct`
• `public` properties/methods can be accessed directly from any code outside of the class
• `protected` properties or methods can be accessed from classes/objects inheriting from that class
• `private` properties and methods can be used only inside of this class.
• access operator for member properties and methods is the dot

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object. [Tutorialspoint — C++ Class Member Functions, 2019]

Public variables are usually discouraged, and the better form is to make all variables private and access them with getters and setters. [Stackoverflow — Public or private variables, 2019]

A class constructor is a special member function of a class that is executed whenever we create new objects of that class. A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain
member variables.

```
#include <iostream>
using namespace std;
class Animal {
public:
// constructor 1
Animal(int age) { itsAge = age ;}
// constructor 2 (standard)
Animal() { itsAge=0;}
```

A destructor is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class. A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc. [Tutorialspoint — The Class Destructor, 2019]

| | |
|---|---|
| name.begin() | iterator for begin |
| name.end() | iterator for end |
| .rbegin() | rev. iterator to rev. begin |
| .rend() | reverse iterator to r. end |
| .cbegin() | const_iterator to beginning |
| .cend() | const_iterator to end |
| .crbegin() | const_reverse_iterator to rev. beg. |
| .crend() | const_reverse_iterator r.end |
| .size() | length of string (# elem.) |
| .length() | length of string |
| .max_size() | maximum size of string |
| .resize() | resize string |
| .capacity() | size of allocated storage |
| .reserve() | change in capacity |
| .clear() | clear string |
| .empty() | test if string is empty |
| .shrink_to_fit() | shrink to fit |

We can define class members static using static keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member. A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can’t put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator :: to identify which class it belongs to.

• must be declared inside of the class
• must be initialized outside of the class
→ static member variables are different to static variables of functions

```
#include <iostream>
using namespace std;
class Animal {
	public:
		// constructor 1
		Animal(int age);
		// constructor 2
		Animal();
		~Animal(); // (painless)
destructor
		int getAge();
		static int animalCount; //
variable initialize outside!
		const static int MAXANIMALS = 10;
// const must be inside
	protected:
		int itsAge = 0;
};
// static variable initialize outside
int Animal::animalCount = 0;
int Animal::getAge () {
	return(itsAge);
}
// no argument constructor
Animal::Animal () {
	animalCount += 1;
	cout << "simple animal creation ..." <<
endl;
}
// parameterized constructor
Animal::Animal (int age) {
	animalCount += 1;
	cout << "complex animal creation ..." <<
endl;
	itsAge=age;
}

// destructor definition (always with no
arguments)
Animal::~Animal () {
	cout << "destroy an animal (painless!)"
<< endl;
	animalCount -= 1;
}
void func () {
	Animal animal ;
	cout << "inside func we have " <<
		Animal::animalCount << " animals
" << endl;
}
int main () {
	Animal animal;
	cout <<"Animal 1's age: "
<<animal.getAge() << endl;
	Animal animal2(12);
	cout <<"Animal 2's age: "
<<animal2.getAge() << endl;
	func();
	cout << "inside main we have " <<
		Animal::animalCount << " animals
" << endl;
	cout << "we leave main now ..." << endl;
	return(0);}
```

By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the static functions are accessed using only the class name and the scope resolution operator `::`.
A static member function can only access static data member, other static member functions and any other functions from outside the class.

You have used the const keyword to declare variables that would not change. You can also use the const keyword with member functions within a class. If you declare a class method const, you are promising that the method won’t change the value of any of the members of the class. To declare a class method constant, put the keyword const after the parentheses enclosing any parameters, but before the semicolon ending the method declaration.

Inheritance:
In object-oriented programming, inheritance is the mechanism of basing an object or class upon another object (prototype-based inheritance) or class (class-based inheritance), retaining similar implementa-tion. Also defined as deriving new classes (sub classes) from existing ones (super class or base class) and forming them into a hierarchy of classes. In most class-based object-oriented languages, an object created through inheritance (a ”child object”) acquires all the properties and behaviors of the parent object (except: constructors, destructor, overloaded operators and friend functions of the base class). Inheritance allows programmers to create classes that are built upon existing classes, to specify a new implementation while maintaining the same behaviors (realizing an interface), to reuse code and to independently extend original software via public classes and interfaces. ... Inheritance was invented in 1969 for Simula.

Terminology
• Class: template for creating objects
• Class (static) variable: Variables shared by all objects of a class
• Class (static) method: functions for the class
• Object: a certain instance of a class
• Instance variable: a variable for each object
• Instance method: functions working for the objects
• Inheritance: methods & variables inherited from other classes
• Overriding: functions and operators can be overwritten

by a child class to give them a new functionality
• Overloading: in languages like Java and Cpp classes can have method with the same name but with different arguments
• Component: An embedded but not inherited class/object inside a new class
• Delegation: concept of delegating methods to embedded components
• Mixins: other classes/objects can be embedded on the fly

In computer science, object composition is a way to combine objects or data types into more complex ones. Compositions relate to, but are not the same as, data structures, and common ones are the tagged union, set, sequence, and various graph structures, as well as the object used in object-oriented programming.

• inheritance - class B is-a Class-A. A dog `is-a` animal.
• composition - Class-B has-a Class-A. A dog `has-a` leg.
→ prefer composition over inheritance if possible

Components
• Inheritance: is_a → Student is a Person
• Component: part_of → Head is part of Student
• choose inheritance:
– both classes are in the same logical domain
– subclass is a proper subtype of superclass
– superclass implementation is necessary or appropriate for the subclass
– enhancements made by the subclass are primarily additive.
• choose composition when:
– there is truly a part of relationship
– you only need very few methods of the base class
– you like to hide methods from the base class
– you like to delegate methods to different components from an object

Multiple inheritance:
... is a feature of some object-oriented computer programming languages in which an object or class can inherit characteristics and features from more than one parent object or parent class. It is distinct from single inheritance, where an object or class may only inherit from one particular object or class.
Multiple inheritance has been a sensitive issue for many years, with opponents pointing to its increased complexity and ambiguity in situations such as the ”diamond problem”, where it may be ambiguous as to which parent class a particular feature is inherited from if more than one parent class implements said feature. This can be addressed in various ways, including using virtual inheritance. Alternate methods of object composition not based on inheritance such as mixins and traits have also been proposed to address the ambiguity.

Abstract class:
... is, conceptually, a class that cannot be instantiated and is usually implemented as a class that has one or more pure virtual (abstract) functions. A pure virtual function is one which must be overridden by any concrete (i.e., non-abstract) derived class. This is indicated in the declaration with the syntax ” = 0” in the member function’s declaration.

In general an abstract class is used to define an implementation and is intended to be inherited from by concrete classes. It’s a way of forcing a contract between the class designer and the users of that class. If we wish to create a concrete class (a class that can be instantiated) from an abstract class we must declare and define a matching member function for each abstract member function of the base class.

Stylistic rules
• every class is in its own file of its own name
`Animal` → `Animal.cpp`
• filenames and header files should have different extensions (cxx, cpp, cc and hxx, hpp, hh)
• class names should start with uppercase letters:
`Animal`
• object names should start with lowercase letters:
`Animal fido("Fido")`
• for longer names use `CamelCase` or `Under_Scores`
• constants should be written in capital letters
• avoid `#defines` use `const`
• use namespaces, at least one for your project
• namespace should/could match the folder name
• avoid multiple inheritance
• you can have your own naming rules but be consistent
• use the new C++ features, at least C++11, best until C++17
• C++17 is almost completly implemented in all the major compilers (gcc, clang and Microsoft MSVC and Apple Clang)
• compile your code not in the source directory but in a directory like `build`
• add comments where neccessary
• document your code with a code documenting system like doxygen or our own mkdoc.py
• short license & author statement at the beginning of file

Input / Output
• iostream – interacting with the terminal
– std::cout writing to the terminal
– std::cin getting input from the terminal
– std::cerr writing to the error channel immediately
– std::clog write to the error channel buffered
– with small programs there is no difference
– with larger programs and in a pipe difference becomes important
– example: my latex compile pipeline would stop if I write on cerr ...
• fstream – principle file stream
– ofstream – output file stream
– ifstream – input file stream

Opening a file
Syntax:
```
void open(const char *filename,
ios::openmode mode);
```

flags:
• ios::app – Append mode. All output to that file to be appended to the end.
• ios::ate – Open a file for output and move the

read/write control to the end of the file.
• ios::in – Open a file for reading.
• ios::out – Open a file for writing.
• ios::trunc – If the file already exists, its contents will be truncated before opening the file.

Regex Syntax I
• characters:
– exact matches:
x q ATG SS2010
• metacharacters:
– characters with special meaning:
[ ] . ? * + | { } () \^ $
• metasymbols:
– sequences of characters with special meaning:
\s (space character) \t (tab stop) \n (newline) \w (word character)
– wordcharacter a-z, A-Z, 0-9, including the _ (underscore)

The terminal tool grep:
```
$ grep -E "regex" filename(s)
$ egrep "regex" filename(s)
```
• regex regular expression (pattern)
• switch -E allows extended mode with regular expressions
• quotes protect special characters like pipes (|)
• egrep is like an alias for grep -E

• "A"... standard IUPAC one-letter codes
• "." position where any aa is accepted
• "[ALT]" ambiguity any of Ala or Leu or Thr
• "[^AL]" negative ambiguity any aa but not! Ala or Leu.
• ".{2,4}", "L{3}" repetition. two to four amino acids of any type, exactly three Leu
• "L{2,4}" two to four Leu, why not :)
• "+" one or more, X+ == X{1,}
• "*" zero or more, X* == X{0,}
• "?" zero or one, X? == X{0,1}
• "^M" N terminal Met
• "A$" C terminal Ala

#include <regex>
C++: regex_search(string, pattern,flag(s))
C++: regex_replace(string, pattern, replace)

Friend function:
A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.
A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

General considerations
1. Friends should be used only for limited purpose.
2. Too many functions or external classes are declared as friends of a class with protected or private data.
3. Friends lessens the value of encapsulation of separate classes in object-oriented programming.
4. Friendship is not mutual. If a class A is friend of B, then B doesn't become friend of A automatically.
5. Friendship is not inherited.
6. The concept of friends is not there in Java.

A copy constructor is a member function which initializes an object using another object of the same class. A copy constructor has the following general function prototype:
`ClassName (const ClassName &old_obj);`
If we don't define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member-wise copy between objects. The compiler created copy constructor works fine in general. We need to define our own copy constructor only if an object has pointers or any runtime allocation of the resource like file handle, a network connection ...etc.

Every object in C++ has access to its own address through an important pointer called this pointer. The this pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.
Friend functions do not have a this pointer, because friends are not members of a class. Only member functions have a this pointer.

When to use the this pointer
• When local variable's name is same as member's name:
`this->x = x;`
• To return reference to the calling object:
`return *this;`

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.
Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

```
Box operator+(const Box&);
```

declares the addition operator that can be used to add two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions.

In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows:

```
Box operator+(const Box&, const Box&);
```

Namespaces:
The namespace keyword allows you to create a new scope. The name is optional, and can be omitted to create an unnamed namespace. Once you create a namespace, you'll have to refer to it explicitly or use the using keyword. A namespace is defined with a namespace block.
[Wikibooks — C++ Programming / Namespace, 2019]

a namespace is a context for identifiers. C++ can handle multiple namespaces within the language. By using namespace (or the using namespace keyword), one is offered a clean way to aggregate code under a shared label, so as to prevent naming collisions or just to ease recall and use of very specific scopes. ... Use namespace only for convenience or real need, like aggregation of related code, do not use it in a way to make code overcomplicated ..

The using directive you can also avoid prepending of namespaces with the using namespace directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the following code.
[Tutorialspoint — C++ Namespaces, 2019]

Namespace aliases allow the programmer to define an alternate name for a namespace.
They are commonly used as a convenient shortcut for long or deeply-nested namespaces.
[cppreference.com — Namespace aliases, 2019]

Namespace summary
• namespaces allow to structure and organize code
• avoid name clashes for variables, functions and classes
• you can put related functionality in the same namespace
• namespace can be extended over several files
• you can import namespace qualifiers into the current scope
• abbreviate long namespace names using aliases
• using namespaces is recommended for not so small projects
• declare functions inside of the namespace, define them outside
• place namespaces into a folder of the same name if they consist of several files or in a file of the same name if they span only one file
• c++ namespaces are much more convenient than R ones:
– no automatic import of all functions
– easier generation and nesting

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.
A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept. There is a single definition of each container, such as vector, but we can define many different kinds of vectors for example, vector <int> or vector <string>.
[Tutorialspoint — C++ Templates, 2019]

The Standard Template Library (STL) is a software library for the C++ programming language that influenced many parts of the C++ Standard Library. It provides four components called algorithms, containers, functions, and iterators.
The STL provides a set of common classes for C++, such as containers and associative arrays, that can be used with any built-in type and with any user-defined type that supports some elementary operations (such as copying and assignment). STL algorithms are independent of containers, which significantly reduces the complexity of the library.
The STL achieves its results through the use of templates.
[Wikipedia — Standard Template Library, 2019]

Sets vs Vector vs Maps
• std::set use is rather limited, if you need no values ok
• std::set is faster if inserting multiple items
• std::set is used if items should be kept ordered
• use sets only if performance is very critical
• we can stick with std::vector and std::map in most cases
• btw: std::array is an unextensible (std::vector)
• std::multimap can be emulated as std::map containing std::vector(s)
→ `multimap<x, y>` is similar to `map<x, vector<y> >` but are slightly easier to handle during loops etc.
• containers of interest: `std::any`, `std::pair` (map loops)
→ Hint: Stick mostly with `std::vector` and `std::map`!

Tuple and Pair (new)
• std::pair - two values of same or different types (special tuple)
• std::tuple - two or more values of same or different types
• tuples can be used to return more than one value

Matrix / Graph project
→ aim: create a class to handle undirected graph
→ basis data structure should be an adjacency matrix based on a nested vector container
→ Graph methods:
• void addVertex(string node)
• void addEdge(string node, string node)
• vector<int> degree()
• int degree(string node)
• float density()
• bool adjacent(string node, string node)
• int pathLength (string node, string node)

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero. Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.
[Tutorialspoint — C++ Exception Handling, 2019]

• try – A try block identifies a block of code for which particular exceptions will be activated. It’s followed by one or more catch blocks.
• throw – A program throws an exception when a problem shows up. This is done using a throw keyword.
• catch – A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an
exception.

Memory in your C++ program is divided into two parts:
The stack: variables declared inside the function will take up memory from the stack.
The heap: This is unused memory of the program and can be used to allocate the memory dynamically when program runs.
Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.
[Tutorialspoint — C++ Dynamic Memory, 2019]

You can use the stack if you know exactly how much data you need to allocate before compile time and it is not too big. You can use heap if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data.

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called new operator.

If you are not in need of dynamically allocated memory anymore, you can (should!!) use delete operator, which deallocates memory that was previously allocated by new operator.

unique_ptr is one of the Smart pointer implementation provided by c++11 to prevent memory leaks. A unique_ptr object wraps around a raw pointer and its responsible for its lifetime. When this object is destructed then in its destructor it deletes the associated raw pointer. unique_ptr has its -> and * operator overloaded, so it can be used similar to normal pointer.

shared_ptr is a kind of Smart Pointer class provided by c++11, that is smart enough to automatically delete the associated pointer when its not used anywhere. Thus helps us to completely remove the problem of memory leaks and dangling Pointers.
It follows the concept of Shared Ownership i.e. different shared_ptr objects can be associated with same pointer and internally uses the reference counting mechanism to achieve this.

Summary Smart Pointers
• !prefere containers like vector or map!
• !prefere (const) references in function arguments!
• use smart pointers instead of new and delete
• use new and delete only if you deal with foreign raw pointers or if you create your own container
• R.20: Use unique_ptr or shared_ptr to represent ownership
• R.21: Prefer unique_ptr over shared_ptr unless you need to share ownership
• R.22: Use make_shared() to make shared_ptrs
• R.23: Use make_unique() to make unique_ptrs
• make snippets

main mess conclusions ...
• During the development of the `IMatrix` class or the dutils namespace a lot of test code accumulates in `main`.
• It would be better to have a separated test approach instead.
• This approach ist called Unit-Testing.
• In its extreme form you write tests first, only thereafter start coding. → Extreme Programming.
• In comparison to other PL, C++ has no default Unit-Test library.
• There is `cppunit` designed after the Java library `JUnit`.
• There are many alternatives in C++.
• Unit testing ensures code integrity during development.
• In larger companies whole departments are responsible for writing the tests.

Test-driven development (TDD) is a software development process relying on software requirements being converted to test cases before software is fully developed, and tracking all software development by repeatedly testing the software against all test cases. This is opposed to software being developed first and test cases created later.

Package installations
• header only libraries (just download and place the file)
• few file libraries (just download and place the files)
• more complex libraries:
– use OS package manager to install (dnf, apt, brew, pacman, winget(?))
– use C++ package manager (cget, conan, vcpkg)
– download and build yourself (sometimes tricky, last choice)

Install packages from source
Single header files:
• just download and place the header file into your source directory
• use `#include "path/to/header.hh"` to use it
Larger packages:
• fetch and unpack the source
• switch into source directory
• run configure or cmake or make
• run make
• run make install

GUI Libraries
• there are some mainstream libs like GTK and Qt
• quite heavyweight if you just like to wrap a small tool
• you then deliver several files often weighting some dozen Mb
• alternatively you can use a more lightweight alternative to wrap an existing applications
• example for this: `https://www.fltk.org`
• applications will be statically linked and should be always less than 1 MB
• fltk did not encourages dynamic linking (so/dll) as this creates at some time dependency problems
• library is that small, that it can be statically linked
• install is than just a copy operation of the executable