3
Structure Behavior with Functions

We have already seen several functions so far, like `main()` and you might have written also functions taking stream objects as arguments for your unit tests. This chapter will summarize all the basics around functions, parameters, return values and arguments. A further topic is the scoping and lifetime rules of local variables and temporaries used as arguments or return values. We will see, how a single function name can be overloaded by defining different versions of the function with different number or types of parameters. You will get a first glimpse of an often vexing issue: function lookup, or how a compiler determines which function overload to actually call. And also we contemplate the means to handle error situations, when a function can not work as it guarantees including throwing and catching exceptions.

3.1 Defining Simple Functions

While in some cases discovering the structure that your piece of software, its design, can be hard, I'd like to give you some guidelines on how to recognize potential for improvement.

- A good function does one thing well and its name reflects that.
- A good function only has up to about three parameters (with some exceptions with a maximum of five).
- A good function consists of only a few lines and does not have a deeply nested control flow.
- A good function gives the caller a guarantee that it returns a useful result.
- An easy to use function accepts all possible input values as defined by its parameter's types. If a function cannot fulfill its task with an argument given, it provides a consistent error reporting mechanism or a default return value.

Whenever a function's body consists of too many statements, so that you feel the urge to separate the lists of statements into sections lead by a comment explaining what the following code should do, then it is time to split that function up. Within Eclipse CDT you

can select the code block and use the "Extract Function" Refactoring to move it into a new function instead. The comment you wrote or wanted to write is often a good indication on how to name that extracted function. Having shorter more focussed functions also allows for easier testing. See the following code for an example function that is in need for refactoring. Can you spot the error in the code at once?

```
bool writeNTimesToFile( std::string filename
                      , std::string content
                      , size_t n){
  //open the file
  std::ofstream file{filename,std::ios_base::app};
  // check if it is OK and then write
  if (! file.good()) return false;
  // write content n times
  while (--n>0){
    // write the content on a line
    file << content;
    file << '\n';
  }
  // write another new line and flush it
  file << std::endl;
  return file.good();
}
```

While this is not an extraordinary complex example it already shows, that combining too much functionality in one function can easily lead to hard-to-test code. Extracting some of the functionality can result in a unit-testable function and one that is relying on the environment. The second function can only be tested indirectly, by its effect on the environment and such a test is often brittle and can fail spuriously, when something in the environment changes. Note, that I also removed the comments explaining the code, since it now should be more obvious what is going on. The extracted function `writeNTimesToStream()` looks like this and can be unit tested. I leave that as an exercise.

```
void writeNTimesToStream( std::ostream& file
                        , std::string content
                        , size_t n) {
  while (--n > 0) {
    file << content << '\n';
  }
}
```

Now the second version of our `writeNTimesToFile()` function has become simpler, but is still untestable, because it relies on the program's environment. But it also contains conditional logic and thus would require two tests to actually test both paths through it. Writing such a test is hard, since it would require to fiddle with the file's

permissions or providing an invalid file name to actually trigger the error exit from the function.

```
bool writeNTimesToFile2( std::string filename
                  , std::string content
                  , size_t n){
  //open the file
  std::ofstream file{filename,std::ios_base::app};
  // check if it is OK and then write
  if (! file.good()) return false;
  writeNTimesToStream(file, content,n );
  file << std::endl;
  return file.good();
}
```

So we need to split that up further to separate the environment-dependent part, that is opening a file, from the environment independent part that operates on the interface of `std::ostream` only. With that we can test the logic.

In addition our function above function begins with a so-called *guard statement* that checks its precondition that the stream is in good shape and returns early if not. However, in our case, removing the premature function return is actually beneficial as shown below by reversing the condition and putting the dependent statements in a block depending on that condition. Still both paths have to be tested.

guard statement

```
bool writeNTimesToStreamIfOk( std::ostream& file
                            , std::string content
                            , size_t n){
  if (file.good()) {
    writeNTimesToStream(file, content,n );
    file << std::endl;
  }
  return file.good();
}
```

The remaining environment dependent version is so simple, that you might consider it not worth testing. It's body consists of two lines without conditional logic. One line calls out to the standard library and the second line calls a (potentially) fully-tested function.

```
bool writeNTimesToFile3( std::string filename
                       , std::string content
                       , size_t n){
  std::ofstream file{filename,std::ios_base::app};
  return writeNTimesToStreamIfOk(file,content,n);
}
```

As an example, on how you now can test the logic of a bad stream, see the following code piece.

```
void testWriting3TimesIfNotOk() {
  std::ostringstream out;
  out.clear(std::ios_base::badbit);
  bool ok = writeNTimesToStreamIfOk(out,"hello",3);
  ASSERTM("writing should have failed",not ok);
  ASSERT_EQUAL("",out.str());
}
```

As an exercise please complete the unit tests for the code created. Also try to write test functions for the overall functionality that is actually writing to a file and is thus environment dependent. Consider the following questions:

- Can you ensure that it will always succeed?
- Can you run it two times in a row?
- What are potential issues making it fail, even though the code is correct?
- What are issues that such tests succeed, even though the code is wrong?

Syntax

classic function definition

There are several syntactical means to define a function. The *classic function definition* syntax, where the return types comes first, like here:

```
double square(double value) {
  return value*value;
}
```

And one can use **auto** as the return type and have the compiler deduce the return type as it happens with lambdas. That style was introduced mainly for template functions where the return type is depending on the concrete type of function arguments. To make the type deduction work, one needs to put the complete function definition before it is used. This requires inline implementation in header files. However, before C++14 or when there are multiple return statements that differ in their return expression type, **auto** as return type required to specify the return type after the parameters using an "arrow" symbol (->). This way it can refer to the arguments of a template instantiation. This is used together with the mechanism of **decltype** that allows to determine the type of an expression without evaluating that expression. For example, **decltype**(3.14) will be equivalent to **double** because that is the floating point literal's value.

decltype

```
template <typename T>
auto square(T value) -> decltype(value*value) {
  return value*value;
}
```

This example, a generic function template (see chapter **??**), provides

trainling return type

a useful application of the *trainling return type*. For types, where the multiplication operator is overloaded, squaring a value might return a different type than the parameter type itself. For example, the inner product of two mathematical vectors delivers a scalar value. For functions that are not to be defined in header files, I recommend that you stick with the classic return-type-in-front version and refrain from using **auto** as return type, except for inline, template, or constexpr functions, that are to be defined in header files.

The trailing return type can also be used with lambda expressions, if the return type deduction would either not work due to multiple return statements or if the deduced type would not be the one that you'd expect as shown in the following example, where ::toupper would return an int.

```
auto capitalize=[](char c)->char { return ::toupper(c);};
```

Parameters

Parameters are given in parenthesis after the function name, by specifying their type and optionally their name. Within the function body, a named parameter can be used to access the argument passed when the function is called. When a function doesn't take any parameters the parenthesis still have to be given. Later we will see how to deal with variable number of arguments in a type-safe way. More information on the type of parameters are given below in section 3.3.

> C spells empty parameter lists like `f(void)`. While C++ compilers will accept that syntax, I do not recommend it. If there is no parameter, show that by putting parentheses together, like `f()`. The reason, why the old C-style was introduced stems from a time where another function parameter syntax was allowed in C. C++'s type safety and function overloading rules no longer interpret `f()` the way C does. It is even proposed for future standardization to allow **void** as a regular type in generic functions to avoid special-cases for it.

function declaration

A *function declaration* that proposes the existence of a function just ends with a semicolon without providing a function body. Such declarations are typically provided in a header file to allow calling the function after its inclusion. An implementation of that declared function is then in a separate `.cpp` implementation file with its definition or in an already compiled library or object file that the linker later integrates with the code calling the function.

function definition

A *function definition* provides a function body, that typically consists of a block of statements enclosed by curly braces {}. However, there is also the possibility to create a function body consisting of a

try-catch-block for handling all exceptions that might occur within the functions code (see section 3.4).

inline function

For very short functions, it can be beneficial to define the function in the header file directly as an *inline function*. You precede the function definition in the header with the keyword **inline** which prevents violating the *one-definition-rule* if that header file is compiled multiple times by including it in several compilation units. For function templates (see chapter **??**) and member functions implemented within their class definition **inline** is implied.

A function that has a non-**void** return type must provide a **return** statement setting the value of the function. [1] The **return** statement always finishes execution of the function and returns to the function's callers. Some people use parentheses around the returned expression, like `return(5);`. I do not recommend that, because it can be easily confused with a function call, just write `return 5;`. Relying on the initializer syntax, one can use braces around a value returned to mark eventual automatic conversion of the value given to the actual defined return type as shown below. This syntax will avoid narrowing conversions as it would do within a variable definition and removes the burden to repeat the return type, if the value needs to be constructed at the return statement.

[1] With the exception of the `main()` function.

```
std::vector<int> demoReturnBraces(){
  return {1,2,3,4,5};
}
```

From C++14 on, explicit or implicit inline functions can use auto as their return type and have it deduced from the return statement(s) in the function's body. However, if there are multiple return statements, all must agree on the type of the returned expression.

3.2 *Visibility and Lifetime*

This section explains a lot of theory. If you already program in a similar language, you might skip most of it on a first reading, because most of the things work like you would expect them.

Lifetime of Variables and Values

One of the differences of C++ with other languages like Java™ is its deterministic lifetime model. It is exactly defined how long a value or a variable lives, when it is destroyed and its resources released[2]. For example, a local variable comes to live at the statement where it is defined and initialized (forgetting initialization is most of the time a severe error). It is destroyed, when the block where the variable definition takes place is left, either by the execution passing the block's end }, leaving a loop or switch-statement's block with the statement **break**; or an inner block of a loop-statement with **continue**;, **return**ing from the function or when a **throw**n exception forces execution to leave the block[3].

[2] This automatism works as long as you rely on the practices presented in this book. If you actually want, you can make your program misbehave.

[3] Or when a `goto` statement leaves the block, but we do not use that kind of control flow mechanism.

Global Variables

Global variables are possible in C++ but I tell you they are *verboten*. The lifetime of a global variable starts before `main()` is called and it ends when the program ends. However, the time when such a global variable is actually initialized, depends on its type. "Trivial" and literal types can be initialized at compile time, but other types might require initialization at run-time. If a program consists of several compilation units the order of initialization across compilation units is not defined by the C++ standard. That alone makes another reason to avoid globals. I have been bitten in the past, by global variable initialization dependency ordering that went wrong, when code was put into shard libraries (aka DLLs). Just DON'T do it. With multi-threading protecting global variables from data races and thus undefined behavior is even a worse problem.

Caution 3.1: sidestep global variables

The Keyword `static` for Local Variables

The *storage class* **`static`** can be used to define local variables in a function that have a lifetime until the end of the program. A local **`static`** variable is initialized the first time when a function is called. However, like a global variable, such local **`static`** variables make functions hard to test and also generate big problems, when the function is called concurrently by multiple threads. To avoid undefined behavior all uses of such a function require mutual exclusion when accessing the **`static`** variable. If your function requires some kind of enduring memory, better place this function into a class and use a member variable of that class as memory that has a lifetime longer than a single function call.

Caution 3.2: use class members instead of local `static` variables

There are a few special cases with respect to the lifetime to remember and where C++ might be different from languages you know. For example, a variable defined as part of a **`for`**-statement will live until the end of the loop, but not after.

In addition to the lifetime of variables, you must also consider the lifetime of *temporary* values created by an expression. For example, a function argument might result from an expression and its temporary value passed as the argument to the function call lives until the function returns. In general one can say that a value of an expression lives until calculation of the enclosing *full-expression* is completed. That means, temporaries created during the calculation live as long as they are actually needed. A full-expression is one that is not a part of another expression, *e.g.*, the condition part in a **`for`**`(;`*condition*`;)`.

temporary

full-expression

If you initialize a reference with a temporary value, then the tem-

porary's lifetime is automatically extended until the end of the lifetime of that reference. In section 3.2 below, you will learn about the rules which references can bind to a temporary value.

Visibiltiy and Scoping

visibility

Closely related to the lifetime of a variable is its *visibility*, that means, where you can actually refer to the variable in the program by its name. Any name defined in an outer block is accessible in a block of a function, unless that inner block defines another variable with the same name. Unless that "inner" variable with the same name does not end its lifetime, the outermost variable can not be referred to in the code. After that the name again refers to the previously visible variable. As an example, have a look at the following function that (re-)uses the name `i` several times within the same function.

```
void showScopingRules(int i, double d){
  unsigned j{1}; // can not use name i instead of j
  std::cout << i<< "\n";
  {
    char i{'d'}; // shadows parameter i
    // parameter i not accessible but d is
    std::cout << i << " " << d << "\n";
  }
  ++i; // that is the parameter no longer shadowed
  for (unsigned i=0;i<j;++i){ // another i
    std::cout << i<< "\n";
  }
  std::cout << i << "\n"; // parameter i again
}
```

shadowing

The *shadowing* of names by using the same name in an inner scope is another reason to prefer shorter functions. There is just less chance to inadvertently redefine a variable name.

scope

Visibility and scoping is not only related with variables, but with all named elements of a C++ program. The range of visibility where a language element is defined is also called its *scope*. Scopes can nest and unless an inner scope redefines the name it is also visible in the inner scope. You can not define a language element with the same name twice in the same scope. This rule explains, which in the case of variable `j` in the previous example, we couldn't use `i`.[4]

[4] Try this as an exercise to observe the compiler messages

namespace-level scope

The scope outside other definitions is called *namespace-level scope*. We must define functions at namespace-level scope and will define most classes and types at this level as well. Variables defined on namespace-level scope are global, so you don't define variables there, only functions and classes.

scope operator

We haven't defined our own namespaces yet, but we already used the namespace called `std` to refer to elements within that namespace by writing `std::` in front of it. That syntax of the *scope operator* with the double colon `::` is provided to access elements from a named

scope.

In addition to the namespace `std` we have also used the *global namespace* that doesn't have a name by itself. But we can refer to elements in the global namespace by preceding them with `::`, *e.g.*, `::std::cout`. Our example function could be called like that:[5]

global namespace

```
::showScopingRules(42,3.14);
```

[5] Output:
```
42
d 3.14
0
43
```

We haven't learned much about defining classes yet, however, every member defined or declared in a class is said to be in class scope and you access such a member later on by `classname::`, the same syntax as with namespace members. More details on the scoping related to classes will be given in chapter 4.

class member

Namespaces can be defined on *namespace*-level scope within other namespaces and referring to such an inner namespace requires multiple scope operators `::` if done in another scope. If a class is defined within a namespace and you want to refer to a member of that class you add another `::` such as in `::std::vector<int>::size_type`. The latter syntax specifying the start of the namespace/class nesting in the global namespace is only required, to circumvent ambiguities, *e. g.*, when you refer to a name that is available in current scope and also as an existing name in the global namespace. As an example of the syntax see namespacesyntax.cpp (Listing 3.1).

namespace

Listing 3.1: namespacesyntax.cpp

```
namespace demo{
void foo(); //1
namespace subdemo{
void foo(){/*2*/}
} // subdemo
} // demo
namespace demo{
void bar(){
  foo(); //1
  subdemo::foo();//2
}
}
void demo::foo(){/*1*/} // definition
int main(){
  using ::demo::subdemo::foo;
  foo();//2
  demo::foo();//1
  demo::bar();
}
```

You can make your life easier, if you want to use a name from a foreign but named scope, *i.e.*, a namespace or a class, you can enter its name in the current scope by a *using declaration*. This looks a bit like an alias declaration (see page 59), that only works for types and templates, but doesn't define a new name. For example, `using std::string;` at the beginning of a function body allows us to refer to class `std::string` without the namespace scope prefix just as `string`

using declaration

. A using declaration shouldn't be placed in the global namespace, because it subverts the namespace mechanism that was introduced to avoid name clashes of program elements from different origins, *e.g.*, different libraries used. A using declaration of a name with multiple declarations, *e.g.*, an overloaded function (see below, page 85) or a class template with specializations (see chapter **??**), will introduce all elements with the same name to the current scope.

anonymous namespace

There is a special case the *anonymous namespace*. If you define a namespace without giving it a name in a compilation unit it hides all names defined in that namespace from all other compilation units. So a function defined in an anonymous namespace can not be called from any other source file than that, where the function itself is defined. Anonymous namespaces are a replacement for the keyword **static** that was used to define global entities to be visible only in the current compilation unit in C. There is only one good use for the keyword **static** left in C++: class members. See more on that in chapter 4.

```
#include <iostream>
namespace { // anonymous
void doit(){ // can not be called outside this file
  std::cout << "doit called\n";
}
} // anonymous namespace ends
void print(){ // callable from other parts if declared
  doit();
  std::cout << "print called\n";
}
```

As you can see, within a single compilation unit, names defined within the anonymous namespace are directly available in the compilation unit, as if a **using namespace** directive would be given. This is implicitly done for the anonymous namespace, because otherwise one would not be able to address the elements defined there.

With the above definitions, we can try to call the functions in another file. But the attempt to call the function from the anonymous namespace fails with a linker error, because names defined in the anonymous namespace aren't available outside the compilation unit where they are defined as shown below.

```
void caller(){
  void print(); // declare print
  print();
  void doit(); // declare doit
  //doit();      // linker error
}
```

You can use the anonymous namespace to hide auxiliary functions and types as well as constants that you need within a module, but that you do not want to expose to the outside users of your module.[6] With that, any changes to those internal structures do not impose

[6] Note: We do not define non-const variables on namespace level!

using namespace

A **using namespace** directive in global scope is considered harmful. Do not use it, even if many examples you see put **using namespace** `std;` before `main()`. It works like you would spell out a using declaration for every element defined in the namespace, which are many and may be even more in the future. If you can make your program more readable by omitting the namespace scoping, then explicitly import only the names you use with a using declaration. Under no circumstances place a **using namespace** `std;` in a header file. The Cevelop IDE provides a refactoring to rid your code of such using directives.

Caution 3.3: no global ***using namespace***

any changes to your module's clients. A second application of the anonymous namespace occurs in headers, where you want to define a special entity for each compilation unit it is `#included` in without violating the one-definition rule. It is quite rare that you would want to apply an anonymous namespace in a header file, because for function definitions, you can use **inline** to circumvent ODR's restrictions.

References

Understanding the lifetime of a variable or value is important, because C++ allows to use and pass references to variables and values. If your program is keeping and using such a reference beyond the lifetime of the referred to variable or value, your program has undefined behavior. Most of the time, everything that you do with references is automatically correct, but there are situations where you might end up obtaining a reference to a "dead" variable, for example, when returning a reference to a local variable from a function as in the following code piece. Fortunately, a compiler will typically issue a warning about such inane code, so always consult the warnings you get, even when your code seems to be compiled successfully.

```
int & doNeverDoThis_ReturnReferenceToLocalVariable(){
  int number{42};
  return number;
}
```

We already came across the terms *lvalue* for variables and *rvalue* for values. A value with a name or an explicit memory location, such as an element in a `std::vector` can be an lvalue, because you can assign a new value to it. Often an rvalue denotes a temporary calculated from an expression. This distinction gets even more complicated when references to such values come into play. There are three useful kinds of references and these are most often used as function parameter types.

- *lvalue references*, like `std::ostream &`, that refer a variable where you imply to have a side-effect on ,
- *const references*, like `int const &`, that can also refer a temporary and prohibit any side-effect, and
- *rvalue references*, like `std::unique_ptr<int> &&`, that enable so-called *move semantic* and *perfect forwarding* (see chapter **??**).

move semantic

Reference types play an important role in parameter definitions and a lesser role when defining variables or return types. Since a reference is just an alias for an already existing thing, it is important that a reference is never used after its original's lifetime expired. Since const-references and rvalue-references can refer to temporaries, there is a special rule extending the life-time of such a temporary beyond the full-expression creating it up to the lifetime of the reference. However, that life-time extension of a temporary is limited to the current scope and the reference directly initialized with the temporary. If you would return such a reference bound to a temporary object from a function, you would return a dangling reference resulting in *undefined behavior*. If you define a reference that is not a parameter to a function, you must initialize it, when it is defined. There is no such thing as a *null-reference* that you might know from Java™ .

As a syntactical but impracticable example for local references look at the following piece of code

```
int i{42};
int &ri{i}; // must initialize ref
int const &cri{i}; // const alias
int const &cr{6*7}; // extend lifetime of 6*7
ri = 43; // changes i, ri only an alias
//--cri ; // doesn't work -> const
int &&rvr{3*14}; // extends lifetime of 3*14
//int &&rvri{i}; // impossible
int &&rvri{std::move(i)}; // steal i's content
```

I hope, lvalue reference and const references are not that much of a mystery, after you learned about their use in parameter passing below. However, new from C++11 on are rvalue-references that are introduced with && after the type in front of the reference's name. Rvalue-references are a language feature that allows to implement deliberate optimization and can be used to rip out the guts of an object when the variable referring the object is no longer used afterwards. This move-semantic is used by classes in the standard library who manage resources like `std::vector`. For the built-in trivial types move is the same as copying, so often you do not need to think about a difference. But a moved-from variable can only be re-assigned or end its lifetime, like variable `i` after it has been used to initialize `rvri` with `std::move(i)` above. You must consider a moved-from variable a kind of an "undead" that a vampire sucked all its blood from. This metaphor is easy to remember, just in your mind draw a mouth over

rvalue-reference's symbol && covering the loops of the ampersands and you will see Dracula's teeth.

A second feature of rvalue references is the means to define types that can only be moved and not copied. We already learned about the situation of non-copyable objects, when we used `std::ostream &` as parameter to pass `std::cout` to a function. A subtype of `std::ostream`, the stream relating to a file `std::ofstream` is such a *move-only type*. This allows us to return a `std::ofstream` from a factory function opening the file. Returning a reference to a local variable of type `std::ofstream` would mean a dangling reference and copying is impossible with all stream objects, because the base class prohibits that for good reasons. Returning a copy will not work, because the base type is not copyable. But `std::ofstream` is move-only and it can be returned from a function with moving its guts to the result.

move-only type

```
std::ofstream fileFactory(std::string filename){
  std::ofstream theFile{filename};
  return theFile;
}
```

This allows to pass values of these types to functions or return them from a function without the need to use pointers. The move semantic guarantees that there is only one instance alive, so the bad things why copy is verboten cannot occur, because there is no copy of the object. There is, however, the drawback that the concrete type `std::ofstrream` must be used as return type and one cannot hide it behind its abstract base type `std::ostream`. [7]

[7] If you have no background in object-oriented programming refer to chapter **??**.

3.3 Parameter Passing

Using references for function parameters that are potentially large objects helps avoid the cost for copying the argument. For example, passing a `std::vector<double>` with a million elements to a function by value (not as a reference), can mean copying eight megabytes. If such a function call happens very often in your program and the compiler cannot optimize away that copying it can be a significant overhead[8]. If your function needs to have a side effect on a variable passed as an argument, as we have seen with the functions taking stream objects as parameters, you'll have to pass that argument as an lvalue reference.

[8] Often that overhead is overemphasized, because in many cases an optimizing C++ compiler can elide the copying, especially if an argument is a temporary. Don't be afraid of passing containers by value. If your code is slow, measure first if unnecessary copying is really the culprit.

```
void printValue(std::ostream &out,int value)
```

Pass by Value

If you just name a plain type without a reference symbol for a parameter your function can consider the name like a local variable of that type. This normal mode of parameter passing is called *pass by value*. Any changes to the parameter within the function have no effects to the caller. You can call a function with pass by value with a

[9] This copy is often elided by the compiler, as a means of allowed optimization

temporary as argument that is copied to the variable space reserved for the function parameter.[9]

Pass by value should be the first and major means to consider when passing simple and small types to a function, especially when the function parameter has to be mutated within the function's body.

As we already have seen for return types, if that pass-by-value parameter type is a move-only type, passing by value means stealing the innards of the argument. That is OK, if it is a temporary value, which will be destroyed anyway after the call. However, if you want to pass a variable of such a type, you need to tell the compiler that you no longer care about its content on the calling site by wrapping the variable with `std::move()` as is shown in factoryfunction.cpp (Listing 3.2) where `std::ofstream` is such a move-only type. Note, that you cannot used `std::ostream` in that case, because that type is abstract and can neither be moved or copied.

Listing 3.2: factoryfunction.cpp

```
#include <fstream>
#include <iostream>
std::ofstream fileFactory(std::string filename){
  std::ofstream theFile{filename};
  return theFile;
}
void writeToFile(std::ofstream file){
  file << "hello, world\n";
}
int main(){
  std::ofstream out=fileFactory("hello.txt");
  out << "hello\n";
  writeToFile(std::move(out));
  // cannot use out here anymore
  if (out << "more?"){
    std::cout << "oops out should be empty now\n";
  }
}
```

For small values like **`int`** or **`char`** pass by value is best. If you intend to not change the parameter's value in a function you should add **`const`** in front of its name. As with variable definitions, this should be the default. A return type should almost always be a non-const value type. Pass by value is also often needed for recursive functions, where each function call requires its own copy of the arguments when they need to change.

Pass by Const-Reference

[10] I use *const-ref* as a short form of const-reference.

The safest and most common parameter passing style is *pass by const-ref*[10]. This allows a caller the freedom to specify arguments that are temporaries, while still have the efficiency of no-copying when large objects are passed. The only drawback you get from passing by const-ref is that you cannot change the parameter's value within

the function. Unless you need to change or return the argument value, for generic code[11] passing by const-ref should be your default, because that works with all parameter types and argument values—even temporaries—and imposes no overhead. Therefore, if in doubt, use pass by const reference, like the following example.

[11] see chapter **??**

```
std::string serialize(std::vector<int> const &v);
```

As with pass by value, pass by const-ref is without problems for recursive functions, since no unintended change can happen and you can easily pass a new value as a temporary down the call hierarchy.

Both call by value and call by const-ref share the property that arguments are automatically converted to the parameter type like it would be on an assignment. For example, by *integral promotion* or *automatic numeric conversion*. If the function name is overloaded this can lead to ambiguities.[12]

[12] See chapter 4 on page 137 for more on automatic conversions.

For example, the code in Listing 3.3 shows two calls that won't compile, because of such an ambiguity. The value `10u` is of type **`unsigned`** and can automatically converted to both **`int`** and **`double`** without either conversion being preferred, so the call to `factorial(10u)` is ambiguous. The same is true for the call with `1e1l` which is of type **`long double`**. That value of `10.0l` again can both be converted to **`double`** and **`int`** by truncating it so the second call in function `demoAmbiguity()` is also ambiguous. If you look closer, you can see that the explicit construction of an **`int`** from the **`double`** value `n` with **`int`**`(n)` in function `factorial(`**`double`**`)` selects the previously defined function `factorial(`**`int`**`)`.

Listing 3.3: factorial.cpp

```
int factorial(int n){
  if (n > 1) return n * factorial(n-1);
  return 1;
}
double factorial(double n) {
  double result=1;
  if (n < 15)
    return factorial(int(n));
  while(n > 1) {
    result *= n;
    --n;
  }
  return result;
}
void demoAmbiguity() {
  factorial(10u); // ambiguous
  factorial(1e1l);// ambiguous
  std::cout << factorial(3) << "\n";
  std::cout << factorial(1e2) << "\n";
}
```

Pass by Reference

We have already seen the need for pass by non-const reference earlier for stream objects. Such a reference parameter can only be filled with an lvalue argument, usually naming a variable at the call site.

In case of the stream objects it is also wise to define the function with the side effect to return the stream reference. This is one of the few cases where returning a reference is safe, since it must have a lifetime at the call site that is longer than the function call itself[13]. For stream objects it allows the call site to chain I/O requests like with the shift-operators and also check the validity of the stream immediately without referring to the variable again. You can see an example for that in returnreference.cpp (Listing 3.4)

[13] That is because it cannot be a temporary object

Listing 3.4: returnreference.cpp

```
std::ostream &print(std::ostream &out, int value){
  out << value;
  return out;
}
void usePrint(){
  if (print(std::cout,42)){
    std::cout << " printed OK\n";
  } else {
    std::cerr << "couldn't print on cout\n";
  }
}
```

Remember, a reference is always an alias to some existing object. A reference parameter just used within a function will always be valid in C++, since there can not be a "null" reference. However, if you happen to pass that reference to another thread, or store it in an object with a longer lifetime than the referred to variable you risk a dangling reference and in case of another thread using the same object you also introduce a potential data race. Both these effects are *undefined behavior* and must be avoided.

In other languages parameters passed by reference are also called in-/out-parameters. In this case they not only pass a value in, but allow the function to have a side-effect to the variable argument at the calling site. If you only have one in-/out-parameter and do not return a value (return type is **void**), it is often better to step away from the side effect and pass in the parameter by const-ref or by value and return the new changed value instead. So instead of defining a function like

```
void increment(int &i){
  ++i; // side effect on argument
}
```

better avoid the reference parameter and implement something like

Caution 3.4: no pointer parameters

> *Pointer parameters* are encouraged by some coding standards as an alternative to reference parameters[WSE+12]. **That is wrong.** While in C that lacks references you need to encode in/out parameters as pointers there is no reason to do so in C++. I will not show you the syntax for "naked" pointers right now, since you will not use them.

```
int plusOne(int i){
  return i+1;
}
```

The difference in calling is that the second version provides the side-effect on the call-site, whereas the first version the side effect is less obvious.

```
void useIncrementAndPlusOne(){
  int j{42};
  increment(j);
  j = plusOne(j);
}
```

When you actually have a function that requires multiple reference parameters then that is a clear indication that your function might be trying to do too much in a single function, as our example in the beginning of this chapter. Such a function is called to lack *cohesion*. At best, avoid side-effects in functions whenever possible. Side-effect free functions are easier to test and also safer to run concurrently.

cohesion

Function Declarations and Default Arguments

As we have already learned in chapter 0 a function must be known by either its declaration or definition before it can be called. For functions reusable in other modules, we place the function's declarations in a header file that has the same name part as its implementation file, but the `.cpp` replaced by `.h`. A further feature of C++ is that a function declaration for a function with parameters can specify *default arguments*. You can only do that for the rightmost parameters, as the position of an argument is significant. For example, consider the declaration in defaultargument.h (Listing 3.5) and its corresponding implementation in defaultargument.cpp (Listing 3.6). The function `incr()` can be called in two ways:[14]

default arguments

[14] please excuse the side-effect

```
  int i{42};
  incr(i); // 43
  incr(i,2); // 45
```

This is a case where different function calls with the same name but different number of arguments refer to the same function. However, there is also the case where you define the same function name with different number or type of parameters. This situation is called

Listing 3.5: defaultargument.h

```
void incr(int &var,unsigned delta=1);
```

Listing 3.6: defaultargument.cpp

```
#include "defaultargument.h"
void incr(int &var,unsigned delta){
  var += delta;
}
```

function overloading

function overloading. Instead of providing a default argument for our function `incr()`, we could also have overloaded the function and provided to different defintions and declarations of it as shown in overload.h (Listing 3.7) and overload.cpp (Listing 3.8). I have used a namespace here, to avoid violating the ODR, because we otherwise would have an ambiguity with the function `incr()` in the global namespace that we defined in defaultargument.cpp (Listing 3.6). We will learn much more about function overloading in chapter **??**.

Listing 3.7: overload.h

```
#ifndef OVERLOAD_H_
#define OVERLOAD_H_
namespace overload{
void incr(int& var);
void incr(int& var, unsigned delta);
}
#endif /* OVERLOAD_H_ */
```

Listing 3.8: overload.cpp

```
#include "overload.h"
namespace overload{
void incr(int& var, unsigned delta) {
  var += delta;
}
void incr(int& var) {
  ++var;
}
}
```

A further syntactical feature of function declarations that you might see, but that I usually do not recommend, is that you can omit parameter names in a function declaration, like `void incr(int &);`. While that saves typing, you should not apply that practice in your own header files, because good parameter names can help callers of your functions to better understand the function's intended semantics. You can even leave out parameter names in a function definition, if you are not using the parameter within the function body. The latter can occur, when you provide class member functions to

be overloaded in subclasses and one specific overload will not make any use of the parameter (see chapter **??**).

Function References as Parameters

While we have learned already a lot about function parameters and you have seen how you could pass lambda functions to generic algorithms in the previous chapter, I'd like to show you, how you can define your own functions taking other functions as arguments. In C++ functions are first class entities that means, you can pass functions as arguments and parameters and you can even have variables holding function references.

To define a reference variable or parameter of function type, you need to be able to specify such a function type. This can be done through omitting all names from a corresponding function declaration.

```
double func(double x);
```

For example, if we want to provide a function that prints out argument and function value of all functions taking a **double** argument and returning a **double** result, such as the trigonometric and other mathematical functions declared in header <cmath>, we can declare our function type as follows:

```
void printfunc(double x, double f(double)){
  std::cout<<"f("<< x <<") = "<<f(x)<<'\n';
}
```

To call a function passing another function as argument, you just need to spell that function's name without providing parentheses or arguments. This even works with lambda functions without captures. We will learn later how to pass lambdas with captures as function arguments (see chapter **??**).

```
  printfunc(1.0,std::atan);
  printfunc(2.0,[](double x){return x*x;});
```

However, we can not have functions as values directly like an **int**: Functions are always treated as references. If we define function parameters that treatment is automatically, as you can see above. But if we want to define a "variable" holding a function, we must define that variable as being of type reference to function.

```
using functype=double(double);
  functype &f=std::sin;
```

This can be achieved either through defining the variable as a reference directly, or by defining the underlying type to be a reference to function type. The syntax in the latter case becomes a bit more complicated, because we need to put parentheses around the reference symbol (&). Otherwise, we would define a function type returning a reference, which is not what we intend.

```
using funcreftype=double(&)(double);
  funcreftype g=std::cos;
```

Specifying a reference variable for a function directly without first declaring a type alias requires the parentheses around the reference symbol and the reference's name (`&name`).

```
  double (&h)(double)=std::tan;
```

While I showed you how to define function references it is very rare that you actually define such. However, passing function references as arguments to other functions is a feature we will at least use quite often, especially with the generic algorithms from the standard library.

Summary Parameters

syntax	name	application
`type param`	call by value	small types, parameter variable modified in function anyway, move-able/move-only types, functions implicit as references
`type const &param`	call by const-ref	default in-parameter passing, all uses where param remains unmodified
`type &param`	call by reference	out-parameter, side effect on param, stream objects
`type &&param`	*call by rvalue-ref*	*in-parameter, forwarding, usually not needed in user code, requires std::move(var) for passing lvalue var at call site*

Table 3.1: parameter type declarations and their use

We will learn a bit more about rvalue-references as parameters later, when we discuss move-enabled and move-only types in chapter 4.

3.4 Simple Functionality Guarantees

const member functions

pure functions

When you design a function you always need to consider if it should have a side-effect or not. Side-effect-free functions of a class[15] are declared as *const member functions*. Side-effect-free regular functions are also called *pure functions*. Functions with side-effects will always have a non-const reference parameter unless they use verboten global variables. A pure function's return type is never **`void`**, because it wouldn't make sense to call it, if nothing is returned, unless it is throwing exceptions. However, not all functions with a side-effect actually return nothing, often a (**`bool`**) return value indicates success or failure of the side effect.

[15] See chapter 4

Depending on the task of a function and its argument values a function might be unsuccessful in calculating its result or providing its intended side effect. Even functions of the standard library that you call might not be able to work with a given set of argument values, for example, you cannot index an element in an empty

`std::vector`. Now the big question is, what should happen, when something goes wrong. There are two sides of that situation: What guarantees does the called function provide and what responsibilities have to be met by its caller.

Often programmers tend to think and unit test only the happy cases where a function will work successfully. As a result such a function might not work with argument values outside the tested conditions. That can be OK, when that function will always be called with its implicit restrictions obeyed. It often provides best performance, because no code needs to run to check the arguments for validity. For example, `std::vector`'s indexing operator will not check its index argument to be in the range of available elements. Calling it with an out-of-range index will just result in *undefined behavior*[16]. That makes indexing a `std::vector` as fast as possible, but also dangerous to use if a programmer makes a mistake. So indexing a `std::vector` puts the responsibility of providing a valid index into its elements on the caller of the indexing **`operator[]`**. This is also called a *precondition*, *i.e.*, that the index given is less than the vector's `size()`. A function requiring a precondition that can be violated is said to have a *narrow contract*[ML11].

[16] Some C++ libraries actually provide a safe variant of such *undefined behavior*, paying performance for error detection ability. Since *undefined behavior* means anything can happen, that is within the bounds of the C++ standard.

precondition

narrow contract

If your code might be used by others unaware of its limitations or not fulfilling its preconditions, it can be better to not ignore the problem of errors. Either you clearly document your preconditions for each function and encourage the users to read your documentation, or you make your function just work regardless of its argument's values and thus removing any precondition on the arguments. In the latter case the function fulfills a *wide contract*[ML11].

wide contract

Even if the explicit preconditions are met, something else might go wrong, for example, because the function relies on some other function that failed. If your function cannot meet its guarantees, then it violates its *postconditions*. If a function's preconditions are met, it should never violate its postconditions in principle. However, as I said, things can go wrong beyond a function's control, for example, a function writing to a log file might fail when the space for the file gets exhausted. Functions with side effects are especially prone to such a situation.

postconditions

If your function cannot compute something useful if its preconditions aren't met or if it fails to provide its guaranteed postcondition, then there are several *error handling strategies* to chose from for dealing with this error situation:

error handling strategies

1. *ignore* the error and provide potentially undefined behavior
2. return a *standard result* to cover the error
3. return an *error code* or error value
4. provide an *error status* as a side effect
5. throw an *exception*

Ignoring Errors

We have already seen error handling strategy 1 with vector indexing. You should rely on undefined behavior only, when you can guarantee it will never happen and when otherwise performance and usability of your function would suffer and you can trust your callers. Often it helps your callers, when you provide a version of your function that detects the situation and throws an exception, like `std::vector` does with its `at(index)` member function that throws a `std::out_of_range` exception if called with an out-of-bounds index.

Covering Errors

Providing error handling strategy 2 can ease the code that calls your function. For example, if we look back at the function `inputName()` from Listing 1.1 we might recognize that it will return an empty string in the case that the input stream is empty or already in a `bad()` state. The corresponding greeting will look like "Hello , how are you?", which is not that splendid. However, if a user decides not to enter a valid name, the function `inputName()` might provide a standard value for those anonymous users. You might have seen such standard values, for example, in your text editor, when you just begin to type and want to save the file. You might end up with a file name like "Untitled.txt" or "Document1.doc". A version of `inputName()` that never returns an empty string could look like the following code.

```
std::string inputName(std::istream &in) {
  std::string name{};
  in >> name;
  return name.size()?name:"anonymous";
}
```

While this is convenient to use, it has a drawback. Just consider, someone wants to use that function in a country that doesn't understand English. Now your function has become useless. So it might be better to provide a means to provide the wanted standard value by the caller. Naïve programmers might employ global variables for that, you don't! Pass in the standard value to be used as an additional last parameter and provide the value you used initially as its default argument. This still allows existing client code to remain unchanged and by providing an additional value on the call client code with a different requirement still can use it like the following variant of `inputNameWithDefault()`[17]

[17] Please remember `default` is a keyword and cannot be used as parameter name.

© Peter Sommerlad - printed: 05/10/2015 - 17:01

```
std::string inputNameWithDefault(std::istream &in
              , std::string const &def="anonymous") {
  std::string name{};
  in >> name;
  return name.size()?name:def;
}
```

This way of covering errors by a standard result can be used wrongly. It only makes sense if code calling the function can continue its processing with the returned value without detecting its "specialness". The following code piece shows a counterexample where the special value is used to detect the error again from the outside. Don't do this, because that is actually error handling strategy 3 returning an error value and no longer a means to cover the error.

```
  auto name=inputNameWithDefault(std::cin,"noname");
  if (name == "noname"){
    return;
  }
  // continue processing
```

Error Value

Returning an error value as in error handling strategy 3 works, when the return type of your function contains more values than your function possibly can return normally. Sometimes a value like `-1` is used to denote an error, if your function would otherwise always return a positive value. Functions that rely on a side effect, might return a **`bool`** value to denote success or error. Prominent examples of this error handling mechanism are the POSIX operating system level API functions[POS90] that are available in C and C++. For example the POSIX `open()`-family system call's return value is defined as follows:

> Upon successful completion, these functions shall open the file and return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, these functions shall return -1 and set errno to indicate the error. If -1 is returned, no files shall be created or modified.

While this style of error reporting alongside the normal function return value can be very convenient for the implementer of the function, it lays a burden on the caller. If the function is mostly called for its side-effect, then the caller might ignore its return value at all and thus never recognize the error. Otherwise, every call of such a function must be followed by a conditional checking for a potential error and then handle that error or report it up the call chain. With the POSIX.1 API we also see that in addition to the return value, details of the error condition are provided by a global variable `errno`. I rarely see code where all possible failure values of `errno` are really checked for. Great care must also be taken, when the `errno` variable is read, since other library and system functions called during that

analysis will change its value. This `errno` global variable is a design more than 40 years old, that is established but shouldn't be taken as a model for more modern error reporting.

Sometimes, one of the possible return values are deliberately defined as an error, because it is improbable that it will *actually* occur. For example, searching within a `std::string` with one of its `find()` member functions defines a value of `npos` as an indicator of "not-found error". The range of results of found positions is defined by an unsigned integer type and `string::npos` is defined to be the value `-1` represented as such an unsigned integer. Therefore, `npos` is of the defined type but a value that one might never be able to encounter, since you will not be able to define a string object that long. Zero, couldn't have been the value of choice to signal the error of not-found, since `0` is a valid index denoting the start of the string.

The basic mechanism of reporting an error by return value breaks down, when the function actually requires all possible values of a type for its result and none of its possible values can be defined to actually denote the error. For example, for our `print()` function from Listing 3.4 that returns its reference parameter it is impossible to return an error value. This leads us to the next strategy.

Error Status Side Effect

When your function takes a reference parameter to an object[18] as `print()` or `inputName()` do, then one option to tell the caller that an error occurred is to set the internal state of that object to an "error state". This is the behavior of the standard stream objects. The stream classes provide "flags" that are set to denote an error. Any flag actually set puts the stream into an error state (`!good()`) and further I/O requests are silently ignored unless the stream object's error flags are `clear()`ed again explicitly. This is important to understand when reading formatted input that can fail because of an invalid input characters are given (see section 1.4).

This error handling strategy 4 can be used, if the underlying error source is outside of the control of the program and thus not a programming logic error, such as a violated precondition. For stream object this is an OK mechanism, also because in contrast to `errno` the error status is sticky and allows continuing of processing, because further side-effect carrying operations on a bad stream are just ignored.

In general, this error reporting strategy makes it easy to ignore the errors but also hard to track, if it is checked very late or never.

A perversion of "error status as side effect" are API's where all functions take an additional "error status" reference parameter.[19] So there is no chance to ignore the error reporting by the caller, but still a lot of possibility to actively ignore the error status. If the API requires several function calls in a row, often such an error variable is re-used and sometimes without being cleared in between. Whenever you design an error reporting, do **not** do it like the following code

[18] All non-const member functions have their object as an implicit reference parameter.

[19] I have had to use such error reporting designs in my past live and the API-client code looked very ugly and it was error prone to handle or ignore the error.

example that requires passing an error status variable.

```
int doSomething(int input, int &error_status);
```

Exceptions

exceptions

throw

While the other strategies for error reporting are also applicable to C, error handling strategy 5 *exceptions* are clearly a C++ feature. While the standard library provides a class hierarchy of exception classes based with `std::exception` you are not bound to only *throw* exception values from that class hierarchy. Any copyable value can be thrown as an exception in C++. However, it is often wise to refrain from throwing an **`int`** (**`throw`** `42`), but to either use an existing class derived from `std::exception` or to create your own type to be used as exception value in a specific situation.

try-catch block

The type of an exception is important, because handling the exception thrown in a *try-catch block*, where one or more **`catch`** clauses are matched against the type of the exception value thrown. The **`catch`** clause with the first matching type up the call stack is chosen by the run-time system to handle the thrown exception, there multiple **`catch`** clauses are considered in the sequence of their definitions. While that *stack unwinding* up to the matching catch clause happens, all local variables get destroyed and their resources reclaimed automatically. The latter is a reason, while destroying an object must not throw an exception in C++.

stack unwinding

To guarantee that no exception actually passes beyond a try-catch block, you can add a "catch-all" clause to its end as **`catch`**`(...){/* */}`. If an exception thrown is not caught or another exception gets thrown, while stack unwinding due to an exception happens, then your program is terminated. The example sillyexceptions.cpp (Listing 3.9) shows the C++ syntax for throwing and catching exceptions and also re-throwing a caught exception within a catch block. Take the code and try to figure out for each **`throw`** statement, which **`catch`** clause will be matched. You can also see from the example code that a whole function body can be a try-catch block. Remember the example is really silly, you usually will throw an exception from a function called within a try block, not directly. You can also see the guideline *throw by value, catch by const-reference* applied, that you should follow when you implement your own exception handling. If you carefully observe the catch-clause with `std::logic_error` is placed before the one with its base class `std::exception`. This is required, because otherwise the **`catch`**`(std::exception` **`const`** `&)` would shadow the other catch clause. This is also the reason, why a catch-all clause **`catch`**`(...)` always has to be the last. Experiment with the code and observe what happens when you reorder the catch clauses or enforce the throwing of one of the exceptions.

throw by value, catch by const-reference

A more useful application of exception handling is a variant of our `inputAge()` function of page 46. Instead of using a separate `std::istringstream` object to convert the input text line into an **`int`**, we

Listing 3.9: sillyexceptions.cpp

```
#include <iostream>
#include <cstdlib> // srand, rand, time functions
#include <stdexcept>
void mightthrow_logic_error(){
  if (rand()%2) throw std::logic_error{"logic"};
}
void must_be_called_with_greater_3(unsigned i){
  if (i <= 3) throw std::invalid_argument{"too small"};
}
int main() try {
  srand(time(0)); // randomize rand()
  mightthrow_logic_error();
  must_be_called_with_greater_3(rand()%6);
  if (rand()%2) throw std::string{"error"};
  if (rand()%2) throw 15;
  throw "hallo";
} catch (std::logic_error const &e) {
  std::cout << "logic error: \'" << e.what() << "\'\n";
  throw; // re-throw the caught exception
} catch (std::exception const &e) {
  std::cout << "exception: \'" << e.what() << "\'\n";
} catch (int const &i) {
  std::cout << "exception int: <" << i << ">\n";
} catch (std::string const &s) {
  std::cout << "exception string: \"" << s << "\"\n";
} catch (...) {
  std::cout << "unknown exception value\n";
}
```

stoi()

chose the function `stoi()`, that will throw an exception if the input can not be converted from the `std::string` given as argument. This helps to simplify the code, at least it avoids creating the stream object and one if statement.

```
int inputAge(std::istream& in) {
  while (in) {
    std::string line{};
    getline(in, line);
    try{
        return stoi(line);
    } catch (...){}
  }
  return -1;
}
```

In contrast to other programming languages with exceptions, C++ exception handling is limited in its features. The main reason is that exceptions should not incur high run-time costs. So the amount of information a typical exception object carries is minimal: you won't get detailed diagnostics of the source position of the throw expression nor a stack trace when you catch an expression.[20] Classes de-

[20] A debugger will usually be able to show such information.

rived from `std::exception` provide the `what()` member function to allow some plain text information to accompany the exception object but not more.

A further benefit of the simpler C++ exceptions is that they do not need to be declared, as for example in Java™ . The only exception declaration possible is given by the keyword **`noexcept`**. When specified after a function's parameter list closing parenthesis, it guarantees that this function will never throw an exception. Since there are situations where no exceptions must occur, this guarantee is required. Without a **`noexcept`** guarantee a function might throw any exception value. The violation of such a **`noexcept`** guarantee forces the program to terminate. The existence of **`noexcept`** can also help the compiler to better optimize your code.

```
int minimum(int a, int b, int c) noexcept {
  return a<b?(a<c?a:c):(b<c?b:c);
}
```

Dynamic exception specification with `throw()`

C++ allows dynamic exception specifications, but the only useful dynamic exception specification is **`throw`**`()` after a function's parameter list, showing the so-called *no-throw guarantee* that is equivalent to C++11's **`noexcept`**. A dynamic exception specification could list a number of types like **`throw`**`(std::exception,`**`int`**`)` and would throw `std::unexpected` if another kind of exception is throw across that function's boundary.

```
int maximum(int a, int b, int c) throw() {
  return a>b?(a>c?a:c):(b>c?b:c);
}
```

Practice showed, that it is almost impossible to correctly determine the lists of types to be thrown, especially within generic code. That observation led to the invention of the **`noexcept`** syntax that is simpler and allows compile-time calculation of its condition, *i.e.*, **`noexcept`**`(compile-time-condition)`, based on the noexcept-value of functions to be called within a function's body, for example.

RETRO 3.1: PREFER `noexcept` OVER `throw()`

After we learned syntax and mechanics, I need to tell you when to actually employ exceptions. Exceptions cannot simply be ignored at run time, but your code can be simpler, if you do not have to check for error conditions after every function you called. That is one of the powers of exceptions as error reporting mechanism, that you can write your code in a way that looks like for the straightforward "happy" scenario and only in dedicated places actually catch

exceptions that might occur.

narrow contract

wide contract

Whenever your function has a *narrow contract*[ML11], that means it asks its callers to fulfill its preconditions then throwing an exception can be a good choice for error reporting. If your function provides a *wide contract* and it guarantees a side effect, that for some reason cannot happen correctly and there is no means of handling that problem internally or providing the erroneous state as part of its side effect then throwing an exception is also a way to go. A wide contract function without side effects cannot go wrong and thus, doesn't need to deal with exception handling, it could actually be declared as **`noexcept`**. There is one place where all the other error handling strategies fail, a class' constructor (see chapter 4). If an object cannot be completely constructed, because either its argument values are wrong or a required resource cannot be acquired you have to throw an exception. This will ensure that no half-baked object remains lying around.[21]

[21] There are some tricky object creation situations in generic code in case of possible exceptions, but we will meet those later.

The standard library header `<stdexcept>` provides some subclasses of `std::exception` that you can use in your own code, if you do not want to invent your own exception types. Useful are `std::logic_error` and its subclasses like `std::invalid_argument` to denote errors in program logic such as a violated precondition. These classes can be constructed with a string argument to provide further information as shown in Listing 3.10

Listing 3.10: squareroot.cpp

```
#include "squareroot.h"
#include <stdexcept>
#include <cmath>
double square_root(double x) {
  if (x < 0)
    throw std::invalid_argument{"square_root imaginary"};
  return std::sqrt(x);
}
```

However, whenever a function can handle an error, *i.e.*, an exception that is thrown during its execution then it should do so and only pass on the exceptions or errors that actually need to be handled by its caller or its caller's callers. On the other hand, you need to learn when and where to actually put your try-catch blocks sensibly. We will see more examples later in this book. Sometimes it is just OK to ignore the possibility of exceptions and let your function's caller deal with them if they happen. A good layering of your system should not pass low-level exceptions directly up to higher layers, but try to handle them, or at least translate them so that they refer to the abstraction level of the layer that failed to handle the error situation. Otherwise, your application users might be offended by incomprehensible error messages.

Throwing exceptions should be reserved to really exceptional situations. Code that misuses exceptions as passing return values, for

example, can be written in C++ but is a very bad design choice. You will learn more about throwing your own exceptions in the remainder of this book and you can look at some guidelines in the C++ FAQ [Cli11].

Testing for Exceptions

If you have a function that might fail, then this is also a situation requiring unit tests. Therefore, you need to be able to actively force the error situation and an ability to check if the expected exception actually occurs. A simple test case for testing the throwing of an exception is shown in the following code. It ensures that the promise I made above about the `at()` member function of `std::vector` is true.[22]

[22] Such a test for an infrastructure component is something you usually don't write, but it can be useful as a *learning test*[Bec02] to ensure assumptions you make about a library

```
void testEmptyVectorAtThrows() {
  std::vector<int> empty_vector{};
  ASSERT_THROWS(empty_vector.at(0),std::out_of_range);
}
```

Our `square_root()` function's exceptional behavior can be tested as follows.

```
void testSquareRootNegativeThrows(){
  ASSERT_THROWS(square_root(-1.0),std::invalid_argument);
}
```

Often it is hard to actually trigger exceptional situations, especially when environmental side-effects are actually the reason for the exception. For example, to actually have `std::vector` throw a `std::bad_alloc` exception that occurs when no more memory can be allocated is hard to trigger. However, later I'll show you an example how you can actually enforce such a situation by providing a test-version of its allocator template argument. This is called a test stub or mock object. **TODO!** forward xref.

Summary

A *precondition* is the requirement to be met by the caller of a function manifested in the arguments provided at the call site. A *postcondition* is the guarantee given by the function, if its preconditions are met, manifested by its return value and side effects. Both together are called the *contract* of the function. If for some reason the contract of a function cannot be fulfilled, then an error situation occurs that might be reported to the function's callers. There exist several mechanisms to report such errors and each one has its individual benefits. If your function's contract can always be fulfilled, *e.g.*, because it is a very simple function, then adding **`noexcept`** to your function documents that. When writing unit tests for your functions always also include tests that raise the error conditions and check that your error reporting mechanism works.

3.5 Exercises

Please go back to your previous exercise's solutions and consider for each function that you wrote, if it has a narrow or a wide contract and if and how your error handling strategy is.

Simulated 7-Segment Digit Display Pocket Calculator

TODO!