CS24 Assignment 4

The files for this assignment are provided in the archive **`cs24hw4.tgz`** on the course website. The archive contains a top-level directory **`cs24hw4`**, in which all other files are contained.

For the questions you need to answer, and the programs you need to write, put them into this **`cs24hw4`** directory. Then, when you have finished the assignment, you can re-archive this directory into a tarball and submit the resulting file:

> *(From the directory containing **`cs24hw4`**; replace **`username`** with your username.)*
> **`tar -czvf cs24hw4-username.tgz cs24hw4`**

Then, submit the resulting file through the course website.

Problem 1: Implementing OOP in C *(25 points)*

(The files for this problem are in the **`cs24hw4/classes`** directory.)

As discussed in class, it is straightforward (if tedious) to map many object-oriented programming concepts straight into the C programming language. This problem will give you an opportunity to experiment with this yourself.

In this problem we will focus on a simple class hierarchy for representing shapes, implemented in C++. C++ is interesting because it allows you to explicitly specify which methods are virtual (dynamic dispatch via virtual function pointer tables) and which are nonvirtual (static dispatch).

You can also declare a method to be *pure virtual*, which means that the class is not providing an implementation; subclasses are expected to provide an implementation of the method themselves. This is simply C++'s mechanism for declaring a method to be abstract. A class containing pure-virtual methods cannot be instantiated; it is missing some of its functionality so it would make no sense to instantiate it.

Back when C++ was still called "C with Classes," programs were actually translated straight into C code and then compiled with the C compiler. The mechanisms for implementing objects are exactly the ones we have discussed in class.

In the **`shapes.h`** and **`shapes.c`** files you will find the C code that implements our shape class hierarchy. You can see how many declarations and functions are needed even for simple classes (and I haven't included destructors), so you can see how much the C++ syntax helps you. This is the kind of code that the "C with Classes" translator would generate, although **`shapes.h`** is much more well-commented so you can understand what is going on.

Your Tasks

1. You won't need to change anything in **`shapes.h`**, but in **`shapes.c`** you will need to fill in the functions that are marked "TODO". Virtually all of these functions are very short to implement.

   See the Tips and Guidelines section below for additional information.

There is also a test program provided, called **`shapeinfo.c`**. It creates a cone, a sphere, and a box, and then prints out some details of each shape using a generic function written against the Shape base-class. You also need to fill in the part marked "TODO" in this file.

Once you have completed these functions, you should be able to compile **`shapeinfo`** using the provided **`Makefile`**, and then you can give it a test.

2. In a file **`cs24hw4/classes/invoke.txt`**, answer the following question. *(3 points)*

   Assume that **`%r8`** contains the address of an object that is a properly initialized **`Shape`** subclass (e.g. a sphere), and we want to invoke the virtual function **`getVolume()`** on this object. Write the series of x86-64 assembly instructions necessary to do this. Make sure to explain the sequence of operations.

   Ignore concerns about callee/caller-save registers, and also ignore how floats are returned since we haven't talked about this in class. Just focus on how the method is invoked.

   If you ever need to debug C++ code at the assembly level, you will frequently see sequences of instructions like this.

Additional Tips and Guidelines

In C++, a subclass constructor is not allowed to directly initialize the superclass data-members. Therefore, the subclass constructor always calls the superclass constructor as the very first step, so that the superclass can initialize its data-members before the subclass constructor does anything else. When you implement the subclass constructors, you should do the same thing. (It will keep your constructors nice and short, too!)

When subclasses pass pointers to their information to the superclass, you might notice that the pointer-types don't match up, and in fact C will complain about this. **You need to perform an explicit cast operation to get the compiler to accept this.** See the **`main()`** function in **`shapeinfo.c`** for an example of this. As mentioned in class, we can cast from the "subclass" type to the "superclass" type because the structs have the same preamble.

Make sure to pay attention to which methods are nonvirtual and which methods are virtual. The calling mechanism is different for each of these kinds of methods. You should not have to change **`shapes.h`** at all.

You should avoid repetition as much as possible! For example, all of the subclass constructors take initial values for their members, and they also have a "**`setXXXX()`**" mutator for setting the values of their members. The constructors should use these mutators. Again, this will keep your code nice and short.

Don't forget to use assertions in your code, where the comments state that inputs are checked or constrained in some way. (Another reason to have constructors reuse mutators; that way your assertions are all in one place and your code stays short.)

Definitely check the answers your program generates! If you get wrong answers, you may have either a math bug or an OOP bug in your code. In C, you might want to write your numbers as decimal numbers instead of whole numbers, e.g. 1.0 / 3.0 instead of 1 / 3.

Problem 2: Exceptional C Code *(32 points)*

(The files for this problem are in the **`cs24hw4/exceptions`** directory.)

For this problem you will create your own x86-64 implementations of **`setjmp()`** and **`longjmp()`**, called **`my_setjmp()`** and **`my_longjmp()`**, as well as a comprehensive test suite for verifying the correctness of these functions.

Background

The C programming language provides two unusual functions **`setjmp()`** and **`longjmp()`**, declared in the standard C header **`setjmp.h`**. These functions allow you to perform *non-local jumps* (i.e. not within the same procedure) in your C programs. The functions use a data-type named **`jmp_buf`**, short for "jump buffer," which is used to save and restore the execution state of your program. This **`jmp_buf`** type is simply an array of words (or quadwords on x86-64), the number of elements depending on both the processor and the OS. These functions work as follows:

`int setjmp(jmp_buf buf)`

> When **`setjmp()`** is called, it stores the current execution state into **`buf`**, then returns 0.
>
> You can use **`setjmp()`** multiple times to create **`jmp_buf`**s corresponding to different points in a program's execution, since **`longjmp()`** takes a **`jmp_buf`** argument.

`void longjmp(jmp_buf buf, int ret)`

> When **`longjmp()`** is called, it restores the execution state saved in **`buf`**, causing the program to *appear* that it has returned from **`setjmp()`** a second time. However, this time the caller sees **`setjmp()`** return a nonzero value: if the argument **`ret`** is nonzero, then that is returned; if **`ret`** is zero then the call returns 1.
>
> You can **`longjmp()`** using a specific **`jmp_buf`** multiple times; there is no limit. The only requirement is that you can only **`longjmp()`** back into a currently executing function. Or, as the documentation puts it: "The **`longjmp()`** routine may not be called after the routine which called the **`setjmp()`** routine returns."

This may seem like some kind of crazy magic, but it's actually very simple – it is all straightforward manipulation of the stack that follows the System V AMD64 ABI calling convention.

> The **`setjmp()`** function saves execution state associated with the registers and stack into the **`jmp_buf`** array, which obviously must be large enough to hold all of this state. Since **`setjmp()`** also follows the System V AMD64 ABI calling convention, it really only needs to record the callee-save registers, and the values necessary to resume execution at the point that **`setjmp()`** was called. This includes the current stack pointer, and the caller's return address (which can be grabbed off the stack).
>
> The **`longjmp()`** function basically does the reverse of what **`setjmp()`** does: it restores all of this execution state from the **`jmp_buf`**, so that when **`longjmp()`** returns, it *appears* that **`setjmp()`** is returning a second time. We are *not* actually returning from the **`setjmp()`** code, but the caller won't know the difference since the execution state is restored as when **`setjmp()`** was called the first time. The only difference is that the return value (in **`%eax`**; it is an **`int`**) is different, which is how the caller can tell that **`longjmp()`** returned.

> This is also how we are able to perform a *non-local* goto with these functions: Since **`longjmp()`** replaces the stack-pointer with the old value saved by **`setjmp()`**, all stack frames from intermediate function-calls are simply chopped off of the stack. Additionally, since **`longjmp()`** restores the caller's return address onto the stack, when **`longjmp()`** returns then execution will end up exactly where the caller originally called **`setjmp()`**.

Some important things are not saved or restored. Only the values specified above are saved and restored, but in general the data on the stack is not saved or restored. This means that if the caller's local variables have been modified since the time **`setjmp()`** was called, the changes to those local variables remain.

This is also why you can only call **`longjmp()`** to return to a function that is still executing. If the function where **`setjmp()`** was called has already returned when **`longjmp()`** is called, then that function's frame is *long gone* from the stack. When **`longjmp()`** restores the execution state from the **`jmp_buf`**, the stack will now have garbage for that frame, and you are going to get some very strange behavior. (Or a segfault, which you are probably familiar with by now.)

Your Task

In the **`cs24hw4/exceptions`** directory you will find an implementation of the C exception handling mechanism described in class. (It is not essential to understand all of the details of this implementation, but an appendix is provided for the curious. You should also test your functions with the example program to ensure they work properly.) As described in class, the exception mechanism depends heavily on the **`setjmp`**/**`longjmp`** pair of functions.

For this problem you will implement your own version of **`setjmp`** and **`longjmp`**, as well as specifying the size of your own **`jmp_buf`** type. Open **`my_setjmp.h`** to see what you will need to do. In this file you will see that the implementation can either use the standard versions of **`setjmp()`** and **`longjmp()`**, or by defining the **`ENABLE_MY_SETJMP`** symbol you can make it use your own versions, named **`my_setjmp()`** and **`my_longjmp()`**.

These functions can only be implemented in assembly language, since they manipulate the stack in ways not possible from C. Implement these two x86-64 functions in the file **`my_setjmp.s`**, making sure to exactly mimic the behaviors of the standard **`setjmp()`** and **`longjmp()`** functions.[1] You must also set **`MY_JB_LEN`** to be the number of elements needed to store all of your execution state.

Once you have finished implementing **`my_setjmp`** and **`my_longjmp`**, you must also create a test suite that verifies the correctness of your implementations. You will be graded on the completeness of your suite, so think carefully about the different behaviors of **`setjmp()`** and **`longjmp()`** and how to test them. (Try to devise 4 or more kinds of tests.) Here are some additional guidelines:

- **Your test program should be called `test_setjmp.c`, and it should include `my_setjmp.h`.** We would recommend that you start out using the standard **`setjmp`** code, not your **`my_setjmp`** code. This way you can verify your test code first, and then switch over to your own version of **`my_setjmp()`** when your tests are trustworthy.
- Add a build rule to the **`Makefile`** that builds **`test_setjmp`**. Add another build rule called "**`check`**" that runs **`test_setjmp`** (building it if necessary), so it's easy to test your code!

[1] Some **`setjmp()`** and **`longjmp()`** functions also save and restore registers for floating-point, MMX/SIMD, etc. **You don't have to do any of that!** Just worry about the general-purpose registers, as described in the problem.

- **Don't use the exception-handling macros in your tests.** Those are only provided as a demonstration of how C exception-handling could work.
- **Make sure that every test exercises a different characteristic of your code.** Each test should verify different interactions or code-paths in your functions. Testing the same code-path five times doesn't buy you anything over testing something once.
- **Every kind of test should live in its own function.** That way your main function can just call each kind of test in sequence.
- **Make sure that every test clearly reports what it is testing, and whether it passes or fails!** Test suites are frequently automated, and you need to make it obvious what is going on. (e.g. "**`longjmp(buf, 0) returns 1: PASS`**")
- Speaking of which… don't forget that **`longjmp(buf, 0)`** should cause **`setjmp()`** to return 1, but **`longjmp(buf, n)`** for *n* != 0 should cause **`setjmp()`** to return *n*. You should verify this behavior.
- You should verify that **`longjmp()`** can correctly jump across multiple function invocations. Don't make all tests **`longjmp()`** within the same function that **`setjmp()`**s. However, for some tests it's fine if you use **`setjmp()`** and **`longjmp()`** in the same function, such as verifying the **`longjmp()`** return-value.
- This is trickier to do, but you should definitely try to verify that your functions don't corrupt the stack in certain obvious ways. For example, you might try putting local variables with known values on both sides of your **`jmp_buf`** variable to ensure that your **`setjmp()`** implementation doesn't go beyond the extent of the **`jmp_buf`**. You might also do this to ensure that **`longjmp()`** properly restores **`%rsp`** and the other callee-save registers, to ensure that local variables can be accessed afterward.

Finally, you should test your version of **`setjmp`** and **`longjmp`** with the example C exception-handling program. If this also works, congratulations!

Problem 3: Garbage Collection *(43 points)*

In the **`subpython`** directory you will find a simple Python-like interpreter. It supports evaluating Python-like expressions in a Read-Eval-Print Loop (REPL). You can build it with the provided **`Makefile`**, and then give it a try:

```
$ ./subpython
Using a memory size of 1024 bytes.
> 2 + 3
5
> "hello"
"hello"
> a = 5
> b = 9
> a + b
14
> lst = [1, 'goodbye', 3]
> lst[1]
"goodbye"
>
```

This interpreter is only *Python-like*. It has a syntax similar to Python, but it only supports very limited functionality:

- Supported data types are floats, strings, lists and dictionaries. Lists and dictionaries are implemented very inefficiently.
- Basic arithmetic is supported.
- Global variables are supported, and can be assigned to multiple times.
- Global variables may be deleted with the **`del`** command, e.g. "**`del a`**". List elements and dictionary keys cannot be deleted.
- Dictionary keys may be strings or floats, but no other type.
- Both list elements and dictionary entries may be the target of an assignment. New dictionary entries may be created by assigning to a key that is not in the dictionary. Existing ones may be updated using a similar mechanism.
- Lists cannot be extended or resized.

This is sufficient to generate very interesting graphs of values within the interpreter. For example, you can do fun things like this:

```
> lst = [1, 'b', 3]
> lst[2] = lst
> lst
[1, "b", [1, "b", [1, "b", [1, "b", [..., ..., ...]]]]]
```

Finally, the interpreter provides two functions:

- **`quit()`** exits the interpreter. (You can also exit the interpreter with Ctrl-D, etc.)
- **`gc()`** manually invokes the garbage collector.

The `subpython` Allocator

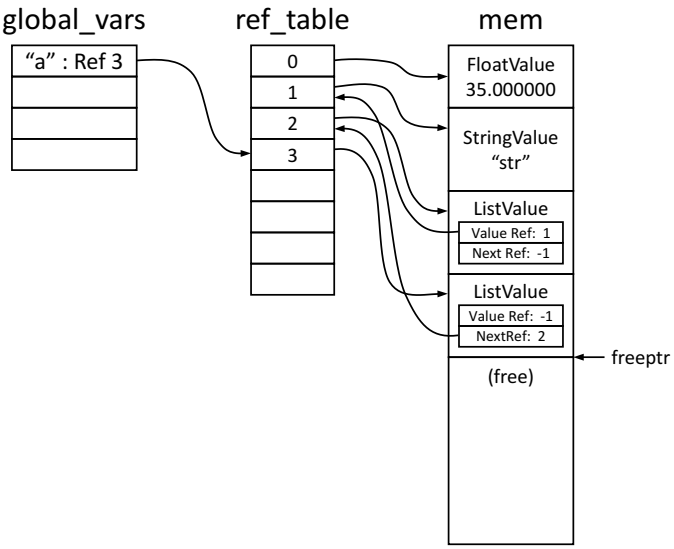
This interpreter uses a simple memory allocator very similar to last week's "unacceptable allocator" for allocating all of its values. While this was unacceptable last week, it is perfect for this week

since the allocator can use garbage collection to identify and reclaim garbage, and it can move around regions of memory to compact available space and avoid fragmentation.

As the interpreter evaluates various expression, it will allocate them from the memory pool. These values are not released though, and eventually the pool will fill up. At this point, the allocator will automatically run the garbage collector. You can also manually invoke the garbage collector with the **`gc()`** function.

To facilitate this, all values are accessed indirectly through *references*. Just by introducing one level of indirection, the allocator can implement some very sophisticated functionality. To help you understand how everything works together, here is a diagram of what the allocator will have in memory after running two commands:

```
> 35
35
> a = ['str']
```



All values are represented by versions of the **`Value`** struct (in **`types.h`**). You will notice that **`Value`** has four "sub-types," **`FloatValue`**, **`StringValue`**, **`ListValue`** and **`DictValue`**. These types use the standard C approach of having the same leading members; this allows a **`Value*`** to be cast to the appropriate sub-type (see the type member), so that the specific members can be accessed.

All **`Value`** objects are allocated in the memory pool (managed in **`alloc.c`**), using the exact same approach used in HW3's "unacceptable allocator." As long as there is available space, new values are always allocated at **`freeptr`**, and then **`freeptr`** is moved forward.

All access to **`Value`** objects is performed through references. In this system, a "reference" is simply an index into the **`ref_table`** maintained by the allocator (also in **`alloc.c`**); this table records the actual address of each value. Global variables (managed in **`eval.c`**) store a **`Reference`** to the value associated with the name. List nodes and dictionary nodes both use **`Reference`**s to record the values they hold. If code needs to look up the actual value from a reference, it must use the **`deref(Reference)`** function provided by **`alloc.c`** to get back a pointer to the **`Value`**.

(Note that -1 is a special value for references; it is a "null" reference that doesn't refer to anything. The file **`types.h`** defines a symbol **`NULL_REF`** that should be used within the code. Calling **`deref(NULL_REF)`** returns a **`NULL`** pointer.)

This allows the allocator to change the actual address of values without breaking any other part of the program. As long as the **`ref_table`** entries are updated properly, a **`Value`** can be moved within the memory pool without any problem. And, since each **`Value`** knows its own **`Reference`**, updating the **`ref_table`** is quite straightforward.

You can make **`subpython`** print out the current state of its memory pool by giving it the **`-v`**

argument. If you do this and feed in the above statements, you will see exactly the above results. Additionally, you can specify a different pool size with the argument **`-m size`**.

Your Task

In its present form, the interpreter will eventually run out of memory because the garbage collector is unimplemented. You must implement a compacting garbage collector for the interpreter so that it can identify and reclaim all garbage.

Implement your garbage collector in the file **`alloc.c`**, between the "**`//// GARBAGE COLLECTOR ////`**" comment and the "**`//// END GARBAGE COLLECTOR ////`**" comment. You can either implement a compacting mark-and-sweep algorithm, or a stop-and-copy algorithm, but what you implement must satisfy these requirements:

- All garbage must be successfully identified as garbage, and reclaimed.
- Your collector must properly handle cycles in the reference graph.
- The collector must compact memory such that all in-use memory is at the start of the pool (before the **`freeptr`** pointer), and all available memory is at the end of the pool (after the **`freeptr`** pointer).
- All entries in the **`ref_table`** that point to garbage must also be reclaimed so that the entries can be reused.

For some approaches, you may need to make changes to the **`Value`** type and its subtypes in **`types.h`**. Just make sure that if you add fields to **`Value`**, you must also add them to the subtypes of **`Value`**; otherwise you will have strange bugs.

Besides that file, virtually all of your work should be in **`alloc.c`**. You should not need to make substantial changes to other parts of the allocator, although you may need to change some things depending on how your collector works. You should not need to understand much about the rest of the interpreter, so don't feel obligated to understand every line of code in the project.

You will likely need to copy around chunks of data in the memory pool. There are several functions declared in the C standard header **`string.h`** that you should consider using:

```
void * memcpy(void *dst, const void *src, size_t len);
void * memmove(void *dst, const void *src, size_t len);
```

Both of these functions copy **`len`** bytes from the region starting at **`src`** to the region starting at **`dst`**. Both of them return **`dst`**. There is a subtle but extremely important difference between these two functions, which is not explained in this assignment, but it could seriously impact the correctness of your implementation. You should read about these functions, and choose the one most appropriate to your specific implementation. Explain the rationale behind your choice in comments in your code.

Make sure to test with both large and small string values; this will uncover any memory-copying bugs in your implementation.

```
void * memset(void *b, int c, size_t len);
```

This function writes the value of **`c`** (cast to an **`unsigned char`**), **`len`** times, to the region starting at **`b`**. This can be extremely useful for debugging. For example, if you reclaim a

memory region, and you want to be sure you don't accidentally use some value in that memory region, you can write a value like **0xCC** to all bytes in the region you have freed. This way, if you try to access a value and it appears to be **0xCCCCCCCC**..., then you will know you are accessing an invalid value.

The **ref_table** is where **Reference**s are mapped to **Value***s. All **Value*** addresses should be within the memory pool; you can check this using the **is_pool_address()** function. To modify a **Reference**, simply write the new address of the **Value** into the appropriate location in **ref_table**. Recall that all **Value** objects know their own **Reference**, so this should be very easy.

When a **Reference** is no longer needed (e.g. because the corresponding **Value** is garbage and is being reclaimed), you should write **NULL** into the **ref_table** for that **Reference**. This will ensure that the reference can be reused.

Note that references are not compacted. Only the memory pool is compacted.

Additional Hints and Tips

To give you a sense of how much work this implementation might be, the simple compacting mark-and-sweep solution is around 100 lines of code, excluding comments. Your implementation might be longer or shorter than this.

If you choose to implement a stop-and-copy collector, you will need to divide the memory pool into two regions. A simple way of doing this is to modify the allocator to use two pools instead of just one, where each pool is half the size of the original pool. This allows you to avoid modifying the allocation code. A more complex approach is to store the "from" and "to" regions within a single pool, but you must also modify the allocation code to use the new pool layout correctly. If your implementation keeps the "from" and "to" regions within a single pool, you may receive a bonus of up to +5 points if your implementation is correct and of high quality.

Some implementations may need to traverse all **Value**s that are currently stored in the memory pool. You can see an example of how to do this in the **memdump()** function in **alloc.c**.

All implementations will need to know what global variables are currently defined. To this end, the evaluator provides a function **foreach_global()** (specified in **eval.h**) that takes a function-pointer, and calls this function once for every global variable that it currently knows about. The passed-in function must have this signature:

```
void function(const char *name, Reference ref);
```

An example of how to use this function is in **print_globals()**; you can look at its implementation in **eval.c**. (This should be all you really need to look at in **eval.c**.)

If the compiler complains about a function argument being unused, you can do this in your code:

```
void someFunction(int importantArg, int unusedArg) {
    // Keep the compiler from complaining about unused arguments.
    (void)(unusedArg);

    ...  // Other implementation
}
```

Appendix: Exceptional Control Flow in C

This appendix describes non-essential details of the example implementation of exceptional control flow in C. The main files to look at are **`c_except.h`** and **`c_except.c`**. The example has been extended to support nested exceptions. Different kinds of exceptions are represented by different integer values, specified in the **`ExceptionType`** enumeration in **`c_except.h`**.

The implementation uses C macros to provide a more exception-like syntax, which makes it easier to use, but more complicated to understand. C macros allow you to perform transformations on source code before it is compiled, but it is a very dangerous feature because there is little control of when and how macros are applied, or what they are allowed to do. Nonetheless, given this C-style exception-handling code:

```
TRY (
  printf("Enter your first number, the dividend:  ");
  n1 = read_double();

  printf("Now enter your second number, the divisor:  ");
  n2 = read_double();

  printf("The quotient of %lg / %lg is:  %lg\n", n1, n2, divide(n1, n2));
)
CATCH (NUMBER_PARSE_ERROR,
  printf("Ack!!  I couldn't parse what you entered!\n");
)
CATCH (DIVIDE_BY_ZERO,
  printf("Ack!!  You entered 0 for the divisor!\n");
  RETHROW;
)
END_TRY;
```

The **`TRY`**, **`CATCH`**, **`END_TRY`** and **`RETHROW`** macros in **`c_except.h`** translate the code into:

```
{
  jmp_buf env;
  int exception = setjmp(env);
  if (exception == NO_EXCEPTION) {
    start_try(&env);    /* Pushes the jmp_buf, for nested try/catch. */
    {
      printf("Enter your first number, the dividend:  ");
      n1 = read_double();

      printf("Now enter your second number, the divisor:  ");
      n2 = read_double();

      printf("The quotient of %lg / %lg is:  %lg\n", n1, n2, divide(n1, n2));
    }
    finish_try();       /* Pops the jmp_buf, for nested try/catch. */
  }
  else if (exception == NUMBER_PARSE_ERROR) {
    printf("Ack!!  I couldn't parse what you entered!\n");
  }
  else if (exception == DIVIDE_BY_ZERO) {
    printf("Ack!!  You entered 0 for the divisor!\n");
```

```
    throw_exception(exception);    /* Generated by RETHROW macro. */
  }
  else {
    /* Exception wasn't handled by any catch-block.
     * Propagate to the next enclosing try/catch block.
     */
    throw_exception(exception);
  }
};
```

Notice that when we enter the **`try`**-block, we call **`setjmp()`** and use the result to tell if an exception has occurred, and if so, which **`catch`**-block to handle it with. If no exception has yet occurred, we execute the body of the **`try`**-block, calling **`read_double()`** and **`divide()`**, either of which may throw an exception using the **`THROW(e)`** macro. This macro simply calls the **`throw_exception()`** function.

If an exception is thrown, the **`throw_exception()`** function calls **`longjmp()`** to jump to the closest enclosing **`try`**/**`catch`** block. This will cause the **`setjmp()`** function to "return a second time" (you know the *real* story now), with a result that indicates the actual exception that occurred. This causes the appropriate handler to be invoked, or if there is no handler for that exception type, the final **`else`**-clause causes the exception to be propagated to the next enclosing **`try`**/**`catch`** block. (Or, if there is none, the program will be terminated.)