TCC CSCI-2843 C++ Programming Language

Lecture 9 Notes

9.1 Exception Handling

9.1.1 Non-exceptional Error Checking

Consider different methods procedural programming typically provides to communicate errors from one level of a program to another:

1) A return value from a function.
2) A global "status" that can be inspected by any and everyone.
3) A "status" variable passed in by pointer/reference.

9.1.1.1 Return Values

An example of (1) would be checking the return value from an invocation of a stream operator:

```
if (!getline(stream, str, '\n')) { … error … }
```

Another example would be the old fopen call to open a file which uses a null pointer to indicate "failure".

```
FILE* fp = fopen("some_data.dat", "rw");
if (fp == NULL) { … file wasn't open … }
```

9.1.1.2 Global Status

An example of method (2) is the POSIX errno:

```
// include to gain access to the extern int "errno"
#include <sys/errno.h>

errno  = 0;
read(socket, buffer, sizeof(buffer));

if (errno != 0) { … error … }
```

9.1.1.3 Status Passed By-Reference

An example of method (3) is passing a **bool** into a function to receive a fail/success status:

```
int f(const std::string& s, bool& status);

bool status = false;
int x = f("1234), status);
if (!status) { … error … }
```

This is done when the return value of the function is already used to convey other information from the function call.

9.1.1.4 Issues

Each one of the above methods has one common limitation: they **all** require explicit checks after the function call to be of any use. This means that there is **always** the opportunity for the caller of a function to **completely ignore** any status that is returned:

```
getline(stream, str, 256, '\n');   // return value not checked – operation COULD have failed
…
FILE* fp = fopen("some_data.dat", "rw");

fread(fp, buffer, sizeof(buffer));          // could cause the program to crash if the file was not open (fp ==
nullptr)
…
```

```
read(socket, buffer, sizeof(buffer));       // socket could be closed or read could have failed
```

These types of methods all rely on "programmer discipline" to work properly, and we all know that is an oxymoron.

Consider some other error detection methods within code. When robustly including error checking, about a third to one-half (or more!) of the code can end up being error handling related:

```
int f() {
    std::ifstream f{"whatever.dat"};
    if (!f) {
        … handle error…
        return 1;
    }
    std::string str;
    for (;;) {
        if (!getline(f,  str, '\n')) {
            if (!f.eof()) {
                break;
            }
            if (f.fail()) {
                … handle failure …
                return 2;
            }
            if (f.bad()) {
                … handle "bad" stream
                return 3;
            }
        }
        … "normal" processing
    }
    return 0;
}
```

Many programmers would dangerously forgo the error checking:

```
int f() {
    std::ifstream f{"whatever.dat"};
    std::string s;
    while (getline(f, str, '\n') {
        … "normal" processing
    }
    return 0;
}
```

While this is far fewer lines of code, it also is less robust and gives less-than-desirable diagnostics when something does go wrong, indeed it just bails from reading the stream at the first error **or** end of stream with **no** error detection whatsoever.

Note that even though in the first (robust) example, we can go through all the trouble of detecting the error and returning a "code" or "status", but there is **no** guarantee that the caller of **our** function will ever pay any attention to it.

Worse, we sometimes see convoluted error handling, such as the "ok" flag method:

```
bool f() {
    bool ok = true;
```

```
    ... insert many lines of convoluted and twisted code here, setting ok to false if there is an error
    return ok;
}
```

The more convoluted the intervening code, the more the chance that the "ok" flag might be set inadvertently, or more likely, not set when it is supposed to be, giving false positives and negatives. At the very least this sort of code is extremely hard to trace through during debugging or extension.

Per all of these motivations, what we would like is a facility that would do three things:

1) Standardize on the way errors are detected and checked
2) **Require** that once an error is detected that it **must** be handled (i.e. **cannot** be ignored)
3) Reduce the syntax for detecting and handling errors, thus "unconvoluting" the code.

The C++ exception facility gives us all of these.

9.1.2 Exceptions

Like many modern languages, exceptions are based on a **try**/**throw**/**catch** model where we "**try**" an operation, and if there are errors, we "**throw**" an exception, which we "**catch**" with code that handles the error. The general syntax is

```
...
try {
    ...
    if (an error occurs) { throw some exception object or type T; }
    ...
} catch (a object or reference to an object of type T) {
    ... handle the exception
}
...
```

What can be thrown as an "exception object"? **Any** object, including primitive types (e.g. **int**, **double**) or objects constructed from classes. For instance, we can do the following:

```
...
try {
    ...
    if (an error occurs here) { throw 1; }
    ...
    if (an error occurs here) { throw 2; }
    ...
} catch (int code) {
    std::cerr << "an error occurred: " << code << std::endl;
}
...
```

We can even use strings:

```
...
try {
    ...
    if (there is an error) { throw std::string("something's wrong"); }
    ...
} catch (const std::string& s) {
    std::cerr << s << endl;
}
```

```
...
```

We can have multiple catch blocks following a try block to catch multiple types of exceptions:

```
...
try {
    ...
    if (an error occurs here) { throw 1; }
    ...
    if (an error occurs here) { throw "something's wrong"; }
    ...
} catch (const std::string& s) {
    std::cerr << s << std::endl;
} catch (int code) {
    std::cerr << "an error occurred: " << code << std::endl;
}
...
```

This is similar to an **if**/**else if**/**else** … such that depending on the type of the object that is **throw**n, the corresponding **catch** will be invoked to handle it. The important thing to note is that it will always pick the **first catch** that matches the type of the object **not the "best"** type.

```
...
try {
    ...
    if (an error occurs) { throw 1; }
    ...
} catch (double r) {
    ...
} catch (int i) {
    ...
}
...
```

This will always catch the **int**, 1, in the **double catch** clause because **double** is type-compatible with **int** and is the **first** that matches, even though there is a catch of **int** that matches **better**.

To catch **any** exception object, we can use the … (literally, three dots) in the **catch**:

```
try {
    if (error) { throw 1; }
    if (another error) { throw "problem"; }
} catch (...) {
    cout << "something went wrong" << endl;
}
```

Since there is no name given to the object that is caught, you cannot query anything to help diagnose the problem. That is why this is usually used as a "last resort" or "catch-all" mechanism when you don't need to know or care about the specific exception. Obviously, this "catch-all" needs to go as the last catch clause when present, since it will match **any** object thrown.

Some things to note about exceptions:

1) **Any** object may be thrown as an exception object.
2) If the object is a non-trivial type, we catch it by a (not necessarily **const**) reference to prevent a copy.

3) Every object within the **try**'s scope that has been fully constructed up to the point that an exception is **throw**n is destructed in the **opposite order** than they were constructed before control is transferred into the **catch** body.
4) Multiple **catch** clauses may be placed after a **try** block (there **must** be at least one), allowing us to catch multiple types of exceptions from one scope.
5) An exception object is always caught by the **first** catch clause that matches its type, **not** the **best** match of its type.
6) Use the **catch** (…) to catch **any** exception.

9.1.3 Unwinding

As mentioned before, objects within a **try** block that are constructed before an exception is thrown are destructed before a **catch** clause is entered. This is true even if an exception is thrown from one function and caught in another:

```
void f() {
    std::string u = "in f";
    …
    if (some error) { throw 1; }
    …
}

void g() {
    std::string s = "in g";
    try {
        std::string t = " in the try";
        …
        f()
        …
    } catch (int code) {
        std::cerr << "code: " << code << std::endl;
    }
}
```

If f throws the **int** exception, the exception mechanism notes that there is no immediate **try**/**catch** blocks surrounding it, so therefore it will destruct **all** automatic objects (local variables) within f that have been constructed so far (e.g. the string, u) and **unwinds** the stack as if it was executing **return**, but instead of returning to the code immediately following the function call, keeps searching for a matching **catch** clause, destructing objects as needed.

In this case, it finds the catch of an **int**, but must destruct all of the automatic objects within **that try** block, which includes the string, t. If there were still no matching **catch** clause, **all** of g's automatic objects would be destructed and g would be unwound and the search would continue. This stack **unwinding** process ensures that everything is cleaned up by the time a matching **catch** clause is entered.

What if there is never any matching **catch** clause found by time main is unwound? The program is terminated with an "uncaught exception" status. Since this is tantamount to a crash, it behooves us to **at least** provide a "last-chance" catch in main:

```
int main() {
    try {
        … do all of your other processing
    } catch (…) {
        std::cerr << "Exception caught in main" << std::endl;
        return 1;
```

```
        }
    }
```

9.1.4 Nontrivial Exception Objects

Remember that since we can throw **any** type of object, we can throw an object of our own type. We've already seen that we can throw a std::string, so why can't we build our own class that encompasses information about an exception? Why not, indeed?

```
class my_exception {
private:
    std::string description;
    int code;
public:
    my_exception(const std::string& init_description, int init_code) :
        description{init_desrciption},
        code{init_code}
    { }

    std::string get_description() const { return description; }

    int code() const { return code; }
};
```

This can be used like

```
...
try {
    ...
    if (some error) { throw my_exception{"something's wrong", 1}; }
    ...
    if (some other error) { throw my_exception{"something else is wrong", 2}; }
    ...
} catch (my_exception& e) {
    std::cerr << e.get_description() << ": " << e.code() << std::endl;
}
...
```

Note the anonymous construction of our exception objects. This keeps copying to a minimum and makes our error detection clean and succinct.

9.1.5 The std::exception Class

The standard library provides a base class for many useful types of constructors and several types of standard exception derivations. To get these exception classes we use

```
#include <stdexcept>
```

The std::exception class is used by the standard itself for a hierarchy of exceptions:

```
exception
    bad_alloc
    bad_cast
    bad_exception
    bad_typeid
    ios_base::failure
    logic_error
```

```
        domain_error
        invalid_argument
        length_error
        out_of_range
    runtime_error
        range_error
        overflow_error
        underflow_error
```

The std::exception class looks like this:

```
class exception {
public:
    virtual const char* what() const;
};
```

This doesn't seem like much, and what there is a little confusing, so let me explain.

Since this is supposed to be a base class, it provides a **virtual** function, what. This function's sole purpose is to return a C-style (nul-terminated) string that indicates what exception was thrown. Why a C-style string? Because one of the types of exceptions that can be thrown is when memory runs out. If it returned std::string, the operation of throwing a std::exception would require dynamic memory allocation (via std::string), which is what caused the exception to begin with. This would not be good, so it returns a **const char*** to a nul-terminated string as not to **require** dynamic memory. That doesn't mean that we **can't** use dynamic memory in **our** exception classes, though – just not if memory was the original problem. The what function is a **const** member function, meaning that it is merely a read-accessor to get the contents of the "what" string.

Remember that when we inherit from the base class we get two things (along with other things)

1) We get to overload any virtual functions
2) We get to use objects of a derived type anywhere an object of the base type is used by reference or pointer.

We can take advantage of all of this – the standard already does.

9.1.6 The std::runtime_error Exception

The std::runtime_error class is one of the more useful exceptions in that it allows us to give it a std::string on its constructor and that string will be returned (through a pointer, of course) as the result of the what function.

```
#include <stdexcept>
...
try {
    ...
    if (some error) { throw std::runtime_error{"uh oh"}; }
    ...
} catch (std::exception& e) {
    std::cerr << e.what() << std::endl;
}
...
```

Note that because std::exception is a base for all standard exceptions, and is therefore a more general type that the derived types, we must be careful of the order that they appear in catch clauses:

```
...
try {
```

```
        ...
    } catch (std::exception& e) {
        ... this catches all std::exception derived types
    } catch (std::runtime_error& e) {
        ... this will never be executed
    }
    ...
```

A smart compiler might be able to warn you about this, but don't hold your breath.

9.1.7 noexcept Functions (previously throw())

Sometimes when code can be guaranteed to **not** throw an exception, the compiler can produce more efficient code or at least use fewer resources generating the code. Furthermore, it would be nice for humans reading the code to understand that it is what is called "exception safe" (i.e. it will not, in any circumstance, throw an exception). This is what the **noexcept** keyword if for, and since it is used to mark functions of this guarantee similarly to **const**, the keyword is placed in the same spot as **const** as part of a function's signature:

```
void f() noexcept { ... }
```

Note – **const** can only apply to member functions, but **noexcept** can apply to **all** functions.

Member functions can be both const and noexcept (as is often the case with trivial accessors):

```
class car {
private:
    int mpg_;
    ...
public:
    ...
    int mpg() const noexcept { return mpg_; }
    ...
};
```

When the mpg member function is invoked, the caller is guaranteed that no exception will be thrown. The compiler can use this information to produce more streamlined code.

Be careful of labelling code **noexcept** when it is not. One common problem (that has even plagued the Standard) is to label functions returning strings as noexcept. It is possible that merely copying a std::string will throw an exception if there is a problem allocating member (remember std::string by default gets its memory from the freestore, which can run out of member, even if this unlikely during most reasonable applications' lifetimes).

9.1.8 A Derived Exception Example

As hinted at earlier, we can inherit from std::exception or any other standard exception class for that matter and provide our own functionality, but keep with the standard interface.

```
#include <stdexcept>
#include <sstream>
#include <string>

class my_exception : public std::exception {
private:
    std::string file;
    int line;
    std::string description;
```

```
        std::string what_string;
    public:
        my_exception(
            const std::string& init_file,
            int init_line,
            const std::string& init_description
        ) :   file{init_file},
            line{init_line},
            description{init_description}
        {
            std::ostringstream s;
            s << file << "@" << line << ' ' << description;
            what_string = s.str();
        }
        std::string get_file() const { return file; }
        std::string get_line() const { return line; }
        std::string get_description() const { return description; }
        const char* what() const { return what_string.c_str(); }
};
```

Note the use of the c_str member to turn the std::string into a C-style string. We could use this like

```
...
try {
    ...
    throw my_exception{__FILE__, __LINE__, "first exception"};
    ...
} catch (std::exception& e) {
    std::cerr << e.what() << std::endl;
}
...
```

9.1.9 Rethrowing Exceptions

From within a **catch** clause, if you decide that you cannot handle the exception that was caught, you can rethrow the exception that you just caught by simply using

```
throw;
```

This allows a **catch** at an outer level to handle an exception if the current **catch** cannot.

9.1.10 Exceptions Enclosing Arbitrary Scopes

It is possible to surround most scopes with a **try**/**catch**.

9.1.10.1 Function bodies:

```
void f() try {
    ...
} catch (...) { ... }
```

This allows any exception in the body to be caught by a **catch** immediately following the function's body, meaning that one can enforce that a function catch an exception without the possibility of it leaking out.

9.1.10.2 Loops

```
for (int i = 0; i < 10; ++i) try {
```

```
    ...
} catch (...) { ... }

while (condition) try {
    ...
} catch (...) { ... }
```

With loops, any exception thrown during an iteration can be caught in the **catch**, handled, and then the next iteration will continue (as long as the catch doesn't **throw** or rethrow).

9.1.10.3 If

An **if** statement can use a **try**/**catch** around its scope:

```
if (condition) try {
    ...
} catch (....) {
}
```

As strange as it may seem, we can do this even using an **if**/**else**

```
if (x == 0) try {
    std::cout << "x == 0\n";
    throw "huh";
} catch (...) {
    std::cout << "then\n";
} else try {
    std::cout << "x != 0\n";
    throw "wha";
} catch (...) {
    std::cout << "else\n";
}
```

9.1.10.4 Other Scopes

This works with just about any other scoped statement, excluding **switch**.

9.1.11 Exceptions in Constructors

Exceptions are particularly useful in constructors, especially since a constructor has no good way (such as a global indicator or dedicated "status" member) of returning a status to indicate that the object was constructed properly. By throwing an exception within the body of a constructor, we prohibit the object from ever being constructed in the first place.

```
class X {
private:
    std::string s;
public:
    X(const std::string& init_s) : s{init_s} {
        if (s.empty()) { throw std::runtime_error("empty string passed into X"); }
    }
};

int main() {
    try {
        X y{""};
        ...
```

```
        } catch (std::exception& e) {
            std::cerr << e.what() << std::endl;
            return 1;
        } catch (...) {
            std::cerr << "oops" << std::endl;
        }
        return 0;
    }
```

Note that if the exception is thrown, the already-constructed member, s is destructed.

9.1.12 Exception Retrying Schemes

Many situations might call for re-attempting an operation once an exception is thrown. Consider this scheme for handling this situation:

```
    const int max_attempts = 3;
    ...
    for (int attempt = 1;; ++attempt) {
        try {
            ... attempt operation
            break;   // operation successful – get out of loop
        } catch (std::exception& e) {
            if (attempt >= max_attempts) {
                throw;   // merely rethrow the last exception (could throw something else)
            }
            ... possibly perform corrective measures or wait on user input to proceed
        }
    }

    // if we get here, the operation was successful within max_attempts attempts
    ... carry on
```

Just food for thought.

9.1.13 System Errors

C++11 provides a new exception, std::system_error, which allows the categorization of error codes as well as the capturing of a message. The std::system_error class makes use of other types:

- std::errc
- std::error_category

9.1.13.1 Error Code Enum

The std::errc enum contains names for POSIX (generic) error codes:

enum Value	POSIX Error Code
success	0,
address_family_not_supported	EAFNOSUPPORT,
address_in_use	EADDRINUSE,
address_not_available	EADDRNOTAVAIL,

already_connected	EISCONN,
argument_list_too_long	E2BIG,
argument_out_of_domain	EDOM,
bad_address	EFAULT,
bad_file_descriptor	EBADF,
bad_message	EBADMSG,
broken_pipe	EPIPE,
connection_aborted	ECONNABORTED,
connection_already_in_progress	EALREADY,
connection_refused	ECONNREFUSED,
connection_reset	ECONNRESET,
cross_device_link	EXDEV,
destination_address_required	EDESTADDRREQ,
device_or_resource_busy	EBUSY,
directory_not_empty	ENOTEMPTY,
executable_format_error	ENOEXEC,
file_exists	EEXIST,
file_too_large	EFBIG,
filename_too_long	ENAMETOOLONG,
function_not_supported	ENOSYS,
host_unreachable	EHOSTUNREACH,
identifier_removed	EIDRM,
illegal_byte_sequence	EILSEQ,
inappropriate_io_control_operation	ENOTTY,
interrupted	EINTR,
invalid_argument	EINVAL,
invalid_seek	ESPIPE,
io_error	EIO,
is_a_directory	EISDIR,
message_size	EMSGSIZE,
network_down	ENETDOWN,
network_reset	ENETRESET,
network_unreachable	ENETUNREACH,

no_buffer_space	ENOBUFS,
no_child_process	ECHILD,
no_link	ENOLINK,
no_lock_available	ENOLCK,
no_message_available	ENODATA,
no_message	ENOMSG,
no_protocol_option	ENOPROTOOPT,
no_space_on_device	ENOSPC,
no_stream_resources	ENOSR,
no_such_device_or_address	ENXIO,
no_such_device	ENODEV,
no_such_file_or_directory	ENOENT,
no_such_process	ESRCH,
not_a_directory	ENOTDIR,
not_a_socket	ENOTSOCK,
not_a_stream	ENOSTR,
not_connected	ENOTCONN,
not_enough_memory	ENOMEM,
not_supported	ENOTSUP,
operation_canceled	ECANCELED,
operation_in_progress	EINPROGRESS,
operation_not_permitted	EPERM,
operation_not_supported	EOPNOTSUPP,
operation_would_block	EWOULDBLOCK,
owner_dead	EOWNERDEAD,
permission_denied	EACCES,
protocol_error	EPROTO,
protocol_not_supported	EPROTONOSUPPORT,
read_only_file_system	EROFS,
resource_deadlock_would_occur	EDEADLK,
resource_unavailable_try_again	EAGAIN,
result_out_of_range	ERANGE,
state_not_recoverable	ENOTRECOVERABLE,

stream_timeout	ETIME,
text_file_busy	ETXTBSY,
timed_out	ETIMEDOUT,
too_many_files_open_in_system	ENFILE,
too_many_files_open	EMFILE,
too_many_links	EMLINK,
too_many_synbolic_link_levels	ELOOP,
value_too_large	EOVERFLOW,
wrong_protocol_type	EPROTOTYPE

9.1.13.2 Error Categories

Error categories provide the main interface

```
class error_category {
public:
    virtual ~error_category();
    error_category(const error_category&) = delete;
    error_category& operator=(const error_category&) = delete;
    virtual const char* name() const = 0;
    virtual error_condition  default_error_condition(int error_value) const;
    virtual bool equivalent(int code, const error_condition&) const;
    virtual bool equivalent(const error_code&, int condition) const;
    virtual string message(int error_value) const = 0;
};
```

The standard provides two categories:

```
const error_category& generic_category();
const error_category& system_category();
```

The generic category contains the error codes available to all implementations, while the system category are system-specific errors.

9.1.13.3 The std::system_error Class

The std::system_error class is provided in the separate <system_error> header and inherits from std::runtime_error:

```
class system_error : public runtime_error {
public:
    system_error(error_code, const string& what_arg);
    system_error(error_code, const char* what_arg);
    system_error(error_code);
    system_error(int code, const error_category&, const string& what_arg);
    system_error(int code, const error_category&, const char* what_arg);
    system_error(int code, const error_category&);

    const error_code& code() const noexcept;
    const char* what() const noexcept;
};
```

To throw a system error, we can do something like

```
throw std::system_error{ENOENT, std::generic:_category()};
throw std::system_error{std::errc::no_such_file_or_directory, std::generic:_category()};
throw std::system_error{GetLastError(), std::system_category()};
```

Because std::system_error inherits from std::runtime_error, which inherits from std::exception, all existing code that catches these bases will also catch std::system_error, allowing easy adoption … if one had been using the Standard's exception classes all along.

9.2 Application Objects

Encapsulation techniques may be applied to the entire application organization. Consider the typical case where an application requires "application-scoped" objects that need stored over the lifetime of the application. Up until now, we would be tempted to make these objects **extern** (aka. global), but this is not well encapsulated as we cannot control access to that data. Making such data "local" to main is not always feasible because it available to other functions is limited to passing as arguments that may not scale very well and is overall not maintainable as applications grow larger.

We can use encapsulation to do better than **extern** or main-local data by providing an **application class** to represent the application as a whole:

```
class application {
private:
    … application-scoped objects
public:
    application(… take in whatever arguments you need to initialize the application…) {
        … initialize application
    }

    void run() {
        subtask1();
        subtask2();
        …
    }
private:
    void subtask1() { … }

    void subtask2() { … }
};

int main() {
    try {
        application app{…supply application initialization arguments…};

        app.run();
    } catch (std::exception& e) {
        … handle application-level exceptions here
        return 1;
    }
    return 0;
}
```

Now the application object that is constructed in main can hold and control the state of all application data using all of the encapsulation techniques that we have learned so far. An application object may take in

arguments to its constructor(s) to provide application-level configuration. Configuration information may be read in from a configuration file, stored in a database, or (as we explore in the next section) taken in as arguments to the application itself.

We can be "clever" with the way we invoke application objects. Consider overloading the function call operator, **operator**(), to start the application and use the return value from this operator as the return value of the application.

```
class application {
    ...
    int operator()() {
        ... return 0 for "OK" or non-zero for other error
    }
};

int main() try {
    application app{};
    return app();
} catch (std::exception& e) {
    return 1;
}
```

9.3 Command Line Arguments

The full signature of main is as follows:

```
int main(int, char**);
```

The standard allows the arguments to be omitted (as we have done so far), but we can use these to pass arguments into an application from the command line. We usually provide names of the arguments (out of convention, not dogma) as follows:

```
int main(int argc, char* argv[]) {
    ...
}
```

These names stand for "**arg**ument **c**ount" and "**arg**ument **v**alues" respectively. Remember this only a convention, you can name them whatever you want. The "argv" argument is an array of pointers to nul-terminated "C" strings as taken from the command line, usually determined by items separated by whitespace. The array always has as least one entry – argv[0], which is always the name of the executable as it was invoked from the command line. Any additional command line arguments start in argv[1]. The "argc" argument is the number of items in the array, and as noted, is always at least 1.

Sometimes you will see argv declared as

```
char** argv
```

as this is **equivalent** to passing a **two-dimensional arra**y or an **array of pointers** (due to the pointer/array relationship in C/C++).

The following program prints any and all command line arguments:

```
int main(int argc, char* argv[]) {
    for (int i = 0; i < argc; ++i) {
        std::cout << argv[i] << '\n';
    }
}
```

If this were compiled into an executable named, "zyx.exe", and invoked with the following command line:

```
zyx abc 123 Hello world
```

this would produce the output:

```
zyx
abc
123
Hello
world
```

Consider the use of command line arguments with application objects:

```
class application {
...
public:
    application(int argc, char**argv) {
        ... configure the application
    }
    ...
    int run() { ... application code ... }
};

int main(int argc, char** argv) try {
    application app{argc, argv};

    return app.run();
} catch (std::exception& e) {
    std::cerr << e.what() << std::endl;
    return 1;
} catch (...) {
    std::cerr << "unknown exception" << std::endl;
    return 2;
}
```

This way, we can configure an application object directly with arguments passed in at runtime.