Dynamically Registering C++ Exception Handlers:
Centralized Handling of C++ Exceptions in Application Framework Libraries

by

Nathaniel H. Daly

A thesis submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science
(Honors Computer Science)
in the University of Michigan
2013

Thesis Committee:

Professor David Kieras, Advisor
Professor David Paoletti, Second Reader

Abstract

The C++ exceptions mechanism enables modularized programs to separate the reporting of exceptional conditions from the handling of those exceptional conditions into the locations where those actions can best be performed. It is often the case in such modular applications that the module capable of detecting an exceptional situation cannot know how it should be handled, and so C++ exceptions allow the module that invoked the operation to react to the condition. Libraries developed separately from the applications using them are a perfect example of modularized code, and thus a perfect candidate for using C++ exceptions, however many of the application framework library paradigms that have become commonplace do not provide adequate exception support to allow for good programming style with regard to catching and handling exceptions. In these libraries, there often exist multiple callback-style entry points that invoke application-specific operations, all of which require exactly identical exception handling, yet there is no way to collect this code together. Instead, the exception handling code must be duplicated across each entry point. In this paper, I will explore the best solutions to this problem currently provided by such frameworks; I will propose an additional solution to this problem: a method for allowing a program to register exception-handler operations templated by exception type; and finally, I will identify any impacts that the newly released C++11 standard and the currently evolving C++14 standard have on the problem.

Contents

1. Introduction

1.1. Motivation for Using Exceptions

This paper focuses on exceptions and exception handling in the context of application framework libraries; I will begin by first providing a high-level overview of the C++ exception handling mechanism.

The exception mechanism was designed to provide an alternative approach to handling errors that arise during the execution of a program, or, more specifically, to handling *exceptional conditions* [34 (p.45)]. Exceptional conditions are situations encountered during a program's execution in which an operation detects an error, and the following conditions are true [19 (p.684), 32 (p.344-345)]:

1. The operation doesn't know how to handle the error,
2. The invoker of the operation wasn't able to detect the error before invoking it, and
3. Some invoker higher up in the chain of invokers might know how the error should be handled.

These conditions are necessary to consider a condition *exceptional*, and without any of them, exceptions are not required. (Consider: if 1. is false, the operation can simply handle the error itself; if 2. is false, the operation never should have been invoked; if 3. is known to be false, the application may simply terminate.) Before exceptions were introduced to mainstream programming languages, there were (and still exist today) many disparate ways of handling exceptional conditions. The most common strategy still widely used in C and C++ is to use a status variable to indicate an operation's success or failure. This could either be a parameter passed in to the function or a return value returned from the function [19 (p.684), 34 (p.45)]. C++ exceptions are significantly safer from programming mistakes (an exception cannot be ignored like a status variable can), they separate

error code from "normal" execution code, and they often achieve all of this with the same amount of or even less error-handling code and runtime costs than the status variable method [13 (p.34)]. C++ exceptions are a very useful and well-established tool that is almost always the correct choice for handling exceptional conditions.

1.2. Application Frameworks

Exception handling is only really effective if the set of all possible types of exceptions that may be thrown is known. Without that information, it is difficult to correctly respond to a thrown exception. The organization of one increasingly common type of library, an *application framework*, results in exactly this undesirable situation.

While exceptions can be very effective in small applications, they are even more useful in large, modular applications -- most notably if the modules are developed separately. Modular applications with many separately designed pieces, often separately designed small libraries, have been a major motivator for the design of C++ exceptions: "exceptions were considered essential for error handling in programs composed out of separately designed libraries" [34 (p.45)]. With smaller libraries, it is most often the case that the application maintains control of execution at the top level and calls into the libraries to perform certain tasks. In these situations, libraries excel at C++ exception support. If a library function encounters an exceptional condition, it can easily throw a detailed, library-specific exception that has information about the error, allowing the application to handle the exceptional condition.

For example, an application might be making use of the **Boost.Regex** library [28], which implements regular expressions in C++. The application could construct an instance of `Boost::basic_regex` from an input string. If this string is not a

valid regular expression, `Boost::basic_regex::basic_regex()` will throw a `Boost::regex_error` exception [28].

```
int main() {
    string s;
    try {
        cin >> s;
        basic_regex reg(s.c_str()); // May throw regex_error if invalid
        // ... use reg to do pattern searching ...
    }
    catch (Boost::regex_error &err) { // basic_regex() threw an error
        cout << "Error: '" << s << "' is not a valid regex. Try again.\n";
    }
}
```

Here, the application's `main()` is the invoker, and the `Boost::basic_regex` constructor is the operation being invoked. This is a textbook scenario for using exceptions: 1. the invoker, `main()`, cannot know if the inputted string is a valid regex -- that is a hard task, which is why the task is delegated to a library in the first place; 2. the **Boost.Regex** library cannot know how a bad input string should be handled -- should it ask the user?, should it terminate?, etc; 3. the application module *does* know how to handle the error -- it will ask the user for another string. Given that all three conditions are satisfied, the **Boost.Regex** library makes use of C++ exceptions and throws `Boost::regex_error`[1] when the exceptional condition occurs.

As seen above, exceptions are very straightforward when a library is used as module in a broader application. However, this is often not the case with many larger libraries. Larger libraries that are used to build an application often contain the main body of the application, and the application-specific code then becomes the small modules that the *library calls into*, instead of the other way around. These libraries are commonly referred to as *application frameworks*. This paper will define an application framework to be any large library that utilizes the above structure, i.e., a library that is organized as the main structure of an application

[1] `Boost::regex_error` is a type defined by boost intended to be thrown as exceptions.

and calls short pieces of application-specific code only following certain events. The application framework makes up the majority of an application built with it, and it interacts with the application by making calls into supplied application-specific code when certain events occur. A good, though neither necessary nor sufficient, indicator that a library is an application framework is that it cannot be used in conjunction with another application framework. In this situation, it now becomes the *library developers* who must be concerned with what exceptions the application-specific code may throw, instead of the *application developer*[2] handling any exceptions the library functions may throw. This is a much harder task. The exception types thrown by a library do not change and can be found listed in a manual. At the other extreme, the exception types thrown by an application depend on its function, and will change from application to application -- or even during its execution. This is what makes exception handling in application frameworks an interesting topic for study.

There are some number of different examples of application-framework-style libraries, but *GUI toolkits*[3] are the most important that exist today, simply because they are common and relevant. As desktop and mobile software development shifts its focus towards Apps, and users continue to expect cohesiveness with other Apps on their devices, GUI toolkits are becoming common and necessary to learn. Figure 1 on the following page presents a common layout of applications built with a GUI toolkit. The lighter-colored boxes are application-specific code. It is easy to see how the application framework calls down into the application-specific code, instead of the other way around. The problem and solu-

---

[2] I distinguish an application from the library it uses and an application developer from the developers of the library. An application developer does not have access to the code in the library, and the library developer cannot know what sorts of applications will use the library.

[3] A GUI toolkit is simply an application framework that provides an Application Programming Interface (API) to access a set of *widgets,* or small units of an application's Graphical User Interface (GUI). For a good high-level overview, see https://en.wikipedia.org/wiki/Widget_toolkit

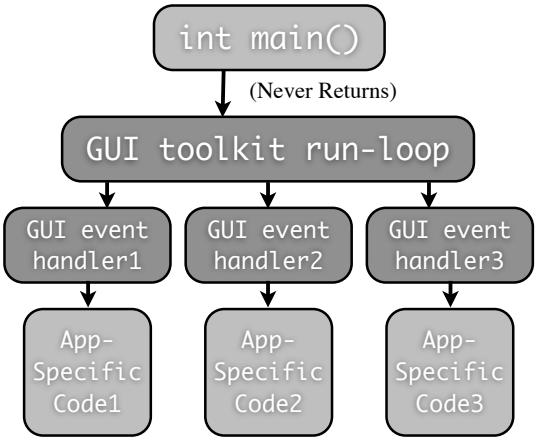

Figure 1. A GUI toolkit Application Framework's Call Graph.
Application-specific code is only reached through calls from the top-layer GUI toolkit. On a GUI event, the toolkit calls the appropriate event handler, which calls into application-specific code.

tions addressed in this paper apply to exception handling in all application frameworks, but in order to narrow its scope, this paper will focus specifically on exception handling in a GUI toolkit. Other application frameworks around which one could frame this discussion include network server frameworks, daemon frameworks, and game engine frameworks; GUI toolkits were chosen mainly due to my familiarity.

1.3. The Problem: Repeated Exception Handling Code

Recall that an application framework remains at the top-level of an application's call stack throughout an application's execution. Application-specific code is run only from specific entry-points, usually implemented as either *callbacks* or *overridden virtual functions*. These application-specific entry-points may throw an exception, but the application framework cannot know what types may be thrown (see §1.2 above). This makes traditional exception handling in the application framework's code impossible. Therefore, in order to prevent an exception from terminating the program, if code called from within an application's entry-point throws an exception, that exception must be caught before propagating into application framework code. A typical GUI application may have many such

entry-points, and so the application developer will be forced to repeat the exception-handling code across each of them. The exception handling code will likely be identical, and must be repeated once for every application-specific entry-point. This is clearly undesirable: code duplication is almost always a problem [29 (p.23-25)]. Yet this is presently the state of things in the majority of C++ GUI toolkits.

To illustrate this problem, let us look at an example application, `MyNetworkApplication`, being developed with a GUI toolkit. Our GUI toolkit uses virtual functions to provide application-specific entry points, and it is these functions that we are concerned with. For example, the GUI toolkit might present the following simple class interface for a button. To help maintain clarity, this paper will adopt the convention that code examples which are part of a library will have a solid bar to their left, whereas application-specific code will not. All of the ideas presented in this paper will work with C++03, but C++11 functionality has been used throughout this paper where it can increase readability.

```
class GUIButton : public GUIViewController {
public:
    virtual void OnButtonPressed()
        { /* Derived classes should override this behavior */ }
    // ...
};
```

This class expects an application to derive its own button classes that perform application-specific code when the button is pressed. For example, the application developer might write the following button:

```
class HelloButton : public GUIButton {
public:
    virtual void OnButtonPressed() override {
        GUIWriteTextToScreen("Hello World\n");
    }
};
```

which, when pressed, writes the text "`Hello World\n`" to the screen using some other GUI library function. Less trivially, an application's button might start some process or modify some application data. If operations invoked from within `OnButtonPressed()` could throw an error, however, the implementation of the overridden function becomes complicated. One situation in which exceptions are likely to be thrown is if the application is developed using the Model-View-Controller (MVC) design pattern [18], in which the GUI toolkit is used to implement the View and Controller portions. With MVC, one reasonable button class might call into the Model to set up Model data or start a Model process. If the parameters to the Model's function call are invalid, the Model may throw an exception. Left unhandled, this would terminate the program. `LogInButton`, below, is an example illustrating this problem:

```
class LogInButton : public GUIButton {
public:
    virtual void OnButtonPressed() override {
        // Get user name and password from text boxes
        //  and attempt to log-in through the Model.
        model->log_in(username_text_box.text(),
                      password_text_box.text());
    }

    // ...

private:
    Model &model;     // pointer to Model; must be set in constructor
    std::weak_ptr<const GUITextBox> username_text_box; // read only
    std::weak_ptr<const GUITextBox> password_text_box; // read only
};
```

Here, `LogInButton` acts as the connection between the application's View-and-Controller (which are implemented by the GUI toolkit) and the application's Model. MVC is a perfect example of the correct conditions for using exceptions. In our example application, the Model's functions do not know what sort of GUI toolkit -- if any -- is being used, and therefore cannot know how to handle input

errors. The button entry-point function `OnButtonPressed()` similarly cannot know if the input is valid, or it would not have to call into Model code in the first place (for example, the `LogInButton` cannot know which username/password pairs are stored in the Model, because that is private data encapsulated by the Model). It does, however know how exceptions should be handled, and so it can wrap the call to `Model::log_in()` call in a try-catch, and handle the exceptions here. Of course, there might be many more such entry-points in the application, all of which take information from the Controllers and pass it to the Model, and/or take information from the Model and display it in the Views. Adding an entry-point requires duplicating the exception-handling code. This is an obvious problem.

Coming back to the example: the application developer knows the Model might throw an exception, and so any calls to `Model::` functions *must* be wrapped in a *try-catch-block*.

```
void LogInButton::OnButtonPressed() {
    try {
        model->log_in(username_text_box.text(),
                      password_text_box.text());
    } catch(ApplicationError &e) {
        // handle error... (probably notify user of the error)
    }
}
```

Every application-code entry point that could throw an exception (which is potentially all of them) must similarly be wrapped with a *try-catch* like the `LogInButton` above. This code repetition problem could become even worse if the entry-points may throw multiple exception-types, each requiring a different response. This is what the virtual function looks like if a call to model could throw three different exception classes, not just `ApplicationError`:

```
void LogInButton::OnButtonPressed() {
    try {
```

```
        // ... Attempt to log in to the Model ...
    }
    catch(UserInputError &e) {
        notifyUserOfInputError(e); // display error message
    }
    catch(NetworkError &e) {
        attemptToReconnectToNetwork(e); // try again or terminate
    }
    catch(std::exception &e) {
        gracefullyEndProgram(e); // likely a programming error, so terminate
    }
}
```

All 11 of the error-handling lines above will have to be repeated for every such application-specific entry-point that the application developer derives. Any changes to what exception types need to be caught and/or how they ought to be handled will require changing *every* place in the code that this is done. It would be desirable from the application developer's perspective to have all of the exception-handling code together in one centralized location. Unfortunately, the nature of the application framework makes this impossible, because the only application-specific code that can be called is accessed through the virtual function entry points. It would be desirable, then, if the application framework could provide a mechanism to enable the application developer to accumulate all the exception-handling code in one place, and this is the problem that is addressed by this paper.

2. Background

2.1. Current C++ GUI Toolkits Exception Support

Besides the the importance and frequency of GUI toolkits in the industry, this paper chooses to focus on GUI toolkits for another reason: current support for C++ exceptions is very poor. Section §6, Discussions, includes a discussion of possible reasons for this lack of support. Here, I will outline at a very high level the varying exceptions support across the major C++ GUI toolkits. A large part of the problem with exceptions support among GUI toolkits is that it is often very poorly documented. Since exception handling is rarely considered an important component of the library, it is often difficult to find complete information on its implementation. Therefore, it is possible that I have mischaracterized some of the toolkits below.

Figure 2 on the previous page provides a list of the major C++ GUI toolkits in use today. With the exception of Microsoft Foundation Classes (MFC), all are cross-platform GUI toolkits. I divide the major C++ GUI toolkits into three groups: those with no explicit support for C++ exceptions, those which support any exceptions derived from a supplied base class, and those with what I deem "good support" for exceptions.

No Explicit Support	Base Class Exceptions Only	Good Support
Juce [8]	VCF [14]	FLTK [31]
QT [11]	MFC [7 , 10]	wxWidgets [1]
Gtkmm [9]	CEGUI [35]	TnFox [17]
Ultimate++ [26]		
YAAF [36]		
GLUI [15]		

Figure 2. GUI Toolkits Listed by Exception Support.

By “good support”, I mean that -- albeit usually with a bit of finesse from the application developer -- the toolkit can be configured to catch any exception types that are thrown and can can be made to handle them in any way. These two requirements are also the requirements for the solution I present in this paper; the difference being that the solution in this paper requires the least amount of work from an application developer.

A toolkit that provides support for “base class exceptions only,” is one that *does* catch and handle exceptions, but only exceptions derived from a supplied base class. For example, MFC provides a base exception class, `CException`. When a `CException` is caught in the main run-loop, MFC will call the virtual function `CWinApp::ProcessWndProcException` with the exception object as a parameter. If the application developer chooses to override this function, the application can handle exceptions of a type derived from `CException` as well.

Finally, a toolkit that provides “no explicit support” is one that either simply ignores exceptions, or that catches and handles exceptions defined by the toolkit in a built-in way that an application cannot change. Unfortunately, this is a common situation in GUI toolkits: QT is compiled with exceptions completely turned off by default [16]; Gtk will terminate if exceptions escape application specific code [9]; Juce is the same [8]; and so on.

2.2. Current “Good Solutions”

As outlined in the previous section, FLTK, wxWidgets, and TnFox all provide “good support” for C++ exceptions. In this section I will describe the solutions presented by wxWidgets and FLTK. Unfortunately, time did not permit exploring TnFox’s solution; see Section §5 Further Work for more information.

The wxWidgets toolkit provides a good solution to prevent repeated exception-handling code. wxWidgets, like MFC described above, provides a virtual function that is called whenever an exception is caught in the main run-loop. Unlike MFC, however, this function, `wxApp::OnExceptionInMainLoop()`, is called when *any* exception is caught. It can then be overridden by the application developer to handle exceptions. The wxWidgets' documentation explains that this function will be called "if you decide to let (at least some) exceptions escape from the event handler in which they occurred. [...] This allows you to decide in a single place what to do about such exceptions." Following the example from wxWidget's documentation, an application can determine the type of exception that was thrown and how to handle it. In section §3.3.1 we discuss the implementation of this solution, as well as its benefits and drawbacks.

The FLTK library also provides a complete solution. FLTK allows an application to register an *event dispatch*, which is a function that wraps FLTK's built-in event handling functionality. By default, in FTLK an input event `event` that occurs in a window `window` is handled with a call to `Fl::handle_(int event, Fl_Window *window)`. However, if an application has registered an event dispatch function, it will pass `event` and `window` to the event dispatch instead. As directed by the documentation, if an application developer wishes to handle exceptions that are thrown out of application-specific code, he or she may provide an event dispatch function that wraps a call to `Fl::handle_(event,window)` with the appropriate try-catch blocks. The documentation provides the following example:

```
int myHandler(int e, Fl_Window *w) {
    try {
        return Fl::handle_(e,w);
    } catch(/* [(sic.) ...) {
        // ...
    }
}
```

```
main() {
    Fl::event_dispatch(myHandler);
    // ...
    Fl:run();
}
```

Through this bit of code, an application can dynamically register which exception types it would like to handle, and how it would like to handle them, thus freeing the application developer from repeating the exception handling code across each of the application's entry-points.

3. Solutions

3.1. High Level Solution

Recall the goal described in §1.3: it would be beneficial to application developers if an application framework library provided a way to write exception handling code only once, rather than having to repeat it for each application-specific entry-point. To see how this might be accomplished, we must look at an application framework's structure. Again, we will be using the example of a GUI toolkit to provide concreteness, though this discussion should apply to the majority of application frameworks.

As described in Figure 1 from §1.3, the main control structure for a GUI toolkit is the run-loop. Throughout an application's execution, the top-level of the function call-stack is always the run-loop, which calls down into application-specific code. A common format for GUI toolkits is to organize the run-loop as a polling style event handler [30]. A simplified version of this type of run-loop would look like this:

```
void GUIApp::run() {
    while (true) { // loop forever until the program exits

        Event input_event = // ...Check for input events...

        if (/* ...mouse was clicked... */) {

            // ...Find the widget that was clicked on...

            widget->handleMouseClick(input_event);
        }
        else if (/* ... */) {
            // ...
        }

        // ...
    }
}
```

Every iteration through the loop is a *frame* in the application. In each frame, any input event to the application is handled based on its type. If the input was a mouse-click, then the widget that was clicked on handles that input event. If the widget's derived type is an application-specific class, then the widget calls into application-specific code.

Returning to our earlier example of the `LogInButton` widget (defined in §1.3), let us assume that the widget which was clicked on is an instance of `GUIButton`. The `GUIButton` would handle the mouse click by calling its virtual function, `OnButtonPressed()`, which acts as an entry-point into application-specific code.

```
void GUIButton::handleMouseClick(click_event) {
    if (click_event.mouse_button == GUI_MOUSE_LEFT) { // if left-click
        OnButtonPressed(); // press the button; run app-specific code.
    }
}
```

If the derived type of `GUIButton` is a `LogInButton`, the call to `OnButtonPressed()` would polymorphically call `LogInButton::OnButtonPressed()`, which finally would call `Model::log_in()` (from §1.3).

Now, it is clear how an exception that escapes from `OnButtonPressed()` would be a problem. `GUIAPP::run()` has no exception handling, so the exception would crash through the while loop and terminate the program. In order for the GUI toolkit to be able to handle an exception and continue execution, the while loop needs to contain a `try-catch` block.[4] But which exception types should this `try-catch` catch? Remember, the motivation for adding this exception code is to group an application's exception handling code in a central location. Therefore, from the application developer's perspective, the most useful GUI toolkit run-loop would catch *exactly* those exceptions an application might throw, and would handle them in exactly the right way. For `MyNetworkApplication`, it would need to catch `UserInputError`, `NetworkError`, and `std::exception`.

```
void GUIApp::run() {
    while (true) { // loop forever until the program exits
        try {
            // ...Check for input events...

            if (/* ...mouse was clicked... */) {

                // ...Find the widget that was clicked on...

                widget->handleMouseClick(click_event);
            }

            // ...
        }
        catch(UserInputError &e) {     // <--------- *Problem*
            notifyUserOfInputError(e);
        }
        catch(NetworkError &e) {
            attemptToReconnectToNetwork(e);
        }
        catch(std::exception &e) {
            gracefullyEndProgram(e);
        }
```

[4] The `try-catch` must be *within* the while loop. If it were outside it, an exception *could* be handled, but there would be no way to continue execution, since we would have exited the run-loop.

```
    }
}
```

Of course, as indicated by the `*Problem*` comment above, this will not actually work: the GUI toolkit was developed separately from the application, and may not even be aware of the exception types the application wants to catch, let alone how to handle them. For example, both `UserInputError` and `NetworkError` are classes defined by `MyNetworkApplication`. Since the GUI toolkit was developed separately from the application, these classes could neither be defined nor declared at the toolkit's compile-time. Therefore, the toolkit cannot explicitly catch the exact types an application might throw. But it does still want these exceptions to be caught -- lest it terminate when they are thrown, so it should instead use a `catch(...){}` block. `catch(...)` will catch any exception type that is thrown, but the block does not have a variable through which to access to the caught exception.

However, we can use the `catch(...)` block to simulate catching all the types an application might throw. From within this catch-block we want to perform the same sort of exception handling that the application is currently performing in each of its entry-points. To achieve this, the application needs to be able to specify which exceptions it wants caught, and how it wants them to be handled. Then in order to handle these exceptions at runtime, the `catch(...){}` block needs to be able to (1) deduce the type of the exception that was thrown, (2) handle the exception appropriately based on its type, and (3) terminate if the exception wasn't handled.

```
void GUIApp::run() {
    while (true) { // loop forever until the program exits
        try {
            // ...Take in input...
```

```
            if (/* ...mouse was clicked... */) {

                // ...Find the widget that was clicked on...

                widget->handleMouseClick(click_event);
            }

            // ...
        }
        catch(...) {
            // 1) Deduce exception type
            // 2) Call any handlers that the application has associated
            //     with that type.
            // 3) If no handlers are associated, terminate.
        }
    }
}
```

The remainder of Section §3 describes how a library could implement `(1)`, `(2)`, and `(3)`.

3.2. Dynamically Deducing Exception Types

This section will introduce a largely unknown bit of C++ behavior that can be used to determine the type of a caught exception. In short, from inside a `catch(...){}` block, the type of the currently active exception can be deduced by entering a new `try{}` block, re-throwing the exception, and then re-catching the exception with a more specifically-typed catch block. In this paper I will refer to this behavior as the *catch-try-rethrow trick*.

As described above, a `catch(...)` statement will catch any kind of exception that is thrown. Unfortunately, the catch block will not have access to the exception as a variable. However, by re-throwing the exception and catching it again, we can enter a new catch block where the exception *is* referenced by a variable. In order for the new catch block to catch the exception, it must catch exactly the type of the thrown exception (or a reference to one of the exception's base-types). Be-

low is an example illustrating this behavior, with numbered comments providing an explanation.

```
try {
    throw /* ...Something... */;  // 0. exception object is allocated here.
}
catch(...) {
    // 1. From here, the exception type is unknown.

    try {  // 2. so we enter a new try-block to try again.
        throw;  // 3. rethrow the exception
    }
    catch(A &e) {
        // 4. the original exception was of type A.
    }
    catch(B &e) {
        // 5. the original exception was of type B.
    }
    catch(...) {
        // 6. we still do not know the type of the exception.
    }
}  // 7. the exception object is deallocated here.
```

The above code looks a bit outlandish, but it will deduce the type of the thrown exception. If the `/* something */` at comment `(0)` is replaced with an instance of type `A`, for example `throw A();`, then the exception will be caught again by `catch(A &e)`, and the block starting at comment `(4)` will be executed. If an instance of type `B` is thrown, the block containing comment `(5)` will be executed. If the exception thrown is of neither type, say `throw C();`, then the last block containing comment `(6)` will be executed, and the type of the exception would still be unknown. This trick, therefore, allows a function to check if a caught exception's type is any of a set of *known* exception-types. That is, the trick cannot be used deduce an unknown type. This is of course because C++ is a statically typed language, and so one can never refer to an object of an unknown type.

We can look to the ISO C++ Standard[5] to find a description of this behavior. First, we need to prove that the exception is still valid when we enter into the new try-block at comment `(2)`. Section §15.1.4 of the standard [4 (p.396)] verifies this fact (emphasis added by me):

> The memory for the exception object is allocated in an unspecified way, except as noted in 3.7.4.1. If a handler exits by rethrowing, control is passed to another handler for the same exception. **The exception object is destroyed after either the last remaining active handler for the exception exits by any means other than rethrowing**, or the last object of type std::exception_ptr (18.8.5) that refers to the exception object is destroyed, whichever is later.

The exception object is allocated and valid at comment `(0)`, and remains so until comment `(7)`, when its last active handler is exited. For a definition of what it means to be *active*, we look to section §15.3.7 [4 (p.398)]:

> A handler is considered active when initialization is complete for the formal parameter (if any) of the catch clause. [...] A handler is no longer considered active when the catch clause exits or when std::unexpected() exits after being entered due to a throw.

Following this definition, the exception allocated at comment `(0)` enters its first active handler at comment `(1)`, and is only destroyed after this handler exits, at comment `(7)`. This should be enough to convince the skeptical reader that the exception object stays active inside the second try-block. The other half to this behavior is the `throw;` statement. Section §15.1.8 [4 (p.397)]:

> A throw-expression with no operand *rethrows* the currently handled exception. The exception is reactivated with the existing temporary; no new temporary exception object is created.

This verifies that the exception rethrown at comment `(3)` will be exactly the exception originally thrown at comment `(0)`. Therefore, any catch statement that

---

[5] I am using the C++11 standard here only because it is the most recent, but very similar language exists in the C++03 standard [3 (p.299-302)], and the same conclusions can be drawn.

could have caught the original exception will successfully catch it after it is re-thrown.

Of course, written as above, this trick seems useless: if the types `A` and `B` are known to be possible exception types, they should have been included as possible catch blocks before the final `catch(...){}`. There doesn't appear to be any reason to nest the type deduction inside of the catch block. However, it becomes significantly less trivial if the contents of the `catch(...)` block above are moved into a function which is defined in a different module. Then, by recompiling one module, we can change what exceptions are caught by the other. For example:

```
// will try-rethrow-catch the currently active exception and handle
extern void deduceExceptionTypeAndHandle();


void do_something() {
    try {
        // ... possibly throw something ...
    }
    catch(...) {
        // 1. From here, the exception type is unknown.
        // So allow deduceTypeAndHandle() to figure it out and handle it.
        deduceExceptionTypeAndHandle();
    }
}


// IN SOME OTHER MODULE:
void deduceExceptionTypeAndHandle() {
    try {  // 2. so we enter a new try-block to try again.
        throw;  // 3. rethrow the exception
    }
    catch(A &e) {
        // 4. the original exception was of type A.
    }
    catch(B &e) {
        // 5. the original exception was of type B.
    }
    catch(...) {
        // 6. we really don't know the type of the exception.
    }
}
```

Here, just as before, if the original exception was either of type `A` or `B`, it will be successfully handled. However, now that `deduceTypeAndHandle()` is compiled separately from `do_something()`, we can dynamically change which types of exceptions `do_something()` catches and how they are handled, without having to recompile the original try-catch block. If the two modules are developed separately, the developer of `do_something()` promises to catch and handle exceptions in a way specified by the developer of `deduceTypeAndHandle()`.

3.3. Associating Handlers with Exception Types

The catch-try-rethrow trick described above can be used in a GUI toolkit's run-loop, allowing it to dynamically determine an exception's type and handle it. Doing so will allow the GUI toolkit to meet the desired behavior outlined in the "high-level solution" described in Section §3.1. There are at least two different approaches that could be taken to implement the behavior: one of them is simpler, but puts much more burden on the application developer; the other is more complex to implement, but it frees the application developer from any responsibility to reinvent the somewhat esoteric technique used above. I will begin by discussing the simpler approach simply to solidify the concept, and then will move on to the more complex solution, which is the solution that this paper recommends should be implemented by GUI toolkits.

3.3.1. Approach 1 - `wxApp::OnExceptionInMainLoop`

The first approach is taken from a real-world GUI toolkit, *wxWidgets*. To understand this solution, we start with the separated `do_something()` and `deduceEx-`

ceptionTypeAndHandle() functions from the previous section, and augment our GUI toolkit's exception handling until it resembles the wxWidgets solution.

Recall that by separating do_something() and deduceExceptionTypeAndHandle() into different modules, we allowed one module to define how the other module would handle exceptions. This same pattern can be used in the GUI toolkit's run-loop. Let us redesign GUIApp to utilize the separated functions, by making deduceExceptionTypeAndHandle() a virtual function. Then, an application-specific derived version of GUIApp can specify exactly how to handle exceptions.

```
class GUIApp {
public:
    void run() {
        while (true) { // loop forever until the program exits
            try {
                // ... might throw something ...
            }
            catch(...) {

                // Call virtual function to deduce the type of and
                //  handle the exception.
                deduceExceptionTypeAndHandle();
            }
        }
    }
    // ...

private:
    // ...

    // This function should be overridden in a derived class to specify
    //  exception handling behavior.
    // Overridden, it should like:      try{ throw; } catch(Type e){...}
    // Note: default behavior is to terminate on any exception.
    // REQUIRES: must be called from within a catch(...){ } block.
    virtual void deduceExceptionTypeAndHandle()
        { /* derived class should specify behavior */ }
};
```

Now, the application developer can provide his or her own exception handling code inside an overridden deduceExceptionTypeAndHandle(). Here, the code

would look very similar to the exception handling code from any of the application-specific entry points. For the `MyNetworkApplication` example from earlier sections, this would catch `UserInputError`, `NetworkError`, and `std::exception`.

```
class MyNetworkApplication : public GUIApp {
    // ...
private:
    // ...

    virtual void deduceExceptionTypeAndHandle() override {
        try {
            throw;
        }
        catch(UserInputError &e) {
            notifyUserOfInputError(e); // display error message
        }
        catch(NetworkError &e) {
            attemptToReconnectToNetwork(e); // try again or exit
        }
        catch(std::exception &e) {
            gracefullyEndProgram(e); // likely a programming error so exit
        }
        // If we made it here, we don't know the type, so let it terminate.
    }
};
```

Here, the author of `MyNetworkApplication` is specifying that the GUI toolkit should catch `UserInputError`, `NetworkError`, and `std::exception`, and handle them in different ways. If an exception of any of these types is thrown from application-specific code and reaches the GUI toolkit, it will be caught by the call to this application-specific `deduceExceptionTypeAndHandle()`.

This now matches the exception handling support in the wxWidgets GUI toolkit. There, `deduceExceptionTypeAndHandle` is named `OnExceptionInMainLoop`. wxWidgets' `wxApp` class has a virtual function `wxApp::OnExceptionInMainLoop()`, which is called exactly like `GUIApp::deduceExceptionTypeAndHandle()` above. An application can override this function in exactly the same way as the example above. When `wxApp`'s run-loop catches an exception, it will call the derived class's

`OnExceptionInMainLoop()`. wxWidget's documentation provides a sample application that implements this behavior. One of the files, `except.cpp`, shows how to implement custom exception behavior:

```
bool MyApp::OnExceptionInMainLoop()
{
    try
    {
        throw;
    }
    catch ( int i )
    {
        wxLogWarning(wxT("Caught an int %d in MyApp."), i);
    }
    catch ( MyException& e )
    {
        wxLogWarning(wxT("Caught MyException(%s) in MyApp."), e.what());
    }
    catch ( ... )
    {
        throw;
    }

    return true;
}
```

This finally frees the application developer from catching and handling exceptions in each of the application specific entry-points. Even though the GUI toolkit code is precompiled and cannot be changed, by overriding the behavior of the supplied function, an application developer can specify which exceptions his or her application should catch, and how they are to be handled. This solves the problem of code repetition, as now all of the application developer's exception handling code is together in one location.

I would argue, however, that this is not an ideal solution. The crux of this idea relies on the application developer making use of the catch-try-rethrow trick, but that piece of C++ behavior is virtually unknown. None of the C++ standards documents, Stroustrup's "The C++ Programming Language", nor ISO C++ conference papers makes explicit reference to this behavior at all. As described above, *indirect* evidence to support it can be found in the ISO standard, but it is

not obvious. Expecting every user of a GUI toolkit to be able to define the above function, even if provided with an example, is an unreasonable expectation.

In this paper, I argue that GUI toolkits should adopt a more involved solution, which I detail more thoroughly in the next section, but if the attitudes about exceptions do not change, and toolkits continue to have only mediocre support for exceptions, at least this paper would have made wxWidgets' solution better-known. Applications that require good exception support can be developed with *wxWidgets* and make use of `wxApp::OnExceptionInMainLoop()` to put all of their exception handling together in one central location.

3.3.2. Approach 2 - `register_exception_handler`

As described in the previous section, the main problem with the solution therein is it relies on every application developer to write non-obvious code. Moreover, every application developer who implements this behavior will write *exactly the same* block of non-obvious code. All developers' implementations of `deduceExceptionTypeAndHandle()` or `wxApp::OnExceptionInMainLoop()` will look the same: `try`, `throw;`, and then `catch` all of the exception types their application will catch and specify their handling. It is the job of a library to manage non-obvious, common code so that users of the library are free from the chore[6], and this case is no different.

The `deduceExceptionTypeAndHandle()` approach boils down to a very simple action: an application developer associating exception types with how they are handled. Through specifying catch-blocks in the overridden function, the applica-

---

[6] From Stroustrup: "For high-level applications programming to be effective and convenient, we need libraries. Using just the bare language features makes almost all programming quite painful." That is certainly true of `deduceExceptionTypeAndHandle()`.

tion is describing how to handle each exception type. Recall the example from the previous section,

```
void GUIApp::deduceExceptionTypeAndHandle() {
    try {
        throw;
    }
    catch(UserInputError &e) {
        notifyUserOfInputError(e); // display error message
    }
    catch(NetworkError &e) {
        attemptToReconnectToNetwork(e); // try again or exit
    }
    catch(std::exception &e) {
        gracefullyEndProgram(e); // likely a programming error so exit
    }
    // If we made it here, we don't know the type, so let it terminate.
}
```

The intent of the above code could be described in three simple lines:

```
On UserInputError: notifyUserOfInputError(e)
On NetworkError: attemptToReconnectToNetwork(e)
On std::exception: gracefullyEndProgram(e)
```

Each of these three lines describes what function should be called when an exception of a certain type is caught. If `UserInputError` is caught, call `notifyUserOfInputError()`; if `NetworkError`, call `attemptToReconnectToNetwork()`; and so on. These lines describe the *intended* behavior, and ideally the code used to express it would reflect that.

To achieve that, I propose an application framework method whose use explicitly mirrors the lines above that describe intent. The application framework should provide an additional function to register exception handlers, `register_exception_handler`. This function will be called by the application-specific code to establish how the framework will handle exceptions. We will use a C++ template to denote the exception type we want handled. We will also use a template to denote the type of the handler, allowing both functions and function

objects to be passed as the handler.[7] This is the declaration of `register_exception_handler`:

```
class GUIApp {
public:
    // ...

    template<typename Exception_t, typename Handler_t>
    void register_exception_handler(Handler_t handle) {
        // ...register handle to be a handler for Exception_t exceptions...
    }
};
```

The function is called in application-specific code to set up the application framework's exception handling. Its use matches the three succinct lines from above:

```
int main() {
  register_exception_handler<UserInputError> (&notifyUserOfInputError);
  register_exception_handler<NetworkError> (&attemptToReconnectToNetwork);
  register_exception_handler<std::exception> (&gracefullyEndProgram);
  // ...
}
```

In each of the above three lines, the application developer specifies an exception type that the application should catch (the type is specified as the template parameter within the <> angle brackets) and the handler that should be called when an exception is caught (the parameter in parentheses; this can be a function object, pointer to function or pointer to member). Then, on an exception, the application framework will deduce the exception's type and call the appropriate handler. We have already discussed how to dynamically deduce an exception's type; what remains to be shown is how it can be done with dynamically registered types and

[7] Templating the handler operation matches the style of the functions from the STL's `<algorithm>` header, in which predicates and operations are always supplied as templated types. Like these functions, Handler_t can be a unary function, a function object, or in C++11, a lambda function.

handlers. For that, we need to introduce some new types and functions to our GUI library.

3.4. Implementation of Dynamically Registered Exception Handlers

When an application registers an exception type to be handled by a certain handler, the application framework needs to store that *(exception-type, handler)* pair. Then, when an exception occurs, it needs to check the type of the thrown exception against the types of all the registered *(exception-type, handler)* pairs. For any *exception-type* matches, the corresponding *handler* should be called.

In order to represent these *(exception-type, handler)* pairs, we create a new type, `ExceptionHandler`. This class, like `register_exception_handler()`, will be templated on `Exception_t` in order to keep track of the *exception-type*. It will also store a copy of the *handler*, and so will be templated on `Handler_t` as well. The `Exception_t` template stores the *exception-type* from the *(exception-type, handler)* pair, and the `handler` member variable stores the *handler* from the pair.

```
template <typename Exception_t, typename Handler_t>
class ExceptionHandler {
public:
    ExceptionHandler(const Handler_t &h) : handler(h) { }

private:

    // a copy of the handler to be called if Exception_t is caught
    Handler_t handler;
};
```

`ExceptionHandler` is templated on the `Exception_t`, so any member functions we add to it next will be able to reference the *exception-type* of the pair. The copy of the *handler* from the pair can be called when an exception of type *exception-type* is caught.

Next, we make use of the catch-try-rethrow trick from section §3.2 by adding a member function that will deduce the type and handle any exception whose type matches `ExceptionHandler`'s *exception-type*. This function, `ExceptionHandler::try_rethrow_catch()`, when called from within a `catch()` block, will attempt to determine the active exception's type, and will handle the exception if its type matches the type on which the particular instance of `ExceptionHandler` is templated. This function looks almost exactly like the overridden `deduceExceptionTypeAndHandle()` function defined in previous sections: it uses the catch-try-rethrow trick to check if the active exception is of a certain type. The difference being that the exception type being checked is a templated type which was defined when the instance object was instantiated.

```
template <typename Exception_t, typename Handler_t>
class ExceptionHandler {
public:
    ExceptionHandler(const Handler_t &h) : handler(h) { }

    // Requires: Must be called from within a catch-block.
    // Effects: If the thrown exception was of type Exception_t,
    //            then call the handler on the caught exception.
    // Returns: true if the exception was handled, else false.
    bool try_rethrow_catch() {
        try {
            throw;
        }
        catch(const Exception_t &e) { // If active exception is Exception_t
            handler(e);               //   handle it.
            return true;
        }
        catch(...) { // don't let the exception escape if wrong type.
            return false;
        }
    }

private:
    // a copy of the handler to be called if Exception_t is caught
    Handler_t handler;
};
```

It should be clear that an ExceptionHandler<A, A_Handler>::try_rethrow_catch() will handle the active exception if and only if the exception is of type A. This matches the desired behavior exactly. Before we look at an example of its use, let us also define a helper function for creating ExceptionHandlers. This way only the *exception-type* needs to be typed explicitly when instantiating ExceptionHandlers.

```
// A helper function to create ExceptionHandlers. This is useful because
//  it can use the handler parameter to deduce Handler_t.
template <typename Exception_t, typename Handler_t>
ExceptionHandler<Exception_t, Handler_t>
                      create_exception_handler(const Handler_t &handler) {
    return ExceptionHandler<Exception_t, Handler_t>(handler);
}
```

As an example of ExceptionHandler's use, let eh_NE be an instance of ExceptionHandler that is templated on NetworkError from MyNetworkApplication. eh_NE.try_rethrow_catch() would only call attemptToReconnectToNetwork() if there was an active exception of type NetworkError. In the following example, just such an ExceptionHandler is constructed, and it only calls attemptToReconnectToNetwork when a NetworkError is thrown:

```
// Handle a NetworkError exception by attempting to reconnect to network.
void attemptToReconnectToNetwork(const NetworkError &e) {
    cout << "Network Error: " << e.error_num << ". ";
    cout <<  "Attempting to reconnect..." << endl;
    // ...
}

int main() {
    // Create an exception handler that handles NetworkError exceptions
    //  by calling attemptToReconnectToNetwork..

    auto eh_NE = create_exception_handler<NetworkError>(
                                            &attemptToReconnectToNetwork);

    try {
        NetworkError ne(NETWORK_DISCONNECT_ERROR); // (error 10)
        throw ne;             // Throw a Network Disconnect Error.
    }
```

```
    catch(...) {
        // This calls attemptToReconnectToNetwork(ne).
        bool result = eh_NE.try_rethrow_catch();
        cout << "try-rethrow-catch 1: " << result << endl;
    }

    try {
        std::runtime_error re("test error");
        throw re;
    }
    catch(...) {
        // This will do nothing.
        // (Because eh_NE::Exception_t doesn't match std::runtime_error)
        bool result = eh_NE.try_rethrow_catch();
        cout << "try-rethrow-catch 2: " << result << endl;
    }
}
```

This will result in the following output:

```
Network Error: 10. Attempting to reconnect...
try-rethrow-catch 1: 1
try-rethrow-catch 2: 0
```

As the output describes, the first call to `eh_NE.try_rethrow_catch()` does handle the thrown exception, because `eh_NE` was templated on `NetworkError`, and it does not handle the second exception. This turns out to be an extremely useful class. A single call to this class's `try_rethrow_catch()` can deduce the type of and handle an active exception.

To handle a thrown exception, the GUI toolkit will maintain a list these `ExceptionHandlers` to keep track of the registered *(exception-type, handler)* pairs. On an exception in the run-loop, it will iterate through this list, calling each `ExceptionHandler`'s `try_rethrow_catch()`. Any `ExceptionHandlers` templated to the correct type will handle the exception.

With this in mind, we can finally implement the desired behavior of the `register_exception_handler` function described earlier. This function will create an `ExceptionHandler` that is templated according to the desired *(exception-type, han-*

dler) pair, and then add it to a list of ExceptionHandlers. Ideally, this would look like the following function:

```
template <typename Exception_t, typename Handler_t>
void GUIApp::register_exception_handler(const Handler_t &handler) {

    // This is the right idea, but this won't compile...
    eh_pairs.push_back(create_exception_handler<Exception_t>(handler));
}
```

Of course, C++ doesn't allow heterogenous container elements, so eh_pairs is impossible (two instances of a templated class, each with two different template specifications, do indeed have two distinct types). The solution is to make eh_pairs a list of pointers to a base class from which our ExceptionHandlers will be derived. This is a common solution for a list of templated types. We rename ExceptionHandler to ExceptionHandler_Impl, and define a new base class, ExceptionHandler, which is simply an abstract interface.

```
// Abstract Base Class
class ExceptionHandler {
public:
    virtual void try_rethrow_catch() = 0; // pure virtual function
};
```

```
// Implementation class
template <typename Exception_t, typename Handler_t>
class ExceptionHandler_Impl : public ExceptionHandler {
    // ... (this is the same as ExceptionHandler was previously) ...
};
```

```
// A helper function to create new ExceptionHandlers.
template <typename Exception_t, typename Handler_t>
ExceptionHandler<Exception_t, Handler_t>*
                        create_exception_handler(const Handler_t &handler) {
    return new ExceptionHandler_Impl<Exception_t, Handler_t>(handler);
}
```

The contents of ExceptionHandler have been moved to ExceptionHandler_Impl, and ExceptionHandler is now empty. Most importantly, ExceptionHandler

is *not templated*, so that it can be used as the pointer-type for a homogenous list. `create_exception_handler` now returns a pointer-to-base for a dynamically allocated `ExceptionHandler_Impl` in order to make use of the polymorphic behavior.

Now, `eh_pairs` holds base class pointers to `ExceptionHandlers`, which actually point to differently templated `ExceptionHandler_Impls`. This finally allows `GUIApp` to both hold a list representing all the *(exception-type, handler)* pairs that have been registered, as well as to be able to iterate through that list and handle any exceptions. To do so, it simply calls the virtual `try_rethrow_catch()` function for each templated `ExceptionHandler_Impl`. The redefined `GUIApp` will look like this:

```
class GUIApp {
    // ...
    std::list<ExceptionHandler*> eh_pairs;

public:
    template <typename Exception_t, typename Handler_t>
    void register_exception_handler(const Handler_t &handler) {

      eh_pairs.push_back(create_exception_handler<Exception_t>(handler));
    }

    // Begin the application. run() never returns.
    void run();

    // ...
};
```

The GUI toolkit will use `eh_pairs` to determine the type of any caught exceptions. Recall the GUI toolkit run-loop, which includes a `try` block and a `catch(...)` block. Inside this `catch(...)` block, the toolkit will loop through each of the `ExceptionHandlers` in `eh_pairs`, and call `try_rethrow_catch()`. If any of the `ExceptionHandler_Impls` were templated on the active exception's type, the registered handler will be called. `GUIApp::run()` now looks like this:

```
void GUIApp::run() {
   while (true) {
      try {
         // ... may throw something ...
      }
      catch(...) {
         bool handled = false; // do any of the eh_pairs match it?

         for(ExceptionHandler *eh : eh_pairs) {
            handled = handled || eh->try_rethrow_catch();
         }

         if (!handled) { // if the exception is never succesfully caught,
            throw;       //  that means there aren't any registered handlers,
                         //  so terminate the program.
         }
      }
   }
}
```

In the above for-loop, every registered *(exception-type, handler)* pair is tested for a matching *exception-type*. If found, the correct *handler* is called. If no handlers match, the program is terminated. These are the three requirements outlined at the beginning of section §3: 1. deduce the type of the exception; 2. handle the exception; 3. terminate if no handlers are registered.

Thus, the application framework has solved the problem of allowing an application to accumulate all of its exception-handling code in one place. The application only needs to make calls to `register_exception_handler` and afterwards any exceptions thrown from application-specific code will be caught and handled appropriately. The application no longer needs to wrap each of its entry-points with repeated exception-handling code.

In this last example, we revisit *MyNetworkApplication* using the `register_exception_handler` structure we've defined. Before the application begins its execution, it makes three simple calls to `register_exception_handler`, each one specifying the handling for a different exception type.

```
// Define exception handler functions
void notifyUserofInputError(const UserInputError &e)      { /* ... */ }
void attemptToReconnectToNetwork(const NetworkError &e)   { /* ... */ }
void gracefullyEndProgram(const std::exception &e)        { /* ... */ }

int main() {

  GUIApp app;

  app.register_exception_handler<UserInputError>(&notifyUserOfInputError);
  app.register_exception_handler<NetworkError>(&attemptToReconnectToNetwork);
  app.register_exception_handler<std::exception>(&gracefullyEndProgram);

  // ... any further GUIApp setup ...

  app.run(); // start the application.
}
```

Now, if any of the above exception types (i.e. `UserInputError`, `NetworkError`, or `std::exception`) are thrown during `app.run()`'s run-loop, they will be handled in the respective handler functions (`notifyUserOfInputError`, `attemptToReconnectToNetwork`, and `gracefullyEndProgram`).

This concludes the presentation of the `register_exception_handler` solution. For a complete implementation that can be directly used in a real application framework, see Appendix 1. For an example of a complete GUI toolkit's run-loop utilizing this solution, see Appendix 2.

3.5. Comparison of Methods

This completes the presentation of the solution advocated for in this paper. This section will compare the relative merits of this solution with the other methods presented above.

The solution described in section §3.3.1, the solution provided by wxWidgets, achieves the goals set out in the Introduction. However, it requires the appli-

cation developer to implement the solution completely, without support from the application framework. Every application developer who is able to produce the correct solution will produce almost identical exception handling code. Even worse, in order to produce the correct solution, every developer will have to understand the example code provided by wxWidgets and understand how to modify it for their application. In this regard, `register_exception_handler` is superior, as it frees the application developer both from understanding and from implementing the solution.

The FLTK solution presented in section §2.2 has the same problem: it must be fully implemented by the user. It requires the application developer to copy code directly from the documentation and adapt it to his or her application. The example code was somewhat buried in the documentation, and took almost an hour to find. The solution presented in this paper, on the other hand, allows an application to dynamically register exception handlers with only one simple line, freeing every the application developer from finding, understanding, and implementing the exact same solution. But, like the wxWidgets solution above, FLTK's solution is much simpler to implement than the `register_exception_handler` solution.

On the other hand, most likely the `register_exception_handler` solution has slightly worse performance than the other two solutions. The performance hit will only be when an exception is caught, and probably won't be significant, but it will probably be worse than the other two solutions. For a more complete discussion of `register_exception_handler`'s efficiency, see Section §5.1 in Further Work.

4. Changes in the C++11 and C++14[8] Standards

The recently released ISO C++ 2011 standard [4] introduced many changes with regards to C++ exceptions, and mainstream compilers are finally completing their C++11 support.[9] Despite a number of additions to the C++ standard library that address exception handling, it does not appear that the exception-related changes in C++11 have a major impact on solving the problem addressed by this paper. Development is currently underway on the next standard, provisionally being called the C++14 standard, but so far there do not seem to be any changes introduced regarding C++ exceptions [5]. There are, however some changes in the C++11 standard that are not explicitly related to exceptions, but might be beneficial in this solution, notably C++ variadic templates. This is described in more detail in section §5.2 of Further Work.

4.1. C++11 Exception Changes

The changes to C++ exceptions introduced by the C++11 standard were mainly focused on throwing and catching exceptions in a mutli-threaded application [4 , 33]. C++11 introduces the `exception_ptr` class, as well as the functions `current_exception()`, `rethrow_exception()`, and `make_exception_ptr()`. All four of these additions address the needs of transferring an exception between threads: "In particular, an `exception_ptr` can be used to implement a re-throw of an exception in a different thread from the one in which the exception was caught" [32 (p.871)]. These tools do not aid us in dynamically deducing an exception's type or in allowing a user to dynamically specify how to handle an exception, the two

[8] Also known provisionally as C++1y

[9] Both GCC and LLVM have implemented all of the "major features" in the C++11 standard [2 , 6 , 12].

actions necessary for a GUI toolkit to handle application-specific exceptions in a central location; therefore, they do not affect possible solutions to the problem.

The other exception-related concept introduced by C++11 is `nested_exception`, which can be used to tack-on information to an exception as it moves up the call tree. Although it may appear to be at first, this feature is also not useful for our problem. From Stroustrup [32 (p.872-873)]:

> The intended use of `nested_exception` is as a base class for a class used by an exception handler to pass some information about the local context of an error together with a [sic.] `exception_ptr` to the exception that caused it to be called. [...] Further up the chain, we might want to look at the nested exception.

The `nested_exception` concept allows a try-block further up the list of invokers to catch an exception with a known type, even though the actual type of the originally thrown exception may be *unknown* at that point. The further-up try-block can then rethrow the `nested_exception`'s nested exception to access the original exception object.

To achieve this, the standard introduces the following functions along with the `nested_exception` class: `ne.rethrow_exception()`, `ne.nested_ptr()`, `throw_with_nested(e)`, and `rethrow_if_nested(e)`. Stroustrup provides this example in his book:

```
struct My_error : runtime_error {
    My_error(const string&);
    // ...
};

void my_code()
{
    try {
        // ...
    }
    catch(...) {
        My_error err {"something went wrong in my_code()"};
        // ...
        throw_with_nested(err);
    }
}
```

Now My_error information is passed along (rethrown) together with a nested_exception holding an exception_ptr to the exception caught.

Further up the call chain, we might want to look at the nested exception:

```
void user()
{
    try {
        my_code();
    }
    catch(My_error &err) {

    // ... clear up My_error problems ...

        try { // NOTE: See † below
            rethrow_if_nested(err); // re-throw the nested exception, if any
        }
        catch(Some_error& err2) {
            // ... clear up Some_error problems ...
        }
    }
}
```

This assumes that we know that **some_error** might be nested with **My_error**.

† At first this may look like an application of using the *catch-try-rethrow* trick to deduce a caught exception's type, but closer inspection reveals that it is *not*. `rethrow_if_nested` throws a *new exception* -- the exception pointed to by its nested `exception_ptr`. So this is *not* using the *catch-try-rethrow* trick, and is not deducing the type of the exception; it is simply throwing a new exception, one that happened to have been thrown earlier and is now stored in an `std::exception`.

In this example, **my_code** is able to add additional information to the the original **Some_error** that was thrown (in this case, the information added is that the exception was thrown from within a call to **my_code**). Of course, **user()** could simply catch **Some_error** directly, but without tacking on the **My_error** class, the catch statement in **user()** would be unable to tell *which part of its try-block* the exception was thrown from.

Unfortunately, however, this doesn't make our job any easier. In order to determine the type of the original exception, we *still need to rethrow the nested exception*, and we *still need to know what types to catch*. In other words, the

catch-try-rethrow trick is still necessary and the templated classes to represent *(exception-type, handler)* are still necessary, so we have gained nothing from this method.

The new C++11 exception functionality is interesting, but not relevant to this problem. The currently proposed C++14 standard does not make any changes to exception handling. Therefore, as far as we have found, the results of this paper are still current and relevant after the recent changes to the C++ language. For more information on C++11 variadic templates, see Section §5.2 of Further Work below.

5. Further Work

5.1. Benchmarking

Due to time constraints, there was no formal study performed on the efficiency of the solutions described herein. Presumably the `ExceptionHandlers` solution presented in Section §3.3.2 will be less runtime-efficient than either of the wxWidgets-style solution or the FLTK solution, but *only* when an exception is thrown. Otherwise there should be no impact on performance. This is probably an acceptable performance drop for the benefits achieved because the efficiency concerns are only present during exception handling, which is a rare event.[10] Still, it would be beneficial to formally measure the performance differences between all of the methods presented here.

---

[10] According to Lippman [27 (p.170)], the Modula-3 Report recommends "that 10,000 instructions may be spent in the exceptional case to save one instruction in the normal case."

5.2. C++11 Variadic Templates

While C++11 did not provide any new exception handling mechanisms relevant to this paper (see Section §4 above), it did introduce the concept of variadic templates [32 (p.809)]. It remains to be seen, but it is possible that variadic templates would be beneficial for the solutions discussed in this paper. Using variadic templates, an application may be able to build a try-catch clause similar to the one described in Section §3.2 entirely using metaprogramming techniques [32 (p.780)].

5.3. TnFox GUI Toolkit

Unfortunately, a lack of time prevented further research into the TnFox GUI toolkit, which claims to be an "exception aware" GUI toolkit. The homepage explains that "every single error is detected and reported with additional support for handling nested C++ exception throws as well as C++ rollback transactions which allow interruptions to gracefully unwind partially completed operations." Clearly, this is a toolkit that has emphasized good exception handling in its design, and as such it certainly merits additional study.

6. Conclusions

It is the responsibility of a good library to free its users from the chore of writing complex, convoluted, obscure, or repetitive code. Yet all of the common GUI toolkits in use today require their users to do just that when it comes to writing exception handling code. In the worst case, the toolkits provide no support at all, and the application developer is forced to write repetitive *try-catch* exception

handlers, duplicating them across each of his or her application-specific entry points. In the toolkits with slightly better support, the exception-handling doesn't need to be repeated -- a great achievement, but becomes instead complex, convoluted, and obscure. The user must either imitate the documentation exactly and carefully copy a convoluted, fragile solution (as in the case of the *FLTK* solution), or independently invent or discover the complex, obscure solution that utilizes catch-try-rethrow all on their own (as with the *wxWidgets* solution). With this situation as the status quo, there is clearly room for improvement. Application developers using a GUI toolkit should not need to reinvent the wheel; they should instead have this complex task solved and prepackaged for them.

And we managed to do just that. To solve this problem, we started by designing our desired behavior. The goal was to allow an application to easily specify its exception-handling requirements in one central location. To do this, we described the one-line function, `register_exception_handler<Exception_t>(handler)`. In this function, the user would dynamically register an exception type to be caught and then handled in a certain, or possibly multiple way(s). With this one line, an application developer can specify all the information necessary to correctly handle exceptional conditions that arise during calls to application-specific code. As we saw above, we managed to implement this behavior utilizing a templated class to represent an *(exception-type, handler)* pair.

This implementation made use of a little-known C++ trick, which throughout this paper I have been referring to as the *catch-try-rethrow* trick. We have shown with reasonable certainty that the catch-try-rethrow trick is standard C++ behavior. A function within an active catch-block can enter into a new try-block, rethrow the active exception, and re-catch the exception with a new catch-block. In addition, it has been experimentally verified in some major compilers: g++ 4.2 and 4.7, Visual Studio 2012, and Clang 3.2. We showed that this trick can be used

within a `catch(...)` block to dynamically determine an exception's type. We showed that an application framework could use this trick to provide a central location for application-specific exception handling. (As an aside, we used this knowledge to implement a solution using the wxWidgets GUI toolkit.) Finally, we showed how to use templated classes to keep a list of *(exception-type, handler)* pairs. With all of this information together: the list of *(exception-type, handler)* pairs, the ability to dynamically check an exception's type against an *exception-type* from a pair, and the ability to dynamically add to and remove pairs from the list, we are able to implement the one-line `register_exception_handler<Exception_t>(handler)` function.

While the presence of an application framework certainly makes exception handling more complicated, it does not inherently preclude good program design. The problem of copy-pasted exception-handling code, though difficult, *is* solvable, and I have presented a complete solution in this paper, as well as descriptions of the best solutions available to application designers today. Ideally C++ GUI toolkits will improve their exception support, possibly utilizing the `register_exception_handler` solution presented here, but if not, an application developer can implement his or her own C++ exception handling solutions through wxWidgets or FLTK.

7. Discussions

I was surprised to find that this problem had not already been satisfactorily solved. In fact, much of what I discovered in this thesis project surprised me: I was surprised most C++ GUI toolkits have relatively poor support for exceptions; I was surprised it was so difficult to uncover the documentation about their sup-

port; and I was surprised that the *catch-try-rethrow* trick appears to be relatively undocumented, and even relatively unknown.

I cannot say with certainty exactly why C++ exception support hasn't been more fleshed out in the prominent GUI toolkits, but it would seem that timing is largely to blame. The main C++ GUI toolkits are all very old; most of them were first introduced before exceptions had been added to the C++ language, or only just after they were first implemented. The first implementations of C++ exceptions made executables both significantly larger and noticeably slower, and many developers formed a distrust of the exceptions mechanism entirely [13 (p.35)]. Likely, it is a fear of executable bloat and runtime performance costs that has dissuaded many toolkit developers from spending the time and manpower on researching better methods for handle exceptions, or has even prevented them from supporting exceptions at all. For example, wxWidgets is the GUI toolkit whose solution was presented in section 3.3.1 as an example of a good solution to exception handling. However, this solution is new as of wxWidgets v3.0, and until 2011 their official stance was that exceptions should be avoided entirely. The toolkit's programming style guide [37] explicitly advised against using exceptions at all:

> **Don't use C++ exceptions**
>
> The C++ exception system is an error-reporting mechanism. Another reasons(sic) not to use it, besides portability, are the performance penalty it imposes (small, but, at least for current compilers, non-zero), and subtle problems with memory/resource deallocation it may create (the place where you'd like to use C++ exceptions most of all are the constructors, but you need to be very careful in order to be able to do it).

Today, however, almost all implementations of C++ exceptions have no runtime overhead whatsoever, and the executable bloat is considerably smaller [13 (p.40)], so there is no reason that they should continue to be avoided.

Another possible reason it has been avoided is that the GUI toolkits use a C middle-layer somewhere in their implementation. This layer could be a different

library used to implement the toolkit, or it could possibly be an operating system's C API. Some GUI toolkits do in fact use a C middle-layer, though not all of them do, and perhaps these toolkits have influenced the development of the others as well. A C middle-layer makes it more difficult to handle C++ exceptions, because the exception will not be propagated safely through a C function.

The second major surprise to me was how difficult it was to obtain information on exception support in many of the GUI toolkits. This paper provides a convenient summary of the exception handling techniques implemented by the major C++ GUI toolkits.

The final surprise was just how undocumented the *catch-try-rethrow* trick turned out to be. Other than showing up in a wxWidgets sample application, the only sources I have managed to find that explicitly describe the trick are three answers to three questions from *StackOverflow.com*, and the two of them that claim evidence of its validity cite the same sections of the standard that I cited here to justify my claims. In one question [23], the user asks for a way to obtain an "exception object address [...] from inside of [a] catch(...) block," and is answered with two answers stating, effectively, "no." In the third (and accepted) answer, *Simon Richter*[11] answers correctly, though, and describes the *catch-try-rethrow* trick almost exactly as I have described it [25]:

> If you know at least something about the type, then yes.
>
> The `catch(...)` syntax does not give a name to the exception object, but it is possible to rethrow the object and use a more specific catch clause:
>
> ```
> try {
>    throw 0;
> }
> catch(...)
> {
> ```

[11] http://stackoverflow.com/users/613064/simon-richter

```
        try {
            throw;
        }
        catch(int &i)
        {
            std::cout << &i << std::endl;
        }
    }
```

Clearly, *Simon Richter* is aware that this is valid C++. He is confident enough to present it as an answer to this question, but I still have not seen it *explicitly* documented anywhere. Another answer goes even further: in his question [21], *John*[12] asks almost exactly the same thing as above, and, as above, receives two answers claiming it is impossible. But, *Fred Nurk*[13]'s answer [24] not only shows how to use the *catch-try-rethrow* trick, it goes on to propose a handler-chain that is remarkably similar to the one proposed in this paper:

> Because C++ is statically typed, you must catch a known type. However, you can call an external function (or set of functions) which handle exception types unknown at the point you call them. If these handlers all have known types, you can register them to be dynamically tried.

He then describes a handler chain that is one step away from what I have described. His chain is a list of functions that return std::string, which can be used to extract an error description from any exception type. Each function enters a new *try-block*, rethrows the current exception, and then attempts to catch it with a known type (and finally extracts the string from it, which is returned). This is one of his example functions:

```
std::string extract_from_unknown_external_exception() {
  try { throw; }
  catch (myStupidCustomString &e) {
    return e.what;
  }
```

---

[12] http://stackoverflow.com/users/197229/john

[13] http://stackoverflow.com/users/511601/fred-nurk

```
    catch (...) {
      throw;  // Rethrow original exception.
    }
  }
```

The user would write one of these for each type that could be caught, and would add them to a list. While this method still requires a lot of duplication of somewhat convoluted code, it is still a hair's breadth away from the solution described in this paper.

Finally, in his answer to a question [22] asking explicitly about the legality of the *catch-try-rethrow* trick, *GManNickG*[14] points out a section in the C++03 standard that seems to identify it as standard behavior [20]. This section is the same section I've identified, section §15.1/4:

> The memory for the temporary copy of the exception being thrown is allocated in an unspecified way, except as noted in 3.7.4.1. The temporary persists as long as there is a handler being executed for that exception.

A wxWidgets example application, these few *StackOverflow.com* answers, and the scraps of the standard they cite, are the only sources I've found that describe this functionality, but it seems to be supported by the C++ standard as legal behavior. Hopefully this paper has convinced the reader of the usefulness of the try-rethrow-catch trick, and will help to make this seemingly under-documented trick more commonly known.

[14] http://stackoverflow.com/users/87234/gmannickg

8. Appendices

8.1. Appendix 1 — The GUIExceptionHandling Module

The `GUIExceptionHandling` module contains an implementation of the catch-try-rethrow trick. It is intended to be a standalone, headers-only module that could be included in any application framework. It is comprised of two files: `GUIExceptionHandling.h`, which contains the interface for using `ExceptionHandlers`, and `GUIExceptionHandling_Impl.h`, which contains the template class implementation.

GUIExceptionHandling.h

```
//
//  ExceptionHandling.h
//
//  Definitions for the Exception Handling functions:
//    ExceptionHandler* create_exception_handler(const Handler_t &handler);
//    void call_exception_handlers(InputIterator begin, InputIterator end);
//

#ifndef GUI_Exception_Handling_h
#define GUI_Exception_Handling_h

#include "GUIExceptionHandling_Impl.h" // Contains details irrelevant to clients

namespace GUIExceptionHandling {

// Exception_t : The type of the exceptions that will be caught by handler
// Handler_t : a function or object that overrides operator()(Exception_t);
// NOTE: handler will be copied.
template <typename Exception_t, typename Handler_t>
ExceptionHandler* create_exception_handler(const Handler_t &handler) {
    return new ExceptionHandler_Impl<Exception_t, Handler_t>(handler);
}

// Loop through error handlers and handle any errors. If no handler matches
//  the error, it will be rethrown out of the function.
// REQUIRES: This function MUST be called from within a catch(){} block!
template <typename InputIterator>
void call_exception_handlers(InputIterator begin, InputIterator end) {

    bool handled = false;

    try {
        // Create a vector of handlers
        // (Since try_rethrow_catch is virtual, it cannot be templated,
        //                                              so it requires a vector.)
        std::vector<ExceptionHandler*> handlers(begin, end);
```

```
        // loop through all the handlers and handle if possible.
        // This will end with a rethrow.
        handlers.front()->try_rethrow_catch(++handlers.begin(),
                                            handlers.end(), handled);
    }
    catch (...) { // Will always enter this catch statement.
        if (!handled) { // Only rethrow if none of the Handlers caught the
error.
            throw;
        }
    }
}

} // namespace GUIExceptionHandling

#endif /* GUI_Exception_Handling_h */
```

GUIExceptionHandling_Impl.h

```
//
//  ExceptionHandling_Impl.h
//
//  Implementation for the ExceptionHandler class.
//

#ifndef GUI_Exception_Handling_Impl_h
#define GUI_Exception_Handling_Impl_h

#include <vector>

namespace GUIExceptionHandling {

// Forward Declaration for friending
template <typename Exception_t, typename Handler_t>
class ExceptionHandler_Impl;


// Abstract ExceptionHandler Base Class.
// Derived class will be templated for Exception type and Handler type.
class ExceptionHandler {
private:    // Everything is private so that this can only be used as a base
            //  class for the ExceptionHandler_Impl class.


    // A virtual function cannot be templated, so we require that try_catch,
    //  below, use a vector::iterator.
    typedef std::vector<ExceptionHandler*>::iterator ExceptionHandlerIter_t;

    // Recursive function call to iterate through list and try-rethrow-catch on
    //  currently thrown exception.
    // REQUIRES: this must be called from inside a catch{} block.
```

```
    virtual void try_rethrow_catch(ExceptionHandlerIter_t begin,
                                   ExceptionHandlerIter_t end, bool &handled)=0;


    // This is the public interface for interacting with ExceptionHandlers.
    template <typename InputIterator>
    friend void call_exception_handlers(InputIterator begin,
                                        InputIterator end);

    // The only actual derived class that will use this class's functions.
    template <typename Exception_t, typename Handler_t>
    friend class ExceptionHandler_Impl;
};


// Implementation of ExceptionHandler. Nests try statements for all
// ExceptionHandlers passed in, and rethrows the current exception.
// Then each ExceptionHandler attempts to catch during the unravelling.
//
// Exception_t : The type of errors that will be caught by handler
// Handler_t : a function or object that overrides operator()(Exception_t);
template <typename Exception_t, typename Handler_t>
class ExceptionHandler_Impl : public ExceptionHandler {
private:    // Everything is private, so that ExceptionHandler_Impl isn't
            //  created anywhere but from create_exception_handler().


    // handler_ should be a callable entity s.t. handler_(Exception_t) is valid.
    // NOTE: handler_ will be copied.
    ExceptionHandler_Impl(const Handler_t &handler_) : handler(handler_) { }

    // For each ExceptionHandler in [begin:end), try to handle the error.
    // RESULT: handled will be set to true if any ExceptionHandler successfully
    // handled the current exception.
    // REQUIRES: this must be called from inside a catch{} block.
    virtual void try_rethrow_catch(ExceptionHandlerIter_t begin,
                                   ExceptionHandlerIter_t end, bool &handled) {

        // nest a try block for each ExceptionHandler
        try {
            if (begin == end) { // base case
                throw;
            }
            ExceptionHandler *next = *begin;
            next->try_rethrow_catch(++begin, end, handled); // unravel until end
        }
        // Each ExceptionHandler gets a chance to try to catch the exception.
        catch(const Exception_t &e) { // Will only catch if e is of Exception_t

            handler(e);     // handler() is only called if Exception_t matches.
            handled = true;
            throw;          // continue up the chain.
```

```
        }
    }

    // Friend this because of the private constructor
    template <typename Exc_t, typename Han_t>
    friend ExceptionHandler* create_exception_handler(const Han_t &handler);

private: // Private member data:
    Handler_t handler;
};

} // namespace GUIExceptionHandling

#endif /* GUI_Exception_Handling_Impl_h */
```

8.2. Appendix 2 — The GUIApp Example Module

The `GUIApp` module is an example of how an application framework would utilize the `GUIExceptionHandling` module. This module defines the framework's main application class, which would be instantiated once in an application. The `App` class provides functions that allow the application to register and unregister exception handlers. The class maintains a vector of `ExceptionHandler`s which it provides to the `GUIExceptionHandling` module's `call_exception_handlers` function.

GUIApp.h

```
//
//  GUIApp.h
//  GUI Widget Library
//
//  Created by Nathan Daly on 11/27/12.
//

#ifndef GUIApp_h
#define GUIApp_h

#include "GUIExceptionHandling.h"

#include <vector>


namespace GUI {
    using namespace GUIExceptionHandling;

class Window;


class App {
```

```
public:

    void run(Window* window);

    // When any code executed within the run() loop throws an instance of
    // Exception_t, any Handler_t's that have been registered for that
    // Exception_t will be called, with the exception passed as the argument.
    // RETURNS: an ID for this exception-handler pair. Use ID to un-register the
    //    exception-handler pair.
    template <typename Exception_t, typename Handler_t>
    int register_exception_handler(const Handler_t &handler);

    // Removes the exception-handler pair from the app.
    void unregister_exception_handler(int exception_handler_id);

private:

    // vector because it's fast to iterate through!
    typedef std::vector<ExceptionHandler*> error_handler_list_t;
    error_handler_list_t handler_list;

};

template <typename Exception_t, typename Handler_t>
int App::register_exception_handler(const Handler_t &handler) {
    handler_list.push_back(create_exception_handler<Exception_t>(handler));
    return (int)handler_list.size();
}


} // namespace GUI


#endif
```

GUIApp.cpp

```
//
//  GUIApp.cpp
//  GUI Widget Library
//
//  Created by Nathan Daly on 11/27/12.
//

#include "GUIApp.h"

#include "SDL/SDL.h" // For event polling
#include <iostream>
using std::vector;

using namespace GUIExceptionHandling;
```

```
namespace GUI {


void App::run(Window* window_) {


    bool running = true;


    while(running) {
        SDL_Event event;

        try {

            SDL_PollEvent(&event);


            switch (event.type) {

                case SDL_MOUSEBUTTONDOWN: {

                    // Send mouse event to correct view.
                    break;
                }
                case SDL_KEYDOWN: {
                    // handle keydown.
                    break;
                }
                default:
                    break;

            }
        }
        catch(...) {

            call_exception_handlers(handler_list.begin(), handler_list.end());

        }
    }
}

} // namespace GUI
```

Bibliography

[1] About the wxWidgets Project, Retrieved, from wxWidgets: http://wxwidgets.org/

[2] C++0xCompilerSupport, Retrieved, from: https://wiki.apache.org/stdcxx/C%2B%2B0xCompilerSupport?action=recall&rev=119

[3] INCITS/ISO/IEC 14882-2003. Programming Languages— C++, Second Edition, 2003.

[4] INCITS/ISO/IEC 3242-2011. Programming Languages— C++, Working Draft, 2011.

[5] INCITS/ISO/IEC 3797-2014. Programming Languages— C++, Working Draft, 2013.

[6] C++98, C++11, and C++14 Support in Clang, Retrieved, from: http://clang.llvm.org/cxx_status.html

[7] CWinApp::ProcessWndProcException 2013), Retrieved, from Microsoft: http://msdn.microsoft.com/en-us/library/93k07whb%28v=vs.100%29.aspx

[8] Documentation. JUCE 2013), Retrieved, from Raw Material Software Ltd: http://fox-toolkit.org/

[9] Documentation Overview. gtkmm C++ Interfaces for GTK+ and GNOME, Retrieved, from: http://www.gtkmm.org/en/documentation.html

[10] Exception Handling in MFC 2013), Retrieved, from Microsoft: http://msdn.microsoft.com/en-us/library/t078xe4f%28v=vs.110%29.aspx#_core_mfc_exception_support

[11] Exception Safety. Qt Project 2013), Retrieved, from Qt Project Hosting: http://qt-project.org/doc/qt-5.0/qtdoc/exceptionsafety.html

[12] Status of Experimental C++11 Support in GCC 4.8, Retrieved, from: http://gcc.gnu.org/gcc-4.8/cxx0x_status.html

[13] Working Group WG 21 of Subcommittee SC 22. Technical Report on C++ Performance. ISO/IEC TR 18015:2006(E)

[14] The Visual Component Framework (October 31 2013), Retrieved, from: http://vcf-online.org/

[15] What is GLUI? GLUI User Interface Library, Retrieved, from: http://glui.sourceforge.net/

[16] dcbasso [Resolved] Compilation problem: “Exception Handling disabled, use -fexceptions to enable”. Programming Style Guide (November 9 2012), Retrieved, from wxwidgets.org: https://qt-project.org/forums/viewthread/21771/

[17] Douglas, N. Welcome to the TnFOX homepage (March 18 2013), Retrieved, from: http://www.nedprod.com/TnFOX/

[18] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns*. Addison-Wesley Reading, 1995.

[19] Goodenough, J. B. Exception Handling: Issues and a Proposed Notion. *Communications of the ACM*, 18, 12 (December 1975).

[20] http://stackoverflow.com/users/87234/gmannickg Answer to "Is re-throwing an exception legal in a nested 'try'?" (Mar 19 2010), Retrieved, from StackOverflow.com: http://stackoverflow.com/revisions/2466156/4

[21] http://stackoverflow.com/users/197229/john C++ - finding the type of a caught default exception (Mar 1 2011), Retrieved, from StackOverflow.com: http://stackoverflow.com/revisions/4885334/1

[22] http://stackoverflow.com/users/234053/alexander-gessler Is re-throwing an exception legal in a nested 'try'? (Mar 17 2010), Retrieved, from StackOverflow.com: http://stackoverflow.com/revisions/2466131/1

[23] http://stackoverflow.com/users/369639/romeno Getting exception object address from catch (…) c++ (Mar 1 2011), Retrieved, from StackOverflow.com: http://stackoverflow.com/revisions/5157169/2

[24] http://stackoverflow.com/users/511601/fred-nurk Answer to "C++ - finding the type of a caught default exception" (Feb 3 2011), Retrieved, from StackOverflow.com: http://stackoverflow.com/revisions/4885632/5

[25] http://stackoverflow.com/users/613064/simon-richter Answer to "Getting exception object address from catch (…) c++" (Feb 3 2011), Retrieved, from StackOverflow.com: http://stackoverflow.com/revisions/5157315/1

[26] koldo Ultimate++ is a C++ cross-platform rapid application development framework (October 31 2013), Retrieved, from: http://ultimatepp.org/

[27] Lippman, S. B. *Inside The C++ Object Model*. Addison Wesley, 1996.

[28] Maddock, J. basic_regex. Boost.Regex 1998-2010), Retrieved, from boost.org: http://www.boost.org/doc/libs/1_55_0/libs/regex/doc/html/boost_regex/ref/basic_regex.html

[29] Meyers, S. *Effective C++ (Third ed.)*. Pearson Education, 2005.

[30] Myers, B., Hudson, S. E. and Pausch, R. Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7, 1 2000), 3-28.

[31] Spitzak, B. Documentation. Fast Light Toolkit 2012), Retrieved, from: http://www.fltk.org/documentation.php

[32] Stroustrup, B. *The C++ Programming Language (Fourth ed.)*. Addison-Wesley, 2013.

[33] Stroustrup, B. Copying and rethrowing exceptions. C++11 - the new ISO C++ standard, Retrieved, from: http://www.stroustrup.com/C++11FAQ.html#rethrow

[34] Stroustrup, B. *A History of C++: 1979-1991*. AT&T Bell Laboratories, 1995.

[35] Turner, P. D. Welcome. Crazy Eddie's GUI System (2004 - 2013), Retrieved, from: http://cegui.org.uk/

[36] Woody, B. Yet Another Application Framework, Retrieved, from: http://www.yaaf.org/index.html

[37] Zeitlin, V. Don't use C++ exceptions. Programming Style Guide, Retrieved, from wxwidgets.org: http://www.wxwidgets.org/develop/standard.htm#no_exceptions