Functions and exceptions

Michael Nowak

Texas A&M University

Acknowledgement: Some lecture slides based on those created by Bjarne Stroustrup for use with his textbook

Overview

- Functions
- Errors
  - Sources of errors
  - Your program
  - Kinds of errors
- Handling non-local errors at run-time
  - How to report an error
- Exceptions
- References

Overview

- Functions
- Errors
    - Sources of errors
    - Your program
    - Kinds of errors
- Handling non-local errors at run-time
    - How to report an error
- Exceptions
- References

Functions

- A `function` is a named block of code that can be passed arguments and returns a value to the caller

Functions

- A `function` is a named block of code that can be passed arguments and returns a value to the caller
- We can declare a function by writing a `declarator` of the form `f(args)`, where `f` is the name being introduced and `args` is the parameter list, for example:

Functions

- A `function` is a named block of code that can be passed arguments and returns a value to the caller
- We can declare a function by writing a `declarator` of the form `f(args)`, where `f` is the name being introduced and `args` is the parameter list, for example:
  - `double mult2(double d);`

Functions

- A `function` is a named block of code that can be passed arguments and returns a value to the caller
- We can declare a function by writing a `declarator` of the form `f(args)`, where `f` is the name being introduced and `args` is the parameter list, for example:
  - `double mult2(double d);`
  - Note: the `base type` specifies the `return type` of the `function`

Functions

- A `function` is a named block of code that can be passed arguments and returns a value to the caller
- We can declare a function by writing a `declarator` of the form `f(args)`, where `f` is the name being introduced and `args` is the parameter list, for example:
  - `double mult2(double d);`
  - Note: the `base type` specifies the `return type` of the `function`
- We can define a function by including the declaration with the definition provided in `{}` directly following the parameter list (like a compound statement, we don't have a terminating semi-colon)

Functions

- A `function` is a named block of code that can be passed arguments and returns a value to the caller
- We can declare a function by writing a `declarator` of the form `f(args)`, where `f` is the name being introduced and `args` is the parameter list, for example:
  - `double mult2(double d);`
  - Note: the `base type` specifies the `return type` of the `function`
- We can define a function by including the declaration with the definition provided in `{}` directly following the parameter list (like a compound statement, we don't have a terminating semi-colon)
  - `double mult2(double d) { return d*2; }`

Functions

- We will get into more details about `functions` later, but its helpful to understand them as they help motivate the necessity of `exceptions`

Overview

- Functions
- Errors
    - Sources of errors
    - Your program
    - Kinds of errors
- Handling non-local errors at run-time
    - How to report an error
- Exceptions
- References

Errors

- When we write programs, errors are natural and unavoidable; the question is, how do we deal with them?
  - Organize software to minimize errors
  - Eliminate most of the errors we made anyway
    - Debugging
    - Testing

"My guess is that avoiding, finding, and correcting errors is 95% or more of the effort for serious software development."
– Bjarne Stroustrup

Overview

- Functions
- Errors
    - Sources of errors
    - Your program
    - Kinds of errors
- Handling non-local errors at run-time
    - How to report an error
- Exceptions
- References

Sources of errors

- Poor specification
  - “What’s this suppose to do?”
- Incomplete programs
  - “but I’ll get around to it... tomorrow...”
- Unexpected arguments to functions
  - “but `sqrt()` isn’t suppose to be called with `-1` as its argument”
- Unexpected input
  - “but the user was suppose to input an integer”
- Code that simply doesn’t do what it was supposed to do
  - “so fix it...”

Overview

- Functions
- Errors
  - Sources of errors
  - Your program
  - Kinds of errors
- Handling non-local errors at run-time
  - How to report an error
- Exceptions
- References

Your program

- Should produce the desired results for all legal inputs

Your program

- Should produce the desired results for all legal inputs
- Should give reasonable error messages for all illegal inputs

Your program

- Should produce the desired results for all legal inputs
- Should give reasonable error messages for all illegal inputs
- Need not worry about misbehaving hardware

Your program

- Should produce the desired results for all legal inputs
- Should give reasonable error messages for all illegal inputs
- Need not worry about misbehaving hardware
- Need not worry about misbehaving system software

Your program

- Should produce the desired results for all legal inputs
- Should give reasonable error messages for all illegal inputs
- Need not worry about misbehaving hardware
- Need not worry about misbehaving system software
- Is allowed to terminate after finding an error

Overview

- Functions
- Errors
  - Sources of errors
  - Your program
  - Kinds of errors
- Handling non-local errors at run-time
  - How to report an error
- Exceptions
- References

Kinds of errors

Compile-time errors Errors found by the compiler

- Syntax errors
- Type errors

Kinds of errors

Compile-time errors Errors found by the compiler

- Syntax errors
- Type errors

Link-time errors Errors found by the linker when it is trying to combine object files into an executable program

Kinds of errors

Compile-time errors Errors found by the compiler

- Syntax errors
- Type errors

Link-time errors Errors found by the linker when it is trying to combine object files into an executable program

Run-time errors Errors found by checks made during a running program; that is, errors detected by

- the computer (hardware and/or the operating system)
- by a library (e.g., the standard library)
- by user code

Kinds of errors

Compile-time errors Errors found by the compiler

- Syntax errors
- Type errors

Link-time errors Errors found by the linker when it is trying to combine object files into an executable program

Run-time errors Errors found by checks made during a running program; that is, errors detected by

- the computer (hardware and/or the operating system)
- by a library (e.g., the standard library)
- by user code

Logic errors Errors found by the programmer looking for the causes of erroneous results

Overview

- Functions
- Errors
  - Sources of errors
  - Your program
  - Kinds of errors
- Handling non-local errors at run-time
  - How to report an error
- Exceptions
- References

Handling non-local errors at run-time

- The caller deals with the error

```
int area1 = area(x, y);
if (area1 < 0)
   /* handle error */
else
    /* no error, continue program execution */
```

Handling non-local errors at run-time

- The caller deals with the error

```
int area1 = area(x, y);
if (area1 < 0)
  /* handle error */
else
   /* no error, continue program execution */
```

- The callee deals with errors

```
int area (int length, int width) {
    dobule a = length * width;
    if (a < 0)
        return 0;
    else
        return a;
}
```

Handling non-local errors at run-time

- The caller deals with the error

  ```
  int area1 = area(x, y);
  if (area1 < 0)
     /* handle error */
  else
      /* no error, continue program execution */
  ```

- The callee deals with errors

  ```
    int area (int length, int width) {
        dobule a = length * width;
        if (a < 0)
            return 0;
        else
            return a;
    }
  ```

- Error reporting

Overview

- Functions
- Errors
  - Sources of errors
  - Your program
  - Kinds of errors
- Handling non-local errors at run-time
  - How to report an error
- Exceptions
- References

How to report an error

- Return an “error value” (not general, problematic)

```
int area(int length, int width)
{
    if(length<=0 || width<=0) return -1;
    return length*width;
}
```

- So, “let the caller beware”

```
int z = area(x,y);
if (z<0) return error(``bad area'');
//...
```

- Problems:
  - What if I forget to check the value returned?
  - For some functions, there isn’t a “bad value”

How to report an error

- Set an error status indicator (not general, problematic, don't)

  ```
  int errno = 0;
  int area(int length, int width)
  {
      if(length<=0 || width<=0) errno = 7;
      return length*width;
  }
  ```

- So, "let the caller check"

  ```
  int z = area(x,y);
  if (errno==7) return error(``bad area'');
  //...
  ```

- Problems:
  - What if I forget to check `errno`?
  - How do I pick a value for `errno` that's different from all others?
  - How do I deal with that error?

How to report an error

- The previous means of error reporting are not general...

How to report an error

- The previous means of error reporting are not general...
- Consider that, most of the time we can't change a function that handles errors in a way we don't like...
  - The author of the `std::vector` can detect run-time errors; however, he/she has no idea what the user would like to do about them

How to report an error

- The previous means of error reporting are not general...
- Consider that, most of the time we can't change a function that handles errors in a way we don't like...
  - The author of the `std::vector` can detect run-time errors; however, he/she has no idea what the user would like to do about them
  - The user of the `std::vector` knows how to cope with such errors; however, he/she cannot detect them (otherwise he/she would find them in his/her own code; not left for the library to find)

How to report an error

- The previous means of error reporting are not general...
- Consider that, most of the time we can't change a function that handles errors in a way we don't like...
    - The author of the `std::vector` can detect run-time errors; however, he/she has no idea what the user would like to do about them
    - The user of the `std::vector` knows how to cope with such errors; however, he/she cannot detect them (otherwise he/she would find them in his/her own code; not left for the library to find)
- So we need a means of reporting errors in a general way...

Overview

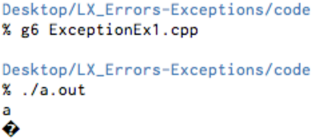
- Functions
- Errors
  - Sources of errors
  - Your program
  - Kinds of errors
- Handling non-local errors at run-time
  - How to report an error
- Exceptions
- References

Exceptions

- Exceptions are C++'s means of separating error reporting from error handling in a general way
  - Just about every kind of error can be reported using exceptions
  - Moreover, you can't forget about an exception: the program will terminate if someone does't handle it...
- You still have to figure out what to do about an exception (every exception thrown in your program)

Exceptions : Example 1

```
#include <iostream>
#include <stdexcept>
#include <limits>
using namespace std;

char to_char(int i) {
        return static_cast<char>(i);
}

int main () {
    cout << to_char(97) << endl;
    cout << to_char(155) << endl;
    return 0;
}
```



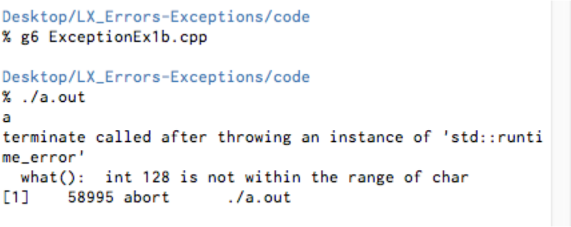
```
Desktop/LX_Errors-Exceptions/code
% g6 ExceptionEx1.cpp

Desktop/LX_Errors-Exceptions/code
% ./a.out
a
�
```

Exceptions : Throw, Try and Catch

```
char to_char(int i) {
    if (i < numeric_limits<char>::min() || numeric_limits<char>::max() < i) {
        const string s = to_string(i);
        throw runtime_error("int␣" + s + "␣is␣not␣within␣the␣range␣of␣char␣");
    }
    // we get here if and only if an exception is not thrown
    return static_cast<char>(i);
}
```

- When an unexpected condition happens, we can `throw` an exception
    - `to_char` will either return the corresponding *char* of the numeric value `i`
    - **or** it will throw a `runtime_error`

Exceptions : Example 1b



```
Desktop/LX_Errors-Exceptions/code
% g6 ExceptionEx1b.cpp

Desktop/LX_Errors-Exceptions/code
% ./a.out
a
terminate called after throwing an instance of 'std::runti
me_error'
  what():  int 128 is not within the range of char
[1]    58995 abort      ./a.out
```

```
#include <iostream>
#include <string>
#include <stdexcept>
#include <limits>
using namespace std;
char to_char(int i) {
    if (i < numeric_limits<char>::min() || numeric_limits<char>::max() < i) {
        const string s = to_string(i);
        throw runtime_error("int␣" + s + "␣is␣not␣within␣the␣range␣of␣char␣");
    }
    // we get here if and only if an exception is not thrown
    return static_cast<char>(i);
}
int main () {
    cout << to_char(97) << endl;
    cout << to_char(128);
    return 0;
}
```

Exceptions : Throw, Try and Catch

- In order to handle the problem, we must indicate that we are willing to `catch` the exception of the type used to report the problem

Exceptions : Throw, Try and Catch

- In order to handle the problem, we must indicate that we are willing to `catch` the exception of the type used to report the problem
- If we do not catch the exception anywhere, the program will terminate (as seen in the previous example)

Exceptions : Throw, Try and Catch

- In order to handle the problem, we must indicate that we are willing to `catch` the exception of the type used to report the problem
- If we do not catch the exception anywhere, the program will terminate (as seen in the previous example)
- Therefore, we introduce a try-block around the code where an exception might occur

```
try {
    cout << to_char(97) << endl;
    cout << to_char(128);
}
```

Exceptions : Throw, Try and Catch

- In order to handle the problem, we must indicate that we are willing to `catch` the exception of the type used to report the problem
- If we do not catch the exception anywhere, the program will terminate (as seen in the previous example)
- Therefore, we introduce a try-block around the code where an exception might occur

```
try {
    cout << to_char(97) << endl;
    cout << to_char(128);
}
```

- The try-block is followed by the *exception handler*, which specifies the type of objects that it can catch

```
catch (const runtime_error& e) { // exception handler
    cerr << "Exception:␣" << e.what() << endl;
}
```

Overview

Functions

Errors

- Sources of errors
- Your program
- Kinds of errors

Handling non-local errors at run-time

- How to report an error

Exceptions

References

References

- Lippman, B., Lajoie, Josee, & Moo, B. E. (2016). *C++ primer* (5th ed.). Addison-Wesley.
- Stroustrup, B. (2014). *Programming: principles and practice using C++* (2nd ed.). Addison-Wesley.