CS 162
Intro to Programming II

Exceptions
Old School

- A typical way to handle error conditions is through the return value.
- For example, suppose we create a `loadFile()` function that returns true if it loaded the file correctly and false if an error occurred.

```cpp
int loadFile(std::string filename) {
    /* do stuff in here */
}
```
Old School

• To check for errors, we need to check the return value of load File every time we call it

```cpp
if( loadFile("words.txt") ) {
    /* file loaded successfully */
} else {
    /* file not loaded successfully */
    std::cout << "File didn't load" << std::endl;
    exit(1);
}
```
Old School

• What happens when a method detects an error condition?
• Terminate the program
  – Very annoying
• Return an error return value
  – But return value may not be checked by caller
  – Caller might not be able to do anything
  – What if you needed to add new return values for new error conditions? Hard to do
Exceptions

• A much better error-handling idea: throw an exception
• Can’t be overlooked
• Must be handled by an exception handler (not just the caller of the failed method)
• Makes code cleaner and easier to read separates code into:
  – normal execution
  – and error handling
Exceptions

• An exception is a simple type or an object that describes an unusual or erroneous situation
• Exceptions are thrown by a program, and may be caught and handled by another part of the program
• A program can be separated into normal execution flow and exception execution flow
Exceptions

• Try-Throw-Catch:

```java
try {
    Do stuff
    If error, throw exception
    Do more stuff
}
catch(type e) {
    Handle the exception e
}
```

Otherwise
Skip this
Exceptions

```cpp
void getAndReplaceChar(std::string& s) {
    try {
        char ch;
        std::cin >> ch;
        if( !isalpha(ch) )
            throw ch;
        replaceChar(s, ch);
        std::cout << s << std::endl;
    }
    catch( char e ) {
        std::cout << "Invalid character " << e << std::endl;
    }
}
```
Exceptions

```cpp
void getAndReplaceChar(std::string& s) {
    try {
        char ch;
        std::cin >> ch;
        if( !isalpha(ch) )
            throw ch;
        replaceChar(s, ch);
        std::cout << s << std::endl;
    }
    catch( char e ) {
        std::cout << "Invalid character " << e << std::endl;
    }
}
```
Exceptions

• You can define your own Exception class to catch specific types of exceptions

```cpp
class DigitException {
public:
    DigitException(char chValue) : ch(chValue) {}
    char getChar() { return ch; }
private:
    char ch;
};
```
Exceptions

```cpp
void getAndReplaceChar(std::string& s) {
    try {
        char ch;
        std::cin >> ch;
        if( isdigit(ch) )
            throw DigitException(ch);
        replaceChar(s, ch);
        std::cout << s << std::endl;
    }
    catch( DigitException e ) {
        std::cout << "Invalid character " << e.getChar() << std::endl;
    }
}
```
Exceptions

• An exception class can be trivial if you don’t need to store much information
• For example, the DigitException class could just be:
  
  ```java
  class DigitException {
  }
  ```
• The code becomes simpler
void getAndReplaceChar(std::string& s) {
    try {
        char ch;
        std::cin >> ch;
        if( isdigit(ch) )
            throw DigitException();
        replaceChar(s, ch);
        std::cout << s << std::endl;
    }
    catch( DigitException ) {
        std::cout << "Invalid character" << std::endl;
    }
}
Exceptions

- The catch blocks are processed in order

```java
catch (DigitException e) {
    /* Do stuff */
}
catch (WeirdCharException e) {
    /* Do stuff */
}
catch (GenericException e) {
    /* Do stuff */
}
```

Do more specific exceptions first.
Exception Specification

• You can catch the exception in a different function

```cpp
void getAndReplaceChar(std::string& s) throw (DigitException)
{
    char ch;
    std::cin >> ch;
    if( isdigit(ch) )
        throw DigitException(ch);
    replaceChar(s, ch);
    std::cout << s << std::endl;
}
```
Exception Specification

• Exception specifications describe which exceptions “get outside” the function
• This allows other calling functions to catch these exceptions
  – Eg. If \texttt{main()} called \texttt{getAndReplaceChar()}, the main function needs to catch the specified exception (see next slide)
int main(int argc, char** argv) {
    try{
        std::string s("computer");
        getAndReplaceChar(s);
    }
    catch( DigitException e ) {
        std::cout << "Invalid character " << e.getChar() << std::endl;
    }
}
Exception Specification

• If you catch an exception inside `getAndReplaceChar()`, there is no need to list it in the exception specifications

• If your function throws an exception that is not in the exception specification, the `unexpected()` function is called which terminates the program
Exception Specification

• You can have multiple exception specifications on a function

```c
void foo() throw (DigitException, WeirdCharException)
```

• You can also specify that a function doesn’t throw exceptions

```c
void foo() throw ()
```
When to Throw an Exception

• You typically throw and catch the exceptions in separate functions

• If you can easily handle the error condition, don’t bother throwing the exception

• Only throw the exception if the error is handled depending on how and when the function is called

• Rely on the programmer who called the function to handle the error <<< Why?
When to Throw an Exception

• Eg. suppose you have a function called `loadFile(string filename)` which loads a file

• If this fails, what should you do? It depends:
  – To give the user a second chance to specify a file name, you throw an exception in `loadFile` but catch it in the function that calls `loadFile`
  – If you want to terminate the program, then just do it without throwing the exception