Unit 8

Agenda

Exceptions

- Definition
- try-catch-throw mechanism
- Hierarchies of exceptions
- Catching exceptions
- Exceptions in constructors, destructor and member initialization
- Exception specifications

Error handling

Errors occur during execution of a program (run-time)

- Detection of an error

- Handling of an error

- In separate modules of the program
- At separate moments
- Different actions

Exception

DEFINITION [Exception] An exception is a run-time error.

Examples: not enough memory, a file cannot be opened, invalid object received for an operation, etc.

How to deal with exceptions?

- terminate the program => not appropriate
- return a value representing “error” => what is an acceptable error code? It has to be checked by the caller.
- return a legal value and leave the program in an illegal state => caller has to test an `errno` state variable
- call a function (error handler) supplied to be called in case of “error” => no control over caller’s code

Useful, ordinary code is mixed with error-handling code => less readable programs, hard to maintain => programs become “brittle”

Exception handling

DEFINITION [Exception handling] Exception handling is a mechanism that allows two separately developed program components to communicate when a program anomaly, i.e. exception, is encountered during the execution of the program.

- Is an alternative to the traditional techniques when they are insufficient, inelegant, or error-prone
- Is complete; it can be used to handle all errors detected by ordinary code
- Allows the programmer to explicitly separate error-handling code from “ordinary code” thus making the program more readable
- Supports a more regular style of error handling, thus simplifying cooperation between separately written program fragments

Designed to handle only synchronous exceptions; asynchronous exceptions require fundamentally different approaches.

Exception handling mechanism in C++

An exception is an object representing an exception occurrence.

1. Code (Component) that detects an error **throws** an exception (‘incarnated’ as a regular object).

2. The effect of throw is to unwind the stack until a suitable **catch** is found (in a function that directly/indirectly called the function that threw the exception).

```
void operation() {
     // start ‘normal’ execution flow
     if(some_exceptional_case_occurred) {
          // throws an exception object
          throw ExceptionType1{};
     }
     // continue ‘normal’ execution flow
}
```

Exception handling mechanism in C++

1. Code that handles an error ‘**tries**’ to execute an operation (call a function).

2. **Catches** all the errors that should be handled.

```
void foo() {
    try { // … code that may raise an exception …
         // other  statements…
         operation();
         // other statements…
    }
    catch(ExceptionType1 e1) { // handle Exception type1
    }
    catch(ExceptionType2 e2) { // handle Exception type2
    }
}
```

First exception is thrown will stop the execution of remaining statements in the try block.

For more examples, see slides “Exceptions in constructors”

Example

```
class FileNotFoundException {
public:
    string describe() { return "File not found exc."; }
} ;

class FileWriteException {
public:
    string describe() { return “File write exc.”; }
} ;

class File {
    FILE* handle;
    string name;
public:
    File(const char* name, const char* mode);
    File(const File&) = delete; // forbid copy ctor
    void write(const char* data, int size);
    ~File();
};

File::File(const char* n, const char* mode)
    : name{n}, handle{fopen(n, mode)} {
    if (handle==nullptr) {
        throw FileNotFoundException { };
    }
}

File::~File() {
    if (handle!=nullptr) {
        fclose(handle);
    }
}
```

```
void File::write(const char* data, int size) {
     size_t written = fwrite(data, 1, size, handle);
     if (written!=size)
          throw FileWriteException { };
}



int main(int, char*[]) {

    try {
        char* test = “Write test in my sample file.”;
        File f(“sample.txt”, “w+”);
        f.write(test, strlen(test));
    }
    catch(FileNotFoundException e) {
        cerr << e.describe();
    }
    catch(FileWriteException e) {
        cerr << e.describe();
    }

}
```

Example

```
class FileNotFoundException {
public:
    string describe() { return "File not found exc."; }
} ;

class FileWriteException {
public:
    string describe() { return “File write exc.”; }
} ;

class File {
    FILE* handle;
    string name;
public:
    File(const char* name, const char* mode);
    File(const File&) = delete; // forbid copy ctor
    void write(const char* data, int size);
    ~File();
};

File::File(const char* n, const char* mode)
    : name{n}, handle{fopen(n, mode)} {
    if (handle==nullptr) {
        throw FileNotFoundException { };
    }
}

File::~File() {
    if (handle!=nullptr) {
        fclose(handle);
    }
}
```

New object created here.

```
void File::write(const char* data, int size) {
     size_t written = fwrite(data, 1, size, handle);
     if (written!=size)
          throw FileWriteException { };
}
```

New object created here.

```
int main(int, char*[]) {

    try {
        char* test = “Write test in my sample file.”;
        File f(“sample.txt”, “w+”);
        f.write(test, strlen(test));
    }
    catch(FileNotFoundException e) {
        cerr << e.describe();
    }
    catch(FileWriteException e) {
        cerr << e.describe();
    }

}
```

Same here..

A copy is created and passed to this catch clause using copy ctor

Example

```
class FileNotFoundException {
public:
    string describe() { return "File not found exc."; }
} ;

class FileWriteException {
public:
    string describe() { return “File write exc.”; }
} ;

class File {
    FILE* handle;
    string name;
public:
    File(const char* name, const char* mode);
    File(const File&) = delete; // forbid copy ctor
    void write(const char* data, int size);
    ~File();
};

File::File(const char* n, const char* mode)
    : name{n}, handle{fopen(n, mode)} {
    if (handle==nullptr) {
        throw FileNotFoundException { };
    }
}

File::~File() {
    if (handle!=nullptr) {
        fclose(handle);
    }
}
```

New object created here.

```
void File::write(const char* data, int size) {
     size_t written = fwrite(data, 1, size, handle);
     if (written!=size)
          throw FileWriteException { };
}
```

New object created here.

```
int main(int, char*[]) {

    try {
        char* test = “Write test in my sample file.”;
        File f(“sample.txt”, “w+”);
        f.write(test, strlen(test));
    }
    catch(FileNotFoundException& e) {
        cerr << e.describe();
    }
    catch(FileWriteException& e) {
        cerr << e.describe();
    }

}
```

Same here..

No more copy!

Grouping of Exception (I)

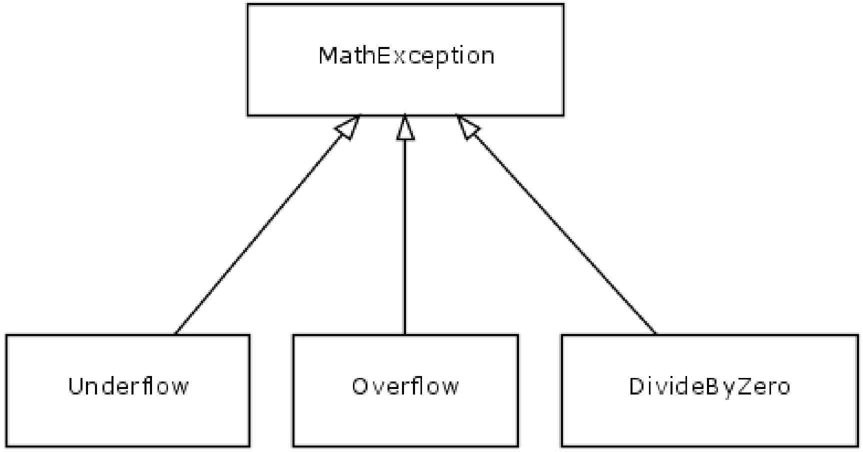
Exceptions fall naturally into families => use inheritance to structure exceptions

```
class MathException {
public:
     virtual string getDescription() {
         cerr << “Math ex”;
     }
};

class Overflow : public MathException {
public:
     string getDescription() {
          cerr << “Overflow ex”;
     }
};

class Underflow : public MathException {
     string getDescription() {
          cerr << “Underflow ex”;
     }
};

class DivideByZero : public MathException {
     string getDescription() {
          cerr << “Division by 0”;
     }
};
```



Exception hierarchy can include multiple inheritance as well.

Exceptions hierarchies increase the robustness of your code.

Grouping of Exception (II)

```
void foo() {
    try {
        // using math functions
    }
    catch(Overflow) {
         // handle Overflow exceptions
    }
    catch(MathException e) {
        // handle any Math exceptions that is not Overflow
        // MathException::getDescription
        cerr << e.getDescription();
    }
}
```

Giving a name is optional.

Only info specific to MathException type is available in catch branch, regardless of the “original” type of the exception (slicing effect).

```
void f() {
    try {
        // using math functions
    }
    catch(Overflow e) {
         // handle Overflow exceptions
         cerr << e.getDescription();
    }
    catch(MathException& me) {
        // handle any Math exceptions that is not Overflow
        // call appropriate getDescription
        // (e.g. of class Underflow if an underflow exception was thrown)
        cerr << me.getDescription();
    }
}
```

To avoid slicing and keep “original” type use **reference** type.

Catching exceptions

```
void f() {
    try {
        throw E();
    }
    catch(H) {
    }
}
```

Handler H is invoked when:

[1] if H is the same type as E

[2] if H is an unambiguous public base of E

[3] if H and E are pointer types and [1] or [2] holds for the types to which they refer

[4] if H is a reference and [1] or [2] holds for the type to which H refers.

`const` can be used to denote exceptions that are not modified

Example: `catch(const MathException& e) { /* */ }`

Basically, an exception is copied when it is thrown so that the handler gets a copy of the original (see objects as function arguments).

REMARKS

1. Order of handlers is important. Why?

2. `catch(...)` – catches any exception because ellipsis indicates 'any argument'

Catching exceptions – handlers order

```
void f() {
    try {
        // some math operations that throw Overflow exception!!
    }
    catch(Overflow e) {
         // handle Overflow exceptions
         cerr << “Overflow: ” << e.getDescription();
    }
    catch(MathException me) {
        // handle any Math exceptions that is not Overflow
        // call appropriate getDescription
        // (e.g. of class Underflow if an underflow exception was thrown)
        cerr << “Math exception: ” << e.getDescription();
    }
}
```

The rule is: catch more specific types first!

...and the output is:

Overflow:

```
void f() {
    try {
        // some math operations that throw Overflow exception!!
    }
    catch(MathException me) {
        // handle any Math exceptions that is not Overflow
        // call appropriate getDescription
        // (e.g. of class Underflow if an underflow exception was thrown)
        cerr << “Math exception: ” << e.getDescription();
    }
    catch(Overflow e) {
         // handle Overflow exceptions
         cerr << “Overflow ” << e.getDescription();
    }
}
```

...and the output is:

Math exception:

Re-throwing exceptions

```
void f() {
    try {
          // code
    }
    catch(MathException& e) {
         if(cannot_handle_it_completely)
              throw; // re-throw this exception
         else
              // do the job & consume the exc.
    }
}
```

Exceptions in constructors

Q: How to report errors from constructors?

A: Exceptions is an elegant mechanism for this.

```
class Vector {
    static const int MAX_SIZE = 1000;
public:
    class BadSize { } ;
    Vector(int sz) {
        if(sz<0 || sz>MAX_SIZE) throw BadSize{};
        // do the actual job
    }
};
```

```
void f(int size) {
    try {
        Vector v(size);
        cout << “Going on...”;
    }
    catch(Vector::BadSize& bs) {
        cerr << “Invalid size “
            << size << endl;
    }
    cout << “Return.\n”;
}

int main() {
    f(-10); // Exception is thrown
    f(5); // OK
    return 0;
}
```

…and the output is:

```
Invalid size -10
Return.
Going on…
Return.
```

Exceptions in members initialization

Q: What happens if a member initialization throws an exception?

A: The constructor can catch this kind of problems by using try-catch block in its initialization list.

```
class X  {
      Vector v;
public:
     X(int size);
};

X::X(int size)
try
   : v(size) // initialize v by size
{
    // body of X::X(int) ctor
}
catch(Vector::BadSize) {
}
```

Copy-constructors and assignment operators are special case of constructors/operators because they are invoked automatically, they deal with acquiring + releasing of resources.

Exceptions in destructors (I)

A destructor can be called:

[1] in ‘normal’ way, when objects are destroyed

[2] during exception handling, when during stack unwinding a scope containing an object with destructor is exited

```
void  f() {
     try {
          X anXObject;
          // some operations that may generate an Exception
     } // => destroy anXObject, using X::~X (), in case [2]
     catch(Exception& e) {
     }

     X anotherXObject;
} // => destroy anotherXObject, using X::~X(), in case [1]
```

Use `uncaught_exception` function in the destructor of class X to decide whether the destructor was called due to case [1] (returns false) or [2] (returns true).

Exceptions in destructors (II)

In case [2] (during exception handling), if an exception “escapes” from the destructor then `std::terminate` function is called to signal an abnormal program termination.

To protect itself from this kind of disaster, a destructor can use try-catch block.

```
X::~X() {
     try {
          // do the task that might raise an exception
     }
     catch(...) {
          // handle any exception here
     }
}
```

Uncaught exceptions?

If an exception is thrown but not caught anywhere in the program, the function `std::terminate` will be called.

To handle all the exceptions that can be thrown in a program, the function `main` should read like:

```
int main(int, char*[]) {
     try {
          // do the actual job
     }
     catch(…) {
          // handle any uncaught exception so far
     }
}
```

Exception specifications (deprecated on C++ 11 onwards)

Specify the set of exceptions that might be thrown by a function as part of function declaration.

```
void add(Matrix& m1, Matrix& m2) throw (MathException,
                                        bad_alloc);


class X {
     void add(int) throw  (Overflow);
};
```

Syntax

```
type name(arg_list) throw (exception1,
                            ...
                            exceptionN);
```

- If exception list is missing then the function can throw **any** exception.

- `void f() throw();` ⟺ `void f() noexcept;` //doesn't throw any exception

REMARKS

- Exception specifications must be included in both function's declaration and definition.
- A virtual function can be overridden only by a function that has an exception-specifications at least as restrictive as its own.
- If `noexcept` operator is specified and function throws an exception then the program unconditionally terminates by calling `std::terminate` function. It doesn't invoke destructors from calling functions
- If other exception than the ones specified in exception specification is thrown => call to `std::unexpected()`, which – by default – calls `std::terminate()`, which calls `abort()`.
- User-defined handlers for std::unexpected, std::terminate using set_unexpected, respectively set_terminate functions provided in standard library.

noexcept Operator

```
void my_fct(T& x) noexcept(Is_pod<T>());
```

- It means that my_fct may not throw exception(s) if the predicate Is_pod<T>() is true but may throw if it is false.
- For example, if T is a POD it does not throw, whereas other types (e.g., a string or a vector) may throw
- The predicate in a noexcept() specification must be a constant expression
- The noexcept() operator takes an expression as its argument and returns true if the compiler “knows” that it cannot throw and false otherwise.

```
template<typename T> void call_f(vector<T>& v) noexcept( noexcept(f(v[0]) )
{
   for (auto x : v)
         f(x);
}
```

- noexcept operator simply looks at every operation in expr and if they all have noexcept specifications that evaluate to true, it returns true. A noexcept(expr) does not look inside definitions of operations used in expr.

Exceptions that are not errors

Exception-handling mechanism in C++ is a non-local control structure based on stack unwinding that can be seen as an alternative return mechanism.

There are legitimate uses of exception that have nothing to do with errors.

```
void f(Queue& q) {
     try {
          for(;;) {
               X m = q.get(); // get throws Empty exception if queue is empty
               // use m
          }
     }
     catch(Queue::Empty) {
          return;
     }
}
```

⇕

```
void f(Queue& q) {
     for(; !q.isEmpty(); X m = q.get())
          // use m
}
```

Implementation remarks

- Dynamic (run-time) detection of exceptions.

- Exception handling can be implemented so that there is no run-time overhead when no exception is thrown and throwing an exception is not all that expensive as compared to calling a function.

- C++ Standard Library (STL) exposes a hierarchy of predefined exceptions: `bad_alloc`, `bad_cast`, `bad_exception`, `overflow_error` etc., all of them derived from class `exception`.

- The standard-library exception classes, such as `runtime_error` and `out_of_range`, take a string argument as a constructor argument and have a virtual function `what()` that will regurgitate that string.

Further Reading

[Stroustrup, 1997] Bjarne Stroustrup – The C++ Programming Language 3rd Edition, Addison Wesley, 1997 [Chapter 14]

[Stroustrup, 2013] Bjarne Stroustrup – The C++ Programming Language 4th Edition, Addison Wesley, 2013 [Chapter 13]