

A taste of Exceptions

Exceptions and exception handling support the development of robust programs.

Robust programs should

- detect unexpected, erroneous situations
- repair the error or exit in a “safe” mode

Motivation for Exceptions

Let us consider the following situation: Students’ favourite courses are stored in a data structure (`DOCfavCourse`). The student rep prints the name of the lecturer teaching the preferred course of each of his friends (in `RigasFriends`).

The problem is, that we need to cater for the cases where we deal with absent students, or with students without preferences. This requires defensive programming, which, without exceptions an be very tedious. ...read on

```
class Student { ... };
```

```
class Lecturer { ... };
```

```
class Course{
public:
    char* name;
    Lecturer* theLecturer();
    // returns the lecturer teaching course by looking
    // up in the DOC database, possibly 0
    . . . };
```

```
class StudentList{
public:
    bool finished(); // at the end of the list
    Student* next(); // the next student in list
    . . . };
```

```
class StudentToCourse{
public:
    // lookup table storing students' preferred courses
    Course* preference(Student* st);
    // returns a pointer to st's preferred course,
    // returns 0 ifr no entry in table
    void addPreference(Student* st, Course* c);
    // sets st's preference to c
```

```

        // overwrites old preferences - if any
        StudentToCourse();
    . . . };

void statistics(StudentList sl, StudentToCourse favs)
// prints the favourite courses of all students
// in the list sl, according to information in favs
{
    // auxiliary variables
    Student* s; Course* c; Lecturer* l;

    // print report on everybody's preferences
    cout << "Dear Rigas, your friends like\n";
    // NOTE Loop simplified, and skips first student!
    while (!sl.finished())
    { // read each student's favourite
        s=sl.next();
        c=favs.preference(s);
        l=(c->theLecturer());
        // print the preferences
        cout << s->name << " likes " << c->name
            << " taught by " << l->name };
    cout << " that's all, Rigas!\n";return; };

int main() {

```

```

StudentToCourse DOcfavCourse;
. . . // James reads the questionnaires
      // and stores in DOCFavCourse

StudentList JamesFriends;
. . . // James enters his friends in myFrinds

statistics (JamesFriends, DOcfavCourse);
}

```

The above program is *not* robust because it does not check for possible null pointers.

We *could* make it more robust as follows:

Solution 1

```

void statisticsR1 (StudentList sl, StudentToCourse favs)
// same as statistics, but robust for null pointers
// a null pointer causes error message and the
// function stops
{ // auxiliary variables
  Student* s; Course* c; Lecturer* l;

```

```

// print report on everybody's preferences
cout << "Dear Rigas, your friends like\n";
while (!sl.finished())
{ // read each student's favourite
  s=sl.next();
  c=favs.preference(s);
  if (c==0) { cerr << "non-existing course liked by"
              << s->name; return; }
  l=(c->theLecturer());
  if (l==0) { cerr << "non-existing lecturer\n";
              return; }
  // print the preferences
  cout << s->name << " likes " << c->name
        << " taught by " << l->name; };
  cout << " that's all, Rigas!\n"; return;
};

```

Solution 1 is OK, but

Solution 2

```

void statisticsR2(StudentList sl, StudentToCourse favs)
// same as solutionR2, but more "forgiving"

```

```

{ // auxiliary variables
  Student* s; Course* c; Lecturer* l;

  // print report on everybody's preferences
  cout << "Dear Rigas, your friends like\n";
  while (!sl.finished())
  { // read each student's favourite
    s=sl.next();
    c=favs.preference(s);
    if (c==0) { cerr << "non-existing course liked by"
                << s->name; }
    else { l=(c->theLecturer());
          if (l==0)
            { cerr << "non-existing lecturer\n"; }
          // print the preferences
          else { cout << s->name << " likes "
                    << c->name << " taught by "
                    << l->name; } } } };
  cout << " that's all, Rigas!\n"; return; };

```

Solution 2 is OK, but

Exceptions

A function that finds a problem that it cannot cope may *throw* an exception, in the hope that its (direct or indirect) caller is able to handle that exception.

Exception throwing is described by the statement

```
throw expression;    // any expression allowed!
```

Exception handlers are the catch- parts of try-catch statements, eg:

```
try{  
    ... // enter friends in RigasFriends  
}  
catch ( ClassA z)    {  
    ... // handler for ClassA exceptions  
}  
catch ( ClassB b)    {  
    ... // handler for ClassB exceptions  
}
```

Any exception thrown is propagated to most enclosing function. If an appropriate handler is found, then that handler is executed, and

after its execution continues after the handler, otherwise the exception is propagated to the next enclosing function call.

A handler is appropriate for an exception if it mentions a superclass of the class of the exception object.

Example of exception throwing

Eg, function `StudentToCourse::preference (Student* s)` should throw an exception if it does not find an entry for `s`.

Define special purpose exception classes:

```
class NonExistingCourse{
public:
    Student* reportedBy;
    NonExistingCourse (Student* s); };

class NonExistingLecturer{ };
```

... and slightly modify the class `StudentToCourse`:

```

class StudentToCourse{
public:
    // lookup table storing students' preferred courses
    Course* preference(Student* st);
    // returns non-null pointer to st's preferred course,
    // throws exception NonExistingCourse(s) if no entry
    void addPreference(Student* st, Course* c);
    StudentToCourse();
private:
    class Link{
        Student* theStudent; Course* theCourse;
        Link* next;
        Link(Student* s, Course* c)
            theStudent(s), theCourse(c) { next = NULL; };
        friend class StudentToCourse; } ;
    Link* theList; };

```

And now, modify the body of StudentToCourse::preference:

```

Course* StudentToCourse::preference(Student* s)
{ Link *aList = theList;
  while ( aList != NULL )

```

```
{ if ( s == aList->theStudent )
    { return aList->theCourse; };
  aList=aList->next; };
throw NonExistingCourse(s);
return 0; /* never executed */ };
```

Similarly for the function `Course::preference`:

```
class Course{
public:
  char* name;
  Lecturer* theLecturer();
  // returns the lecturer teaching course by looking
  // up in the DOC database,
  // if no entry found, throws NonExistingLecturer
  . . . };
```

Example of exception handling

Solution 3

We modify the `main` function as follows:

```

int main() {
    StudentToCourse DOCfavCourse;
    // Rigas reads the various questionnaires
    // and stores in DOCFavCourse
    StudentList RigasFriends;
    // Rigastian enters his friends in myFrinds
    try{
        StudentToCourse DOCfavCourse;
        . . . // read questionnaires,
              // store in DOCFavCourse
        StudentList RigasFriends;
        . . . // enter friends in RigasFriends
        statistics(RigasFriends,DOCfavCourse);
    }
    catch ( NonExistingCourse nc)
    {   cerr << "non existitng course reported by"
        << ( nc.reportedBy->name) << "\n";
    }
    catch ( NonExistingLecturer)
    {   cerr << "non existitng lecturer \n";
    };
    return 0; }

```

Here the functions `preferences (...)` or `theLecturer ()`, may throw exceptions which will be propagated to most enclosing function call, ie `statistics (...)`. No handler is found, therefore the exception is propagated to the next enclosing function call, i.e. `main ()` in search of a handler. Thus, `NonExistingCourse` or `NonExistingLecturer` exceptions thrown will be caught by the handlers in `main ()`.

Solution 3 corresponds to

Solution 4

Alternatively, we could have left the `main ()` function unmodified and inserted handlers in `statistics (...)`:

```
void statistics(StudentList sl, StudentToCourse favs)
{
    // auxiliary variables
    Student* s; Course* c; Lecturer* l;

    // print report on everybody's preferences
    cout << "Dear James, the preferences\n";
}
```

```

while (!sl.finished())
{ // read each student's favourite
  try{
    s=sl.next();
    c=favs.preference(s);
    l=(c->theLecturer());
    // print the preferences
    cout << s->name << " likes " << c->name
          << " taught by " << l->name;
  }
  catch ( NonExistingCourse nc)
  { cerr << "non existitng course reported by"
        << ( nc.reportedBy->name) << "\n"; }
  catch ( NonExistingLecturer)
  { cerr << "non existitng lecturer \n"; }
};
cout << " that's all, James!\n";
return;
};

```

The difference between solution 3 and solution 4 is:

Exceptions - Conclusions

Robust programming through exceptions (eg solution 3/4) is better than through explicit checks (eg solution 1/2), is better because exceptions:

- clearly distinguish “normal” and “abnormal”, unexpected behaviour,
- support better, and more flexible error reporting error reporting,
- decouple error detection (**throw**) from error handling (**try-catch**),
- one handler for several errors possible,
- flexibility in error handling, eg handler for same error may depend on the dynamic context

Exceptions - Applications

We could have used exceptions is earlier examples:

-
-

Exceptions - Summary

- may be thrown for any unexpected erroneous situation, not just null pointers
- support development of robust code.
- allow for unconventional flow of control, and for functions to have two “return paths”
- are any values; they may be objects, and they can be used to pass additional information to the exception handler.
- are independent of the object oriented paradigm, but are an essential ingredient for programming in the large; in C++ they use the oo paradigm
- will come handy for the group projects and future
- Unfortunately, the compiler does not check that any thrown exception will have a handler (in Java it does)

The Programming Paradigms Compared

	Object Oriented	Declarative
the main question is		
cases expressed through		
side-effects expressed through		
good for		
....		

end of course – thank you all!