Programming Languages

Exception handling & Concurrency

Giuseppe Narzisi
G22.2110-002
Summer 2008
C++ standard exceptions

• The **C++ Standard library** provides a base class specifically designed to declare objects to be thrown as exceptions.
  – Usual default and copy constructors, operators and destructors,
  – Plus an additional virtual member function called **what** that returns a null-terminated character sequence (char *)
  – It can be overwritten in derived classes to contain some sort of description of the exception.
Example

```cpp
#include <iostream>
#include <exception>
using namespace std;

class myexception: public exception
{
    virtual const char* what() const throw() {
        return "My exception happened";
    }
} myex;

int main () {
    try {
        throw myex;
    }
    catch (exception& e) { cout << e.what() << endl; }
    return 0;
}
```
Standard library - exceptions

- All exceptions thrown by components of the C++ Standard library throw exceptions derived from this `std::exception` class. These are:

<table>
<thead>
<tr>
<th>exception</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bad_alloc</td>
<td>thrown by new on allocation failure</td>
</tr>
<tr>
<td>bad_cast</td>
<td>thrown by dynamic_cast when fails with a referenced type</td>
</tr>
<tr>
<td>bad_exception</td>
<td>thrown when an exception type doesn't match any catch</td>
</tr>
<tr>
<td>bad_typeid</td>
<td>thrown by typeid</td>
</tr>
<tr>
<td>ios_base::failure</td>
<td>thrown by functions in the iostream library</td>
</tr>
</tbody>
</table>

- For example, if we use the operator new and the memory cannot be allocated, an exception of type `bad_alloc` is thrown:

```cpp
try {
    int * myarray = new int[1000];
}
catch (bad_alloc&) {
    cout << "Error allocating memory." << endl;
}
```
The empty *throw* statement

• By placing an *empty throw* statement in the code handling an exception the received exception is passed on to the next level able to process that particular type of exception
Polymorphic exception

```cpp
void initialExceptionHandler(Exception *e)
{
    // show the plain-text information
    cout << e->toString() << endl;
    // Can we process it?
    if (e->severity <= Exception::Warning)
        e->process(); // It's either a message
    // or a warning
    else
        throw; // No, pass it on
}
```

• `initialExceptionHandler()` can be used for a variety of thrown exceptions, as long as the argument used with `initialExceptionHandler()` is compatible with the nature of the received exception.
Thread Scheduling

• The Java language specification does not dictate how threads are scheduled.
  – Most JVM implementations are using a pre-emptive scheduler.

• Threads do have *priorities*.
  – A lower priority thread will never run as long as a higher priority thread is ready.
  – Priority can be accessed with `getPriority` and `setPriority`
Synchronized Methods

• Concurrent access to objects can be very dangerous.
  – Consider a linked list, where one thread is removing the “last” element in the list at the same time another thread is inserting a new element.

• To prevent concurrent access to objects, methods can be declared synchronized.
Dining Philosophers
Edsger Dijkstra (1971)

• N “philosophers” (threads) and N “chopsticks” (resources).
  – Each philosopher needs a specific pair of chopsticks to eat.
  – Philosophers sometimes eat (using resources) and sometimes just sit and think (no resources).

• The problem is to allocate chopsticks to philosophers so that there is no deadlock, no livelock and no starvation.
Deadlock

• What if every philosopher sits down about the same time and picks up his left chopstick?
Starvation

- Two philosophers are fast thinkers and fast eaters
- They can lock their chopsticks and eat. After finish eating and before their neighbors can lock the chopsticks and eat, they come back again and lock the chopsticks and eat.
Trivial solution

• The deadlock problem is trivially solved for dining philosophers
  – make any one of the philosophers “left handed”.
  – literally, have philosopher 0 pick up chopstick 1 before picking up chopstick 0.
  – This fix will “break the cycle” of dependence that leads to deadlock.
Waiter solution

• Waiter Solution
  – Philosophers must ask his permission before taking up any forks.
  – Since the waiter is aware of which forks are in use, he is able to arbitrate and prevent deadlock.