8/18/2010

Exception-safe Code

Paul Mercea

dkfz.
GERMAN
CANCER RESEARCH CENTER
IN THE HELMHOLTZ ASSOCIATION

Example function

- example function in class for representing GUI menus with background images used in threaded environment

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
    lock(&mutex);                    // acquire mutex
    delete bgImage;                  // get rid of old background
    ++imageChanges;                  // update image change count
    bgImage = new Image(imgSrc);     // install new background
    unlock(&mutex);                  // release mutex
}
```

```
class PrettyMenu{
public:
 ...
 void changeBackground(std::istream& imgSrc);
 ...
private:
 Mutex mutex;
 Image* bgImage;
 int imageChanges;
};
```

Example function - „as bad as it gets“

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
    lock(&mutex);                    // acquire mutex
    delete bgImage;                  // get rid of old background
    ++imageChanges;                  // update image change count
    bgImage = new Image(imgSrc);     // install new background
    unlock(&mutex);                  // release mutex
}
```

Example function - „as bad as it gets“

leak of resources

corrupted data structures

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
    lock(&mutex);                          // acquire mutex
    delete bgImage;                        // get rid of old background
    ++imageChanges;                        // update image change count
    bgImage = new Image(imgSrc);           // install new background
    unlock(&mutex);                        // release mutex
}
```

- `unlock()` never gets executed; mutex is held forever
- `bgImage` points to a deleted object
- `imageChanges` has been incremented

Exception-save function guarantees

- basic guarantee
  - if an exception is thrown, everything in the program remains in a valid state
  - exact state of the program may not be predictable
- strong guarantee
  - if an exception is thrown, the state of the program is unchanged
  - such functions are *atomic,* either they succeed completely or the program state is as if they‘d never been called
- nothrow guarantee
  - always doing what promised to do
  - all operations on built-in types (ints, pointers, etc.) are nothrow

Improving changeBackground( )

- Using resource management classes

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
    Lock ml(&mutex);                  // acquire mutex
                                      // and asure its later release
    delete bgImage;                   // get rid of old background
    ++imageChanges;                   // update image change count
    bgImage = new Image(imgSrc);      // install new background
    unlock(&mutex);
}
```

```
class Lock {
public:
 explicit Lock(Mutex * pm):mutexPtr(pm)
 {lock(mutexPtr);}

 ~Lock(){unlock(mutexPtr);}

private
 Mutex* mutexPtr;
};
```

Improving changeBackground( )

- Using smart pointer and reorder statements

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
    Lock ml(&mutex);                    // acquire mutex
                                        // and asure its later release

    bgImage.reset(new Image(imgSrc));   // replace bgImage's
                                        // internal pointer
                                        // with the result of
                                        // the „new Image"
                                        // expression

    ++imageChanges;                     // update image change count

}
```

```
class PrettyMenu
{
 ...
 std::tr1::shared_ptr<Image> bgImage;
 ...
};
```

Results

After ...

- using resource management classes
- using smart pointer
- and reordering statements

... changeBackground( ) can *almost* offer the **strong exception safety guarantee**.

Results

Why just almost ?

- the problem is the parameter `imgSrc`
- if the `Image` constructor throws an exception it‘s possible that the read marker for the imput stream has been moved; such movement would be a change in state

- Solution → **„copy and swap“**
  - make a copy of the object to be modified
  - make all needed changes to the copy
  - after successfully completed changes, swap the modified object with the original in a non-throwing operation

Things to Remember

- Exception-safe functions leak no resources and allow no data structures to become corrupted, even when exceptions are thrown. Such functions offer the **basic**, **strong**, or **nothrow guarantees**.
- The **strong guarantee** can often be implemented via copy-and-swap, but the strong guarantee is not practical for all functions.
- A function can usually offer a **guarantee** no stronger than the weakest guarantee of the functions it calls.

Further information

- Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*