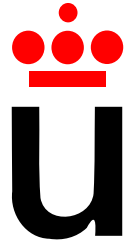
8. Manejo de excepciones

Julio Vega

julio.vega@urjc.es



Universidad
Rey Juan Carlos

Contenidos

1. Introducción
2. Manejo básico de una excepción
3. Ejemplo básico de manejo de excepción: la división por cero
4. Ejemplo de relanzamiento de una excepción
5. Ejemplo de excepción `bad_alloc` al usar `new`

- Una excepción es la indicación de un problema dado en ejecución.
- El nombre ya implica que el problema ocurre con poca frecuencia.
- El manejo de excepciones permite crear rutinas para resolverlas.
  - En muchos casos permite continuar la ejecución del programa.
- El manejo de excepciones es ideal para procesar errores síncronos.
  - Aquellos que ocurren cuando se ejecuta una determinada instrucción.
  - E.g. arrays fuera de rango, desbordamiento aritmétrico, división por 0.
- El manejo de excepciones no es para trabajar con eventos asíncronos.
  - Aquellos que ocurren en paralelo a la ejecución del programa.
  - E.g. completar I/O disco, recepción mensajes red, I/O teclado/ratón.

- La clase `exception` es la clase base estándar de C++.
  - Recoge todo tipo de excepciones y está definida en `<exception>`.
  - Ofrece la función virtual `what`, que devuelve mensaje de error.
- Una clase derivada de `exception` es `runtime_error`.
  - Clase base estándar de C++ para errores en tiempo de ejecución.
  - Esta clase está definida en el *header* `<stdexcept>`.
- Para definir excepción propia, lo más cómodo es mediante herencia.
  - De este modo, tenemos acceso a la función `what`.
  - E.g. heredando de la clase `runtime_error`.
  - Lo más sencillo es definir un constructor con el mensaje de error.

- La excepción se puede lanzar manualmente mediante `throw`.
  - E.g. cuando al intentar hacer división, el denominador = 0.
    - `if (den == 0) throw DivisionException();`
- La instrucción `throw` (o la función que la contiene) deberá.
  - Estar contenida en un bloque de *posible excepción*: bloque `try`.
  - Este va seguido de otro bloque, el `catch`, para su tratamiento.
- Un programa puede seguir su ejecución tras tratar la excepción.
- Al terminar manejador `catch` el programa sigue tras último `catch`.
  - No se vuelve al punto en que ocurrió la excepción (punto lanzamiento).
  - También se seguiría en tal punto si no hubiese excepción en el `try`.

```
try {
  result = getDivision (n1, n2);
  cout << "The getDivision result is: " << result << endl;
} catch (DivisionException &except) {
  cout << "Exception: " << except.what() << endl;
}
```

- El bloque `try` incluye instrucciones que podrían causar excepción.
  - Además de las instrucciones que se omitirían si hubiera tal excepción.
- El bloque `catch` es el encargado de atrapar y manejar la excepción.
  - Manejará la excepción del tipo según parámetro dado entre paréntesis.
    - O si la excepción es de un tipo derivado del especificado por parámetro.
  - Es más eficiente atrapar un objeto excepción por referencia.
    - Se evita la sobrecarga de copiar el objeto de la excepción lanzada.
  - Puede haber varios `catch`; entonces solo se ejecuta el del tipo dado.
    - Si no hay un `catch` que coincida, la ejecución se termina inmediata/.
  - Es un error sintáctico poner más de un parámetro a un bloque `catch`.
- Sería un error sintáctico implementar código entre ambos bloques.

```
#include <stdexcept> // contains runtime_error

class DivisionException : public std::runtime_error {
public:
  DivisionException()
    : std::runtime_error ("trying to divide by zero!") {}
};
```

```
#include <iostream>
#include "DivisionException.h"
using namespace std;

// get division but throw divisionException if divide by zero
double getDivision (int num, int den) {
  if (den == 0) throw DivisionException(); // function is ended

  // otherwise:
  return static_cast<double> (num) / den;
} // end function getDivision
```

```
int main() {
  int n1, n2;
  double result;

  cout << "Please, type two integers:\n";

  while (cin >> n1 >> n2) {
    try {
      result = getDivision (n1, n2);
      cout << "The getDivision result is: " << result << endl;
    } catch (DivisionException &except) {
      cout << "Exception: " << except.what() << endl;
    }

    cout << "Please, type two integers:\n";
  }
  cout << endl;
}
```

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

void rethrowException() {
  try {
    cout << "rethrowException(): Throwing an exception\n";
    throw exception();
  } catch (exception &except) {
    cout << "rethrowException(): Exception caught & rethrown\n";
    throw; // exception is raised!
  }
  cout << "rethrowException(): After caught & rethrown\n";
}
```

```
int main() {
  try {
    cout << "main(): Calling rethrowException()\n";
    rethrowException();
    cout << "main(): After calling rethrowException()\n";
  } catch (exception &except) {
    cout << "main(): Exception caught\n";
  }
  cout << "main(): After caught exception()\n";
}
```

- Estándar de C++: si `new` falla → lanzar excep. `bad_alloc`.
  - Esta excepción está definida en el *header* `<new>`.
- Algunos compiladores no están conformes con esta especificación.
  - E.g. *Microsoft Visual C++ 6.0* devuelve 0 al fallar el `new`.
  - Otros, lanzan `bad_alloc` con o sin incluir la cabecera `<new>`.

```
#include <iostream>
#include <new>
using namespace std;

int main() {
  double *p[100];

  try {
    // size_t is widely used for counts
    // it represents size of any objet in bytes
    for (size_t i = 0; i < 100; ++i) { // each p->(huge memory)
      p[i] = new double[999999999]; // may cause exception!
      cout << "p[" << i << "] points to 999.999.999 doubles\n";
    }
  } catch (bad_alloc &except) {
    cerr << "Exception!: " << except.what() << endl;
  }
}
```

8. Manejo de excepciones

Julio Vega

julio.vega@urjc.es