TEMA 4

GESTIÓN DE ERRORES

Cristina Cachero, Pedro J. Ponce de León

1 Sesión (1.5 horas)

Versión 0.6

Gestión Errores
Objetivos

- **Saber utilizar `try`, `throw` y `catch` para observar, indicar y manejar excepciones, respectivamente.**
- **Comprender las ventajas del manejo de errores mediante excepciones frente a la gestión de errores tradicional de la programación imperativa.**
- **Comprender la jerarquía de excepciones estándar.**
- **Ser capaz de crear excepciones personalizadas**
- **Ser capaz de procesar las excepciones no atrapadas y las inesperadas.**

Gestión de Errores
Motivación

- Gestión de errores 'tradicional' (o al estilo de C)

```
int main (void)
{
  int res;
  if (puedo_fallar () == -1)
    {
      cout << "¡Algo falló!" << endl;
      return 1;
    }
  else cout << "Todo va bien..." << endl;
  if (dividir (10, 0, res) == -1)
    {
      cout << "¡División por cero!" << endl;
      return 2;
    }
  else cout << "Resultado: " << res << endl;
  return 0;
}
```

Flujo normal
Flujo de error

Gestión de Errores
Motivación

- Nos obliga a definir un esquema de programación similar a :
  - Llevar a cabo tarea 1
    - *Si se produce error*
      - *Llevar a cabo procesamiento de errores*
  - Llevar a cabo tarea 2
    - *Si se produce error*
      - *Llevar a cabo procesamiento de errores*

- A esto se le llama código ***espaguetti***

- <u>Problemas de esta estrategia</u>
  - Entremezcla la lógica del programa con la del tratamiento de errores (disminuye legibilidad)
  - Puede degradar el rendimiento del sistema (a veces, el código de tratamiento de errores consta de más instrucciones que el código normal)
  - El código cliente (llamador) no está obligado a tratar el error
  - Los 'códigos de error' no son consistentes.

Gestión de Errores
Motivación

- ¿Qué podemos hacer en lugar de crear código *spaguetti*?
  - Abortar el programa
    - ¿Y si el programa es crítico?
  - Usar indicadores de error globales
    - El código cliente (llamador) no está obligado a consultar dichos indicadores.
  - USAR EXCEPCIONES
    - Esto SÍ que mola...

Gestión de Errores
Excepciones: **Concepto**

- Una **excepción** es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias.

- Son objetos (instancias de clases) que contienen información sobre el error
  - Un mismo método puede notificar diferentes tipos de errores.

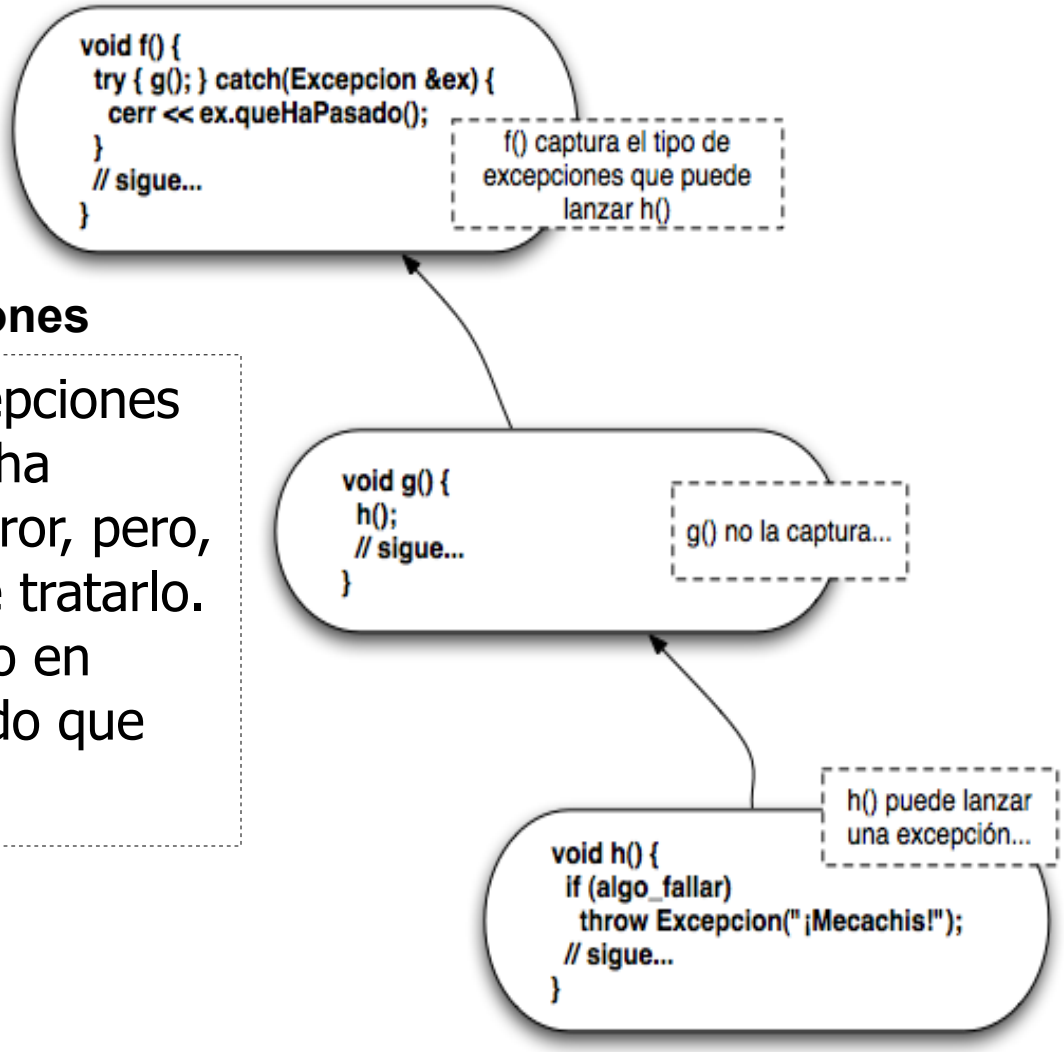
- Las excepciones se tratan mediante sentencias de control del flujo de error que separan el código para manejar errores del resto mediante :
  - **`throw`**, **`try`** y **`catch`**

- Por defecto, una excepción no se puede ignorar: hará que el programa aborte.
  - Una excepción pasará sucesivamente de un método a su llamador hasta encontrar un bloque de código que la capture.

Gestión de Errores
Excepciones: **Comportamiento**

- Las excepciones son **lanzadas** (`throw`) por un método cuando éste detecta una condición excepcional o de error.
- Esto interrumpe el control normal de flujo y provoca (si el propio método no la captura) la finalización prematura de la ejecución del método y su retorno al llamador.
- Si el llamador no captura la excepción, su ejecución terminará y la excepción 'saldrá' de nuevo hacia un ámbito más externo y así sucesivamente hasta encontrar un lugar donde es capturada
- Una excepción no capturada provocará que el programa aborte.

Gestión de Errores
Excepciones: **Uso correcto**



Uso correcto de las excepciones

- Un método que lanza excepciones se limita a señalar que se ha producido algún tipo de error, pero, por regla general, no debe tratarlo. Delegará dicho tratamiento en quienes invoquen al método que produce la excepción.

Gestión de Errores
Excepciones: **Sintaxis C++**

```
try
{
  // Codigo de ejecución normal
  Func(); // puede lanzar excepciones
  ...
}
catch (Tipo1 &ex)
{
// Gestión de excep tipo 1
}
catch (Tipo2 &ex)
{
// Gestión de excep tipo 2
}
catch (...)
{
/* Gestión de cualquier excepción no
    capturada mediante los catch
    anteriores*/
}
//Continuación del código
```

```
void Func() {

  if (detecto_error1) throw Tipo1();
  …
  if (detecto_error2) throw Tipo2();
  ...

}
```

Gestión de Errores
Excepciones: **Sintaxis Java**

```
try
{
  // Codigo de ejecución normal
  Func(); // puede lanzar excepciones
}
catch (Tipo1 ex)
{
// Gestión de excep tipo 1
}
catch (Tipo2 ex)
{
// Gestión de excep tipo 2
}
finally {
  // se ejecuta siempre
}
```

```
void Func() {

  if (detecto_error1) throw new Tipo1();
  …
  if (detecto_error2) throw new Tipo2();
  ...

}
```

Gestión de Errores
Excepciones: **Sintaxis**

- En C++ y en Java:
  - El bloque **try** contiene el código que forma parte del funcionamiento normal del programa
  - El bloque **catch** contiene el código que gestiona los diversos errores que se puedan producir

- Sólo en JAVA:
  - El bloque **finally** de Java proporciona un mecanismo que permite a sus métodos limpiarse a sí mismos sin importar lo que sucede dentro del bloque **try**. Se utiliza el bloque **finally** para cerrar ficheros o liberar otros recursos del sistema. El bloque finally se puede ejecutar:

    (1) tras finalizar el bloque try o

    (2) después de las cláusulas catch.

Gestión de Errores
Excepciones: **Funcionamiento**

- Funcionamiento:
  1. Ejecutar instrucciones try
     1. Si hay error, interrumpir el bloque try e ir a bloque catch correspondiente
  2. Continuar la ejecución después de los bloques catch

- La excepción es capturada por el bloque **catch** cuyo argumento coincida con el tipo de objeto lanzado por la sentencia **throw**. La búsqueda de coincidencia se realiza sucesivamente sobre los bloques **catch** en el orden en que aparecen en el código hasta que aparece la primera concordancia.
  - Implica que **el orden de colocación de los bloques catch es determinante**. Por ejemplo: si se incluye un manejador universal, éste debería ser el último.

- En caso de no existir un manejador adecuado a una excepción determinada, se desencadena un protocolo que, por defecto, produce sin más la finalización del programa. En C++, es una llamada a `std::terminate()`

Gestión de Errores
Excepciones: **Lanzamiento**

- La cláusula **`throw`** lanza una excepción
  - Una excepción puede ser cualquier cosa: int, float, string,... o cualquier objeto.

Lanzamiento

```
int LlamameConCuidado(int x) {
    if (condicion_de_error(x) == true)
      throw x;
    //... código a ejecutar si no hay error ...
  }
```

Captura

```
int main() {
  try {
    LlamameConCuidado(-0);
  } catch (int ex) {
    cerr << "No tuviste cuidado: " << ex << endl;
  }
}
```

Gestión de Errores
Lanzamiento (excepciones de usuario)

- Es habitual tipificar el error creando clases de objetos que representan diferentes circunstancias de error.
- La ventaja de hacerlo así es que
  - Podemos incluir información extra al lanzar la excepción.
  - Podemos agrupar las excepciones en jerarquías de clases.

```
class miExcepcion {
     int x; string msg;
   public:
    miExcepcion(int a, string m) : x(a), msg(m) {}
    string queHaPasado() const {return msg; }
    int getElCulpable() const { return x; }
 };
```

Gestión de Errores
Excepciones: **Captura**

- La instrucción catch es como una llamada a función: recibe un argumento.

```
int LlamameConCuidado(int x) {
    if (condicion_de_error(x) == true)
      throw miExcepcion(x,"¡Lo has vuelto a hacer!");
    //... código a ejecutar si no hay error ...
  }
```

Llamada a constructor

```
int main() {
   try {
     LlamameConCuidado(-0);
   } catch (miExcepcion& ex) {
     cerr << ex.queHaPasado()
          << ex.getElCulpable() << endl;
   }
}
```

Argumento por referencia

Gestión de Errores
Excepciones: **Captura**

```
int main() {
  try {
    LlamameConCuidado(-0);
  } catch (miExcepcion& ex) {
    cerr << ex.queHaPasado()
         << ex.getElCulpable() << endl;
  }
}
```

Argumento por referencia

Al capturar la excepción por referencia y agrupar las excepciones en jerarquías de herencia, se puede usar el principio de sustitución:

```
class festival {};
class Verano : public festival {};
class Primavera: public festival {};
```

Gestión de Errores
Excepciones: **Excepciones derivadas**

```
void fiesta(int i) {
     if (i==1) throw(Verano());
     else if (i==2) throw (Primavera());
     else throw(festival() );
}

int main(){
   int f=...;
   try { fiesta(f); } // bloques catch en el orden adecuado
     catch(const Verano& )  { cerr << "Festival de Verano"; }
     catch(const Primavera&){ cerr << "Festival de Primavera" ; }
     catch(const festival& ){ cerr << "Festival" ; }

/* Si se captura la clase base primero se pierde la posibilidad de
comprobar la clase derivada de la excepción que ha sido lanzada
realmente */

   try { fiesta(f); }
    catch(const festival& ){ cerr << "Festival (de que tipo??!!)"; }
    catch(const Verano& )  { cerr << "Festival de Verano" ; }
    catch(const Primavera&){ cerr << "Festival de Primavera" ; }

}
```

Gestión de Errores
Excepciones: **especificación de excepción**

- En C++ un método puede indicar, en su declaracion, que excepciones puede lanzar (directa o indirectamente) mediante la **especificación de excepción.**

- Este especificador se utiliza en forma de sufijo en la declaración de la función y tiene la siguiente **sintaxis**:

```
throw (<lista-de-tipos>)  // lista-de-tipos es opcional
```

- `<lista-de-tipos>` indica que el método puede lanzar únicamente esas excepciones.

```
int f() const throw(E1, E2);
```

`f()` sólo puede lanzar excepciones de tipo E1 o E2.

Gestión de Errores
Excepciones: **especificación de excepción**

- La ausencia de especificador indica que la función puede lanzar cualquier excepción:

```
int f() const; // puede lanzar cualquier excepción
```

- Un especificador vacío indica que el método no lanza excepciones:

```
int f() const throw(); // no lanzará ninguna excepción
```

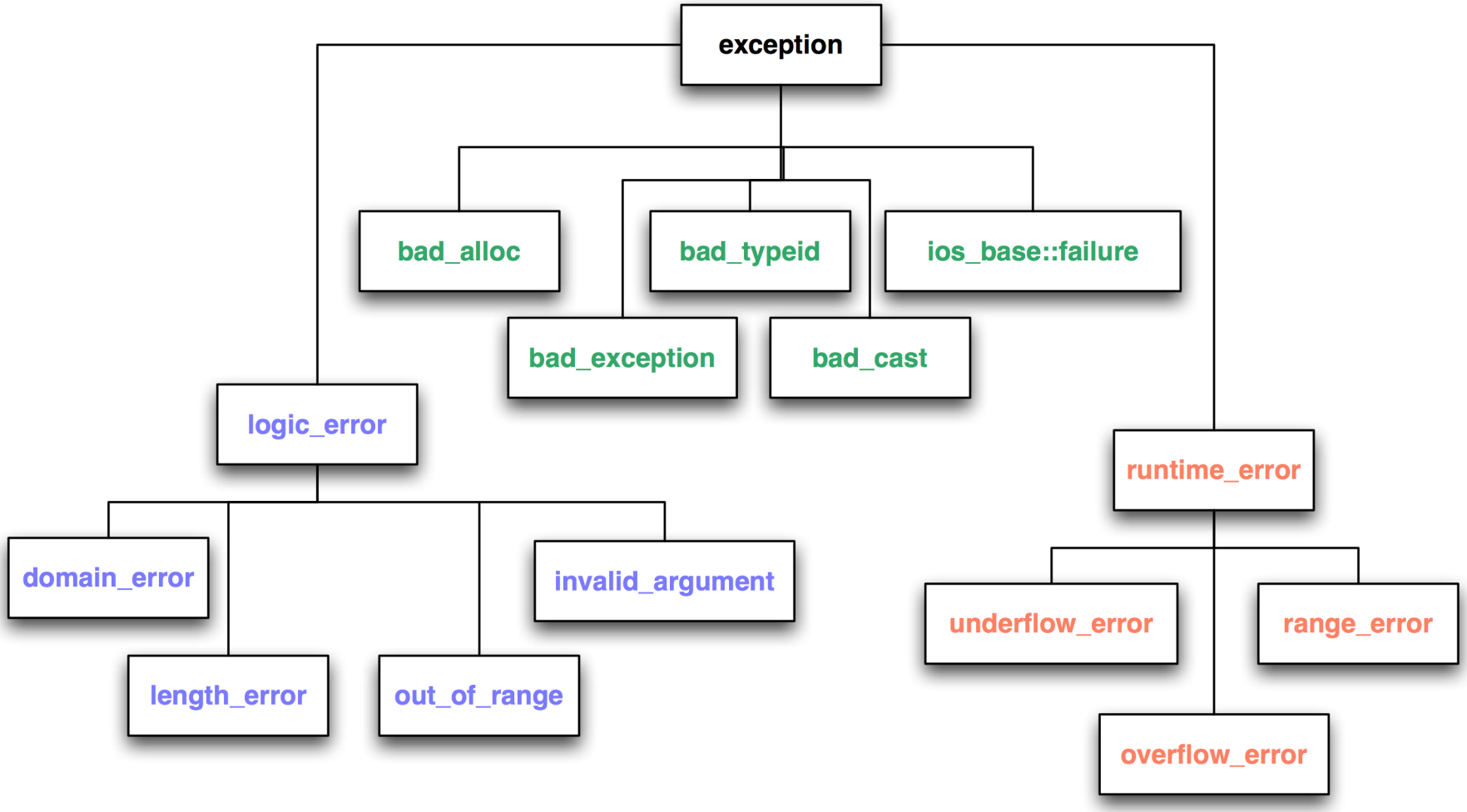
Gestión de Errores
Excepciones estándares en C++

- Todas las excepciones lanzadas por componentes de la STL de C++ son excepciones derivada de la clase base **`exception`**, definida en la cabecera <**`exception`**>, que tiene la siguiente interfaz:

```
class exception {
    public:
    exception () throw();
    exception (const exception&) throw();
    exception& operator= (const exception&) throw();
    virtual ~exception () throw();
    virtual const char* what () const throw();
  };
```

  - Proporciona un método virtual **`what()`,** que devuelve una cadena con un mensaje verbal (dependiente del compilador que estéis utilizando) que refleja el tipo concreto de excepción.
    - Este mensaje puede ser sobrescrito en clases derivadas para contener una descripción personalizada de la excepción.

Gestión de Errores
Excepciones estándares en C++



Gestión de Errores
Excepciones estándares en C++

- **bad_alloc**: lanzada por el operador **`new`** si hay un error de reserva de memoria
- **bad_cast**: lanzada por el operador **`dynamic_cast`** si no puede manejar un tipo referenciado
- **bad_exception**: excepción genérica lanzada cuando un tipo de excepción no está permitida en una función determinada
  - El tipo de excepciones permitidas se especifica con la cláusula `throw()`.
- **bad_typeid**: lanzado por el operador **`typeid`** a una expresión nula
- **ios_base::failure**: lanzado por las funciones en la librería ***iostream***

Gestión de Errores
Excepciones estándares en C++

- Ubicación de las excepciones:

excepcion	fich. cabecera
`bad_alloc`	`<new>`
`bad_cast`	`<typeinfo>`
`bad_typeid`	`<typeinfo>`
`bad_exception`	`<exception>`
`logic_error (y deriv.)`	`<stdexcept>`
`runtime_error (y deriv.)`	`<stdexcept>`
`ios_base::failure`	`<ios>`

Gestión de Errores
Excepciones estándares en C++: reserva memoria

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

int main()
{
   double *ptr[50];
   try {
        for (int i= 0 ; i < 50; i++) {
          ptr[i] = new double[50000000];
         cout << "Reservando memoria para elemento " << i << endl;
        }
   }
   catch (bad_alloc &ex){
      cout << ex.what() << endl;
   }
   cout<<"Termino programa normalmente"<<endl;
   return (0);
}
```

Gestión de Errores
Excepciones estándares en C++: otros errores

```
int main () {
   try
   {
      /* ERROR ENTRADA/SALIDA CIN*/
       cin.exceptions(ios::failbit);
      cout << "Mete lo primero que se te ocurra, distinto de entero: " << endl;
      int entero; cin >> entero;
     /* ERROR SEGMENTATION FAULT: NO ES CAPTURABLE SALVO QUE TRABAJEMOS CON STL
      char *p=NULL;
      cout<<*p<<endl; */
   }
   catch (ios_base::failure &ex){
       cout<<"Error de I/O, mensaje: "<<ex.what()<<endl;
   }
   catch (std::exception& stdexc)
   {
      cout << "Error general, mensaje: " << stdexc.what() << endl;
   }
   catch (...)
   {
      cout << "Error general no derivado de exception" << endl;
   }
   cout<<"Termino el programa normalmente"<<endl;
   return (0);
}
```

Gestión de Errores
Excepciones de Usuario en C++

- EJERCICIO
  - **Definid una excepción de usuario llamada** `ExcepcionDividirPorCero` que sea lanzada por la siguiente función al intentar dividir por cero:
    - `float div(float x, float y) { return x/y; }`
  - Escribe un programa que invoque a div() y trate correctamente la excepción.

Gestión de Errores
Excepciones de Usuario en C++

- SOLUCIÓN:

```
class ExcepcionDividirPorCero : public exception {
  public :
      ExcepcionDividirPorCero() : exception() {}
      const char * what() const throw() {
        return "Intentas dividir por cero”; }
};
```

Gestión de Errores
Excepciones de Usuario en C++

```
int main()
{
   float dividendo, divisor, resultado;
   cout << "PROGRAMA DIVISOR" << endl;
   cout << "Introduce Dividendo : " ;
   cin >> dividendo;
   cout << "Introduce Divisor : " ;
   cin >> divisor;
   try {
         resultado = div(dividendo, divisor);
         cout << dividendo << "/" << divisor
              << "=" << resultado;
   }
   catch (ExcepcionDividirPorCero &exce)
   {
      cerr << "Error:" << exce.what() << endl;
   }
   return (0);
}
```

¿Sería correcto colocar esta instrucción detrás del bloque catch?

Gestión de Errores
Excepciones: Eficiencia

- El mecanismo de manejo de excepciones exige almacenar información adicional sobre el objeto de excepción y cada sentencia catch, con el fin de realizar la concordancia en tiempo de ejecución (ya que puede existir polimorfismo) entre la excepción y su manejador. Además, también es necesario guardar información sobre la estructura de cada función, para poder determinar si una excepción fue lanzada desde un bloque try.

- Esto supone una sobrecarga adicional en términos de velocidad y tamaño de programa, incluso cuando no se lanza nunca una excepción.

- Ojo! Incluso si no se usan excepciones directamente, probablemente se estén usando implícitamente. Por ejemplo, los contenedores STL suelen lanzar sus propias excepciones (p.ej. la función at() lanza un out_of_range si se intenta acceder fuera de los límites del contenedor). También otras funciones de la librería estándar (p.ej. funciones de la clase string) lanzan excepciones.

Gestión de Errores
Excepciones: Malos usos de excepciones

- Algunos programadores usan (mal) el manejo de excepciones como una alternativa a bucles for, bloques do-while o simples 'if'.
- Ejemplo:

```
#include <iostream>
using namespace std;
class Exit{}; //los objetos exit son usados como una excepción
int main()
{
 int num;
 cout<< "Introduce un número (99 para salir)" <<endl;
 try
 {
  while (true) // bucle infinito
  {
   cin>>num;
   if (num == 99)
     throw Exit(); //salir del bucle
   cout<< "introdujiste: " << num << ". Introduce otro número " <<endl;
  }
 }
 catch (Exit& )
 {
  cout<< "Fin!" <<endl;
 }
}
```

Gestión de Errores
Excepciones: Resumen

- **Ventajas:**
  - **Separar el manejo de errores del código normal**:
    - El manejo de excepciones permite al programador quitar el código para manejar errores de la línea principal.
    - Permite escribir programas más claros , robustos y tolerantes a fallos.
  - Agrupar los tipos de errores y la diferenciación entre ellos
  - Detectar el error en un lugar (código 'servidor') y tratarlo en otro diferente (código cliente).
  - Obliga al código cliente a tratar (o ignorar) expresamente las condiciones de error.
- **Inconvenientes**
  - Sobrecarga del sistema

Gestión de Errores
Excepciones: Ejercicio propuesto

- **Ejercicio**
  - Define una clase PilaEnteros que gestione una pila de enteros. La pila se crea con una capacidad máxima. Queremos que la pila controle su desbordamiento al intentar apilar más elementos de los que admite, mediante una excepción de usuario ExcepcionDesbordamiento. Define en C++ la clase PilaEnteros y los métodos Pila(), ~Pila() y apilar().
  - Implementa un programa principal de prueba que intente apilar más elementos de los permitidis y capture la excepción en cuanto se produzca (dejando de intentar apilar el resto de elementos).

Gestión de Errores
Excepciones: Ejercicio propuesto

- **Solución**

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

class ExcepcionDesbordamiento : public exception {};


class PilaEnteros{
    public:
        Pila(int max=10);
        void apilar (int);
        int desapilar();
        ~Pila();

    private:
        int maximo; // Tamaño máximo de la pila
        Vector<int> pila; // la pila
        int cima; //número de elementos actualmente en la pila-1
};
```

Gestión de Errores
Excepciones: Ejercicio propuesto

- **Solución**

```
PilaEnteros::PilaEnteros(int max) : maximo(max), pila(max), cima(-1)
{ if (maximo<0) maximo=0; }

PilaEnteros::~Pila(){
   pila.clear();
   maximo=-1;
   cima=-1;
}
```

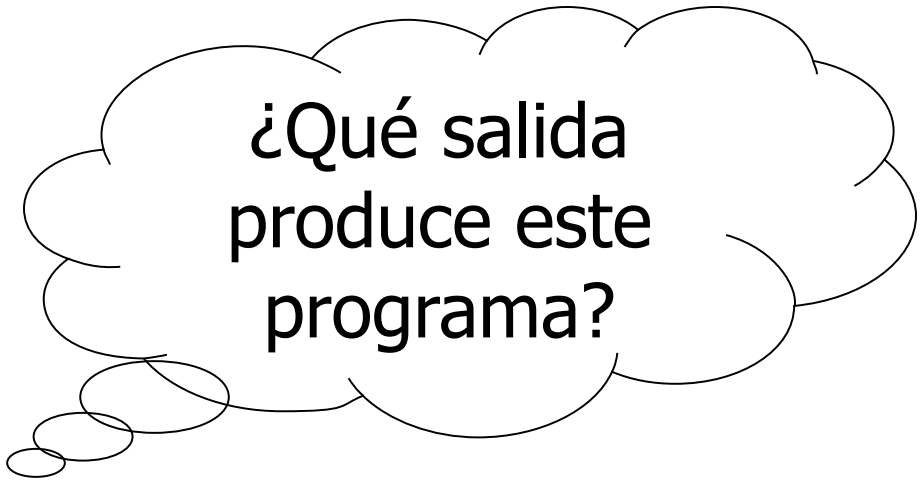
Gestión de Errores
Excepciones: Ejercicio propuesto

- **Solución**

```
void PilaEnteros::apilar (int elem){
   cout<<"Llamo método apilar"<<endl;
   try {
      if (cima<maximo-1) {
         pila[++cima]=elem;
         cout<<"Apilo"<<endl;
      } else
         throw ExcepcionDesbordamiento();
   }
   catch(exception& e){
        cout<<"Catch método apilar"<<endl;
   }
}

int main(){

    PilaEnteros pi(2);
    for (int i=0;i<20;i++)
       pi.apilar(i);
    return (0);
};
```



Gestión de Errores
Excepciones: Ejercicio propuesto

- **Solución**

```
int main(){

    PilaEnteros pi(2);
    try{
      for (int i=0;i<20;i++)
        pi.apilar(i);
    }
     catch(exception &e){
       cerr<<"Catch de main: tipo error exception"<<endl;

    }
    catch(...){
       cerr<<"Catch de main: tipo error desconocido"<<endl;
    }
    return (0);
};
```

Gestión de Errores
Excepciones: Ejercicio propuesto

- **Alternativa: no capturar excepción en Apilar**
  - En este caso es obligatorio que el main capture la excepción si no queremos que el programa termine de manera inesperada

```
void PilaEnteros::apilar (int elem){
      cout<<"Intento apilar"<<endl;
      if (cima<maximo-1)
         pila[++cima]=elem;
      else
        throw ExcepcionDesbordamiento();
};

int main(){
    PilaEnteros pi(2);
    try{
       for (int i=0;i<20;i++) pi.apilar(i);
    }
    catch(exception &e){
       cerr<<"Catch de main: tipo error exception"<<endl;
    }
    catch(...){
       cerr<<"Catch de main: tipo error desconocido"<<endl;
    }
}
```

Gestión de Errores
Excepciones: **Relanzar una excepción**

- **¿Cómo lanzar una excepción capturada en un nivel de anidamiento X hacia los niveles superiores?**

```
void PilaEnteros::apilar (int elem){
  try{
      cout<<"Intento apilar"<<endl;

      try{
      if (cima<maximo){
         elementos[++cima]=elem;
         cout<<"Elemento apilado"<<endl;
      }
      else
        throw ExcepcionDesbordamiento();
      }
      catch(...){
         cerr<<"Catch interno de apilar"<<endl;
         throw; // relanzamiento
      }
   }
   catch(...){
        cerr<<"Catch externo de apilar"<<endl;

   }
}
```

```
int main(){
     PilaEnteros pi(2);
     try{
     for (int i=0;i<5;i++)
        pi.apilar(i);
     }
     catch(...){
         cerr<<"Catch del main"<<endl;

     }
     return (0);
};
```