Exceptionhandling in C++

Types of errors and/or exceptions

- Syntax error: the compiler errors that result from incorrect coding.
- Logical error: are errors that ends up in incorrect results. For example incorrect algorithm for a calculation. The syntax is correct in this case. Such errors can be found and processed only during program testing and/or design review.

Types of errors and/or exceptions

- Runtime error/exception: are anomalies / irregularities that occur during execution of a program, and can be caused by eg. denied memory access of different types, division by zero, and / or numerical overflow!.
- These exceptions are not so normal but somewhat predictable.

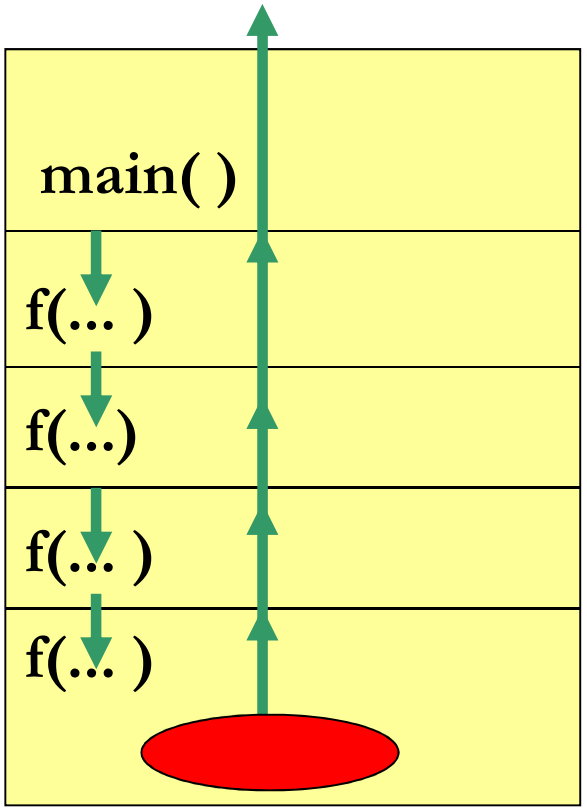
How do we handle errors and exceptions

- Don’t manage them at all!: results in Core dump, undesirable.
- Create an error message and exit (): perhaps from the same place where the exception / error occurred. if(syre-apparaten dividerade med 0) exit(-1);
  - assert(...) , return(a_non_valid_value), return(a_valid_value) and leave the program in an unknown / incorrect state!
- Manage exception "gracefully": and maybe continue execution.

ASSERT( ) macro

```
#include <cassert>
void rangeValidation(int range)
{
   / / if true, nothing happens, otherwise a
   message is written out and exits with exit (0);
    assert (range >= 0 && range < 10);
}
```

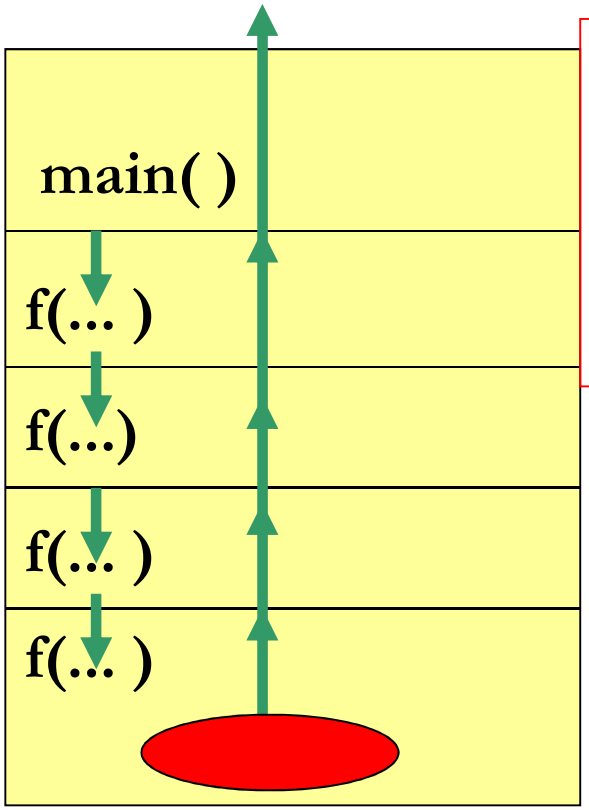
"Problems" with errors and/ exception handling



The Stack

- Routine and/or blocks of code that detects a fault often do not know what to do, and the calling routine normally know what to do do not know that the error has occurred.
- We need to solve that!....

In C and/or C++, we can solve the "problem" with error-handling!



The Stack

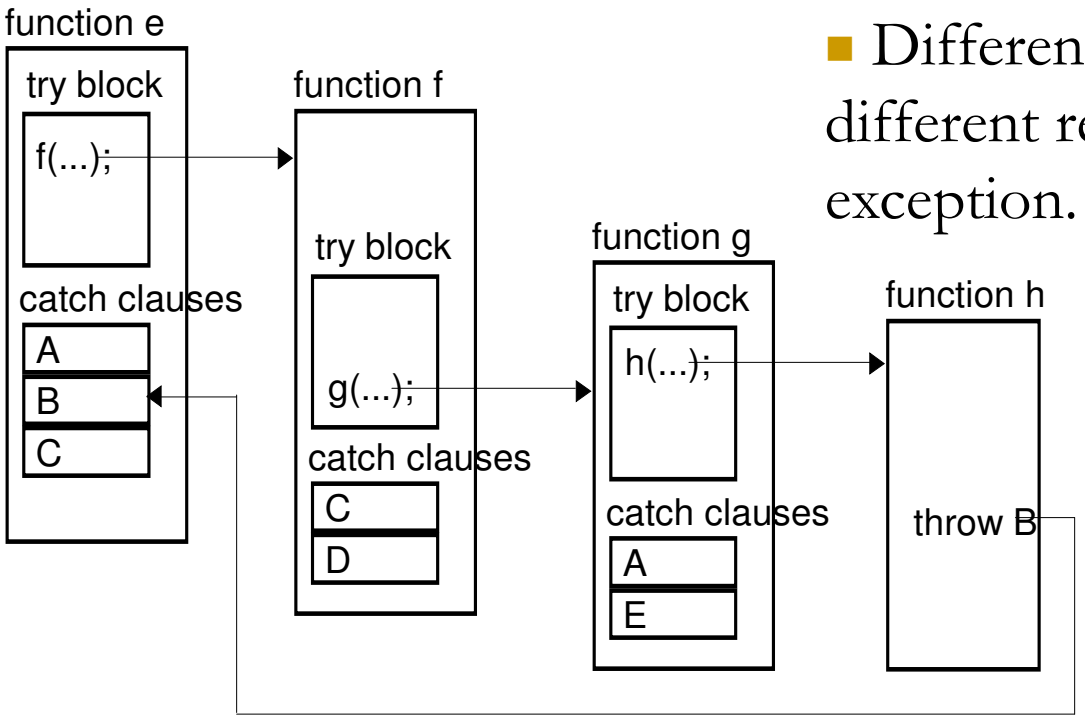
```
.....
if (rc = f(...) ) {
   //Aj då, något gick
      fel, hantera det.
 return rc;
}
```

```
int f(...)  {
   ....
 if (something wrong)
   return -1;
 else
   return 0;
}
```

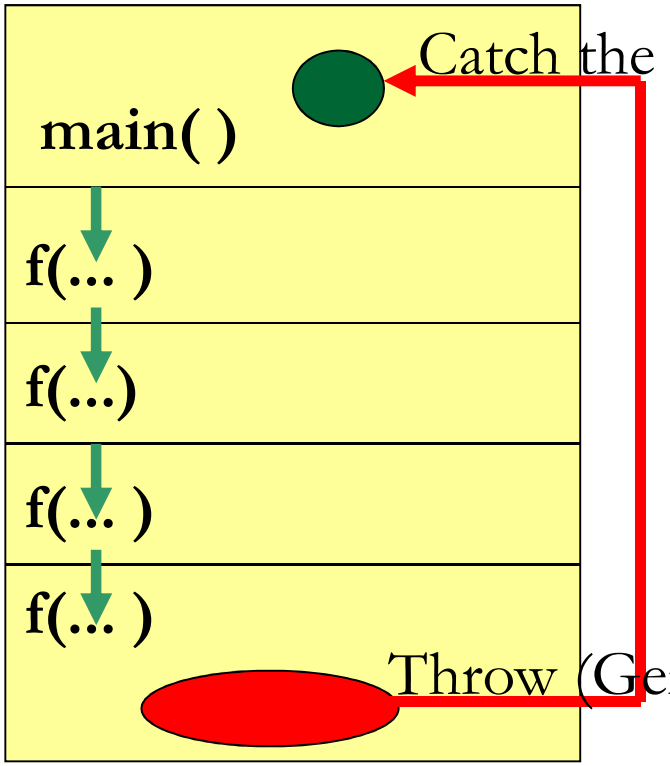
1) Return value is used only for error and nothing else.
2) Must always check for error even if the error value is only passed on to the next level

In C++, we solve the "problem" with exceptions

- The calling routine knows better about how the exception should be handled.
- Different procedures might give different responses to a specific exception.



In C++, we solve the "problem" with exceptions



Catch the exception, when it generated

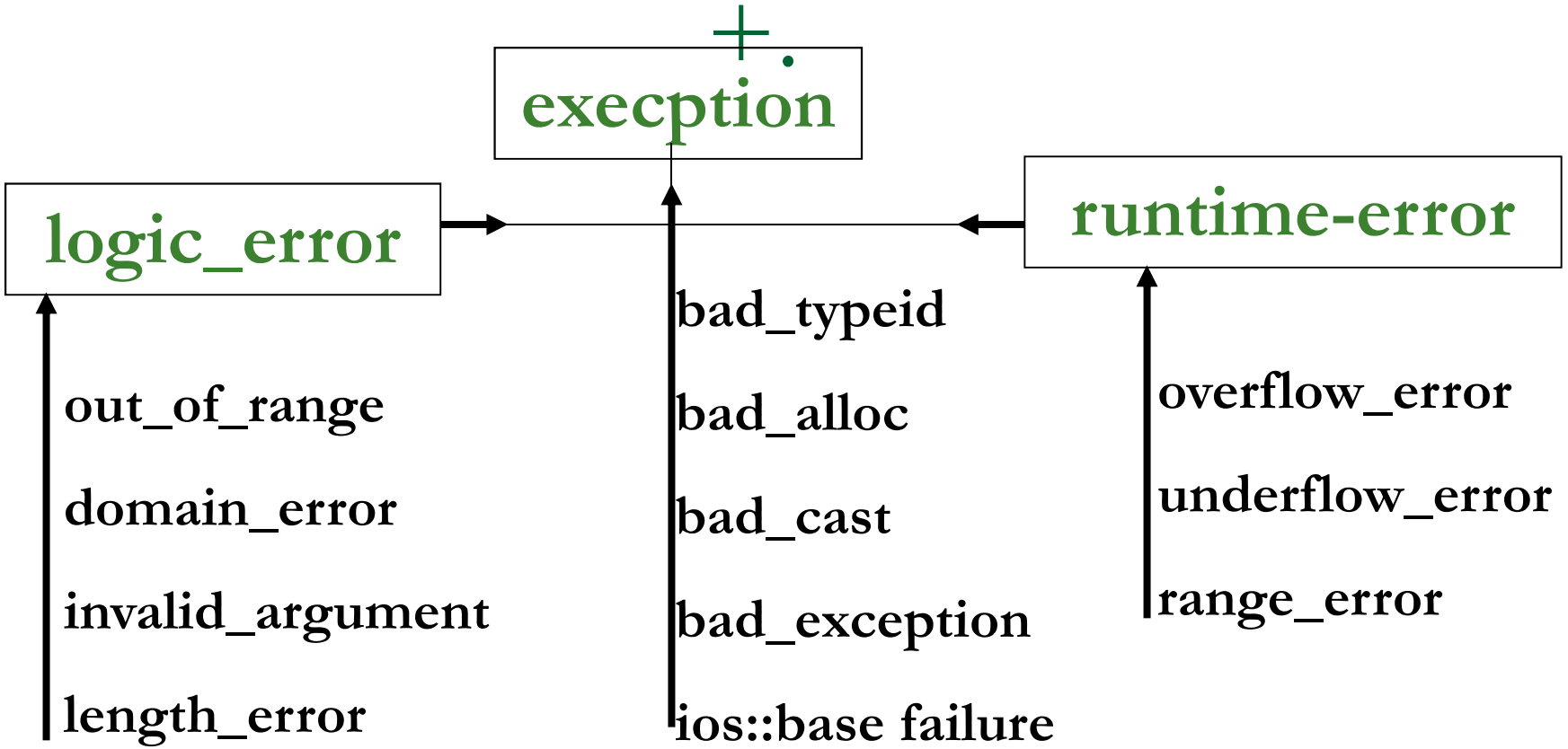
Throw (Generate) the exception, when it occurs

The Stack

Exceptions in C++

- Is a mechanism for transferring control to another point/ routine other than where the exception occurred.
- Are predefined / primitive variables or class object
- The routine that discovers, GENERATES ("throws") an exception. The exception is the object being thrown (sent back).
- All types of objects (int, etc.), although class objects, can be sent back to another routine that takes care of the 'catch' exception.
- Exception object contains enough information about the exception.

Standard (exception) library in C++



bad_exception is in <exception> , ios::base failure in <ios> ,
bad_typeid in <typeinfo> and the rest in <stdexcept>

Generell syntax for try resp. catch

```
try {
// try block
}
catch (typ1 arg) {
// catch block
}
catch (typ2 arg) {
// catch block
}
....
catch (typN arg) {
// catch block
}
```

1. Code that may cause an exception is surrounded by a try block. Functions called within try block can also generate an exception.
2. Try block may contain a few number of statements at any time, or even the entire main function!
3. Exceptions generated, captured by one or more catch blocks directly following (not necessarily so) try block. Catch block handles the exception type. ie. the data type specified in the catch-block arguments and is the same as the exception object-type.

General syntax for throw

throw exception;

1. throw generates the expected exception.
2. If this throw to be caught, the throw statement must be executed either inside a try block or another function that is called from the try block, directly or indirectly
3. As default will terminate()→abort() function will be called if no matching catch is found. One can also define your own terminate_handler.

Remember:

When an exception is thrown, the execution of the program continues AFTER the try-block, NOT after the throw statement. See examples in comming slides

Example 1: run nr. 1

```
#include <iostream>
using namespace std;
int main()  {
   int x = 5, y = 0, resultat;
   int exceptionCode = 25;
   try   {  //try-block start
            if (y == 0)   {
          throw exceptionCode;
      }
      resultat = x/y;
   }  //try-block end
```

```
catch (int exceptionCode)  {  //catch-block start

      if  (exceptionCode == 25)   {
         cout << "Divide by zero"
              << endl;
      }
      else   {
         cout << "Exception of   unknown type "
              << endl;
      }

   }//catch-block end

   cout << " Bye" << endl;
return 0;
} //End-main
```

Output: Divide by zero   , Bye.

Example 1: run nr. 2

```
#include <iostream>
using namespace std;
int main()  {
   int x = 5, y = 0, resultat;
    try   {  //try-block start
          if (y == 0)  {
        throw " Divide by Zero ";
      }
      resultat = x/y;
   }  //try-block end
```

```
catch (char *e)  {  //catch-block start

        cout << e << endl;
}//catch-block end

   cout << " Bye" << endl;
return 0;
} //End-main
```

Note that the exception object may be of any type.

Utskrift: Divide by zero , Bye.

Exempel 1: försök nr. 2+

```
#include <iostream>
#include <string>
using namespace std;
int main()  {
   int x = 5, y = 0, resultat;
   try   {  //try-block start
      if (y == 0)  {
        throw string("Divide by Zero ");
      }
      resultat = x/y;
   }  //try-block end
```

```
catch (string e)  {  //catch-block start
        cout << e << endl;
   }//catch-block end
cout << "Bye" << endl;
return 0;
} //End-main
```

Note that the exception object may be of any type.

Utskrift: Divide by zero , Bye.

stack unwinding

When an exception is thrown , the runtime mechanism will first look for a suitable handler (catch) in the same (current) scope (block). If such is not found, the current block exits and the search continues to the next higher level and so on. This is an iterative process , where the search continues until the correct handler is found. The exception is said to be taken care of as soon as the right handler found and if so the stack is cleaned from any local objects that are created on the way from the TRY block until an exception was thrown (and throw) . This process is called stack unwinding . If the correct handler is not found anywhere , the program ends with terminate ( ) . Note that C + + ensures the destruction of all local objects ONLY if the correct handler is found. If an exception is not caught (handled), whether the destruction of all local objects is done or not during the" stack unwinding " process is implementation-dependent. To ensure that everything goes right, add a handler that can catch all kinds of exceptions.

Example 2: stack unwinding

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

struct DivideBySomething{ };
struct DivideByZero{ };
double functionA(int val);
int functionB(int val);
int  main()
{
   try {
        cout<< functionA(0);
   }
   catch (DivideByZero) {
     cout<< "DivideByZero"<<endl;
   }
return 0;
}
```

```
double functionA(int val)
{
  return 2/ functionB(val);
}
```

```
int functionB(int val)
{
   if (val ==0)    {
      throw DivideByZero();
   }
   else
       if (val > 100)    {
         throw DivideBySomething();
   }
            return val;
}
```

What happens if val>100 , and no catch

Exempel 2+: stack unwinding (corrected)

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

double functionA(int val);
int functionB(int val);
int  main()
{
   try {
        cout<< functionA(0);
   }
   catch (string s) {
     cout<< s <<endl;
   }
  catch(...) {
     cout<< "else thing" <<endl;
   }
return 0;
}
```

```
double functionA(int val)
{
  return 2/ functionB(val);
}
```

```
int functionB(int val)  throw (string)
{
   if (val ==0)    {
      throw  string(" Divide By Zero ");
   }
   else
       if (val > 100)    {
         throw string(" DivideBySomething ");
   }
           return val;
}
```

Exempel 2++: stack unwinding

```
#include <iostream>
#include <string>
#include <stdexcept>
using namespace std;

double functionA(int val);
int functionB(int val);

int  main()
{
   try {   cout<<functionA(105);   }
   catch (string s) { cout<< s <<endl;  }
  catch (…) {
     cout<< " else thing " <<endl;
   }
return 0;
}
```

```
double functionA(int val)
{
  return 2/ functionB(val);
}
```

```
int functionB(int val)  throw (string)
{
   if (val ==0)    {
      throw  string(" Divide By Zero ");
   }
   else
       if (val >100 )    {
         throw bad_exception();
   }
           return val;
}
```

Output: else thing

Example 1: run nr. 3

Suppose we in an exception object require: a message and a number of variable values .... What we can do is to create our own exception type.

```
#include <iostream>
#include <string>
using namespace std;
struct DivideByZero  {
//public:
   DivideByZero(int n, int d) : num(n), denom(d), message("Divide by zero") { }
   //~DivideByZero() { }
   int getNumerator() { return num; }
   int getDenominator() { return denom; }
   string getMessage() { return message; }
private:
   int num, denom;   string message;
};
```

Example 1: run nr. 3 (cont.)

```
int main()
{
   int x = 5, y = 0, resultat;
   try {
      if (y == 0) {
         throw DivideByZero(x, y);
      }
      resultat = x/y;
   }
   catch (DivideByZero e) {
      cout << e.getMessage() << endl;
      cout << "Numerator: " << e.getNumerator() << endl;
      cout << "Denominator: " << e.getDenominator() << endl;
   }
return 0;
}
```

Range_Error without ASSERT( )

```
void functionA()
{
  //....do something
    throw range_error(string("some info"));
}

int  main()
{
   try {
            functionA();
   }
   catch (range_error   & r) {
     cout<<r.what() <<endl;
   }
return 0;
}
```

Example:

```
int main() {
    string s = "Object-Based Programming";
    int index, len;
    cout << s << endl;
    while( true ) {
        cout <<"Enter index and length to erase: ";
        cin >> index >> len;
        try {
            s.erase( index, len );
        } catch ( out_of_range ) {
            continue;  //returns TRUE to while-statement==continue
          }
      break; //returns FALSE to while-statement==cancle and go outside while.
    }//end-while
    cout << s << endl;
    return 0; }
```

What is the output?!

```
int main() {
  int test;
  while (cin >> test) {
    try {
        cout << "1 ";
        f(test);
        cout << "2 ";
       }catch (out_of_range e) {
         cout << "3 ";
         cout << e.what();
         cout << "4 ";
       }
    cout << "5 " << endl;
  }
 return 0;
}//end main
```

```
void f(int x) {
    cout << "6 ";
    if (x < 0) {
        throw out_of_range("7 ");
    }else if (x > 100) {
        throw length_error("8 ");
    }
    cout << "9 ";
}//end f
```

1. What is the output for 1?
2. What is the output for -1?
3. What is the output for 1000?

And more Example:

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

void createList(int* &Array, int Size);
int getUserInput( );

int main()
{
    int* Array=0, Dimension=0;
    try {
                createList(Array, Dimension);
    }

    catch (int e) {
                cerr << "Cannot allocate: " << e << endl;
    return -1;
    }

    catch (bad_alloc b)   //The re-thrown exception willl be catched here!
    {
        cerr << "Allocation failed" << endl;
        return -1;
```

Example: (cont.)

```
void createList( int*& Array, int Size)  { //Array is a  reference to a pointer
    Size = getUserInput();
    try {
        Array = new int[Size];
    }
    catch (bad_alloc b) {        // you can catch an exception
        throw(b);                // and re-throw it
    }
}

int getUserInput( )  //This function can throw any kind of exception!
{
    int Response;
    cout << "Please enter the desired dimension."<< endl;
    cin >> Response;
    if (Response <= 0) {  throw (Response); }    // caught in main()
     return Response;
}
```