12 Exceptions

There are situations in which errors can appear when running a program, mostly in the interaction between the program and its environment, when certain values cannot be controllable in the program..

For example, most of the standard I/O functions return a value which indicates if the respective operation ends successfully or not. In the case when the operation failed, an exceptional situation results in the program, which must be treated, so that the program does not lead to greater errors. For this reason, it is indicated as after each call of a standard function that can fail, to test the final value returned by the respective operation:

```
ifstream fout("f_out.txt");
if (!fout) {
  // File open error – must be treated here
}
// Operations performed when no error occurred
```

Not always there is sufficient information for treating a certain error, in the program zone where it was detected. In these cases, the information about the error must be reported to a larger context, usually in the function block which calls the function where the error it is detected.

The C language allows three ways of treating these situations:

a) Returning the error information from the called function to the calling function. This information can be returned by parameters, or by the returned value. In the case when such error information cannot be returned, some global error indicators can be set in the called function and tested in the calling function.
b) Using the standard **`raise`** and **`signal`** functions, for managing the signals of the system. The **`raise`** function can generate an event, and **`signal`** can determine what actions must be performed when the respective event appears.
c) Using the standard **`setjmp`** and **`longjmp`** functions, for performing a non-local jump in the program. These functions allow the jump between different functions of the program: **`setjmp`** has the role to save the program context, and **`longjmp`** to restore the context in case of error.

These three ways have some disadvantages. The first variant, which is the most used by the programmers, supposes the write of the code rather complicated for sending and receiving error information. The last two variants are rather hard to use and suppose a very good knowledge of the standard library of the C language.

The C++ language has an alternative to these variants, which is simpler to use and more efficient, ***exceptions***, and also a mechanism of ***handling these exceptions***.

12.1 Throwing and catching exceptions

Exceptions represent a way to control the program execution when errors appear. The mechanism of handling exceptions supposes the following actions:

- To *suspend* the current execution of the function that detected the error, to *generate* an exception and also an object associated to it, which contains information referring tis error;
- The program execution is *continued* in another zone, where the exception and its associated object is *catch*, and necessary actions for error handling are *performed*.

Handling an exception supposes *two* distinct elements:

- A *non-local transfer* of the program execution between two distinct zones of the program, which is performed automatically by the compiler;
- *Throwing* and *catching* of an exception and its associated object between the two zones of the program.

Throwing an exception and its associated object is performed by a special statement, which has the following syntax:

```
throw [ <expression> ] ;
```

For example, supposing that in a program, an *Err* class was defined, having as members a string and a constructor, the next sequence generates an exception:

```
throw Err("Error at file opening");
```

When executing this sequence:

- The execution of the current function is suspended:
- An object corresponding to the expression following the keyword **`throw`** is created;
- This object is returned to a larger context, even if does not correspond to the type of the return value of the function.

Before transferring the program execution to another context, all objects created from the current context are destroyed. The object related to the exception will be destroyed later, after its reception.

In the case when the **`throw`** statement appears in the body of the function, the mechanism of exception handling forces the current function to end. When leaving the current function is not desired, then a special block, called **`try`** block, can be used.

The syntax of the **`try`** block has the structure:

```
try {
        // Code that can generate exceptions
}
```

and its semantics is the following: the context to which the exception is sent (if it is the case) is immediately exterior to the **try** block. In this way, the actions of catching and handling the exception is performed in the same function.

The catching of an exception is performed by a special construction, called **catch** *clause*, or handler of the exception. The syntax of such a handler is the following:

```
catch (<type> <ident>) {
  // Code for handling the exception
  // associated to the <ident> object
}
```

where **<type>** represents the data type of the object associated to the exception, and **<ident>** is seen as a formal parameter and represents the object name.

A **catch** clause allows handling a single exception, and it must follow a **try** block.

When in a **try** block are generated more exceptions of the same type, a single **catch** clause is enough for handling these exceptions. If several exceptions of different type are generated in a **try** block, more **catch** consecutive clauses can be specified, one clause for each type of exception. For example:

```
try {
  // sequence that can generate 2 types of exceptions
} catch(type1 id1) {
  // handling exception of type 1
} catch(type2 id2) {
  // handling exception of type 2
}
```

Example 12.1. The following program uses the exception handling mechanism, for simulating the actions of throwing and catching of an exception.

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

class A {
public:
  A() { cout << "Constructor A\n"; }
  ~A() { cout << "Destructor A\n"; }
  void f();
};

class B {
public:
  B() { cout << "Constructor B\n"; }
  ~B() { cout << "Destructor B\n"; }
  void g();
};

void A::f() {
  cout << "f in class A\n";
  throw 1;
```

```
  cout << "Statements that will not be executed\n";
}

void B::g() {
  A a;
  cout << "g in class B\n";
  a.f();
  cout << " Statements that will not be executed\n";
}

int main() {
  B b;
  try {
    b.g();
  }
  catch (int k) {
    cout << "Exception handling. k = " << k << endl;
  }
  return 0;
}
```

In the precedent program, in the function *f* from the class *A* is thrown an exception. The execution of the function *f* is interrupted, and the execution of the program is not continued in the calling function *g*, but in the **`main`** function.

Remark. A **`try`** block can exist also at the function level, in which case it replaces the function block. For example:

```
void f() try {
  throw "Function f";
}
catch (const char* msg) {
  cout << msg << endl;
}
```

A sequence of **`catch`** clauses can contain a ***default `catch` clause***. This clause is usually specified on the last position in the sequence, and it contains the "…" operator inside the clause parameters. A default **`catch`** clause specifies the fact the respective handler treats an exception whose type cannot be determined (there is no specific handler for the respective type).

Example 12.2.

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

int main() {
  try {
    int n;
    cin >> n;
    switch (n) {
    case 1: throw 2;
    case 2: throw "Sequence";
```

```
      case 3: throw 12.25;
      }
   }
   catch (char const *mess) {
      cout << "Treating char const * : " << mess << endl;
   }
   catch (int k) {
      cout << "Treating int : " << k << endl;
   }
   catch (...) {
      cout << "Type unknown\n";
   }
   return 0;
}
```

In conclusion, an exception is generated (*thrown*) by a **`throw`** statement and it is received (*caught*) by the first **`catch`** clause, exterior to the context from which it was thrown.

If no matching handler is found, the standard function **`terminate()`** is called;

12.2 Exception specifications

The C++ language does not impose as a certain function which generates exceptions to be declared explicitly. A good programming style imposes to define such a function to inform the possible users of the code about possible thrown exceptions.

The declaration of exceptions that can be thrown by a function can be realized by a specific construction, called ***exception specification***. This is inserted immediately after the function declarator (it follows the function parameter list), as in the following syntax:

`<function declarator> throw '(' <type> {, <type> } ')'`

Remarks.

1. Between parantheses there are specified the data types of the objects associated to exceptions thrown by the function. For example, the sequnce:

```
void f() throw(int, A);
```

specifies that the function *f* can throw two types of exceptions, associated to the data types **`int`** and *A* respectively.

2. When the declarator of a function has no exception specification, this means that the function *can throw any type* of exceptions.

3. When after the keyword **`throw`**, the list type is empty, this means that the respective function *does not generate exceptions*. Example:

```
void f() throw();
```

Because violations of an exception specification of a function are detected only at run-time, the above syntax is called ***dynamic exception specification*** in the standard

C++ 2011. Moreover, dynamic exception specification is *deprecated* in this standard (the last standard of the C++ language until now).

Despite the fact that the C++ community has greeted enthusiastically the introduction of exception specification (several years after the introduction of exceptions, in 1989), exception specifications have proven close to worthless in practice, while adding a measurable overhead to programs. There are several problems with exception specifications in C++, such as:

- *Run-time checking*: C++ exception specifications are checked at runtime rather than at compile time, so they offer no programmer guarantees that all exceptions have been handled;
- *Run-time overhead*: Run-time checking requires the compiler to produce additional code that also hampers optimizations;
- *Unusable in generic code*: Within generic code, it is not generally possible to know what types of exceptions may be thrown from operations on template arguments, so a precise exception specification cannot be written.

This is the reason for which ***dynamic exception specification*** became *deprecated* in the C++ 2011 standard (known as C++11, which is the current ISO C++ standard).

In practice, only two forms of exception-specifications are useful: an operation might throw an exception (any exception) or an operation will never throw any exception. This fact led to a new syntactic construction for exception specifications, called ***noexcept specification***. This contains the keyword **`noexcept`**:

- **`noexcept ( <constant-expression> )`**
- **`noexcept`**

where **`<constant-expression>`** is a constant expression that can be evaluated to **`true`** or **`false`**.

The form **`noexcept`** is equivalent to **`noexcept(true)`**, and both of them specify that the corresponding function *does not throw any exception*. The form **`noexcept(false)`** specifies that the function *can throw any exceptions*, as in the case of the declaratory of the function does not contain an exception specification. With the introduction of ***noexcept specification***, programmers can now express the two kinds of exception guarantees that are useful in practice, without additional overhead.

The keyword **`noexcept`** is regarded as an ***operator*** in C++ 11, following the syntax:

`noexcept( <expression> )`

The **`noexcept`** operator does not evaluate **`<expression`**>:

- The result is **`false`** if the expression contains at least one of the following constructs:
  - A call to any type of function that does not have non-throwing exception specification, unless it is a constant expression;
  - A **`throw`** expression;

    - A **dynamic_cast** expression when the target type is a reference type, and conversion needs a run-time check
    - A **typeid** expression when argument type is polymorphic class type
- In all other cases the result is **true**.

A ***noexcept specification*** on a function *is not check a compile-time*. It represents a method for a programmer to inform the compiler whether or not a function should throw exceptions. The compiler can use this information to *enable certain optimizations* on non-throwing functions as well as *enable* the **noexcept** operator, which can *check at compile time* if a particular expression is declared to throw any exceptions.

If a search for a matching exception handler leaves a function marked **noexcept** or **noexcept(true)**, the standard function **terminate** is called immediately. For example:

```
template <class T>
  void f() noexcept(T()) {}

void g() noexcept(true) {}
void h() noexcept { throw 42; }  // also noexcept(true)

int main() {
    f<int>(); // OK, noexcept(int()) = true
    g(); // OK
    h(); // compiles, but it calls 'terminate'
}
```

12.3 Specific cases for handling exceptions

There are certain cases which can appear when exceptions are thrown and caught. In this paragraph two situations will be described: unhandled exceptions and exceptions re-thrown to a wider context.

12.3.1 Unhandled exceptions

There are situations when for a function which throw exceptions, a set of exception handlers are specified, but because of certain reasons, a different exception can be thrown, which does not match any specified handler. For example, when in a block function, another function is called, about which it is not known if it throws exceptions, and eventually, what kind of exception types it generates.

Two special functions are used by the exception handling mechanism for coping with errors related to the exception handling mechanism itself: **terminate** and **unexpected**:

- **terminate** function is called mainly in two important cases:

    - when the exception handling mechanism cannot find a handler for a thrown exception
    - when the search for a handler encounters the outermost block of a function with a **noexcept** specification that does not allow the exception
- **unexpected** function is called when a function with a dynamic exception specification throws an exception that do not appear in the list of the dynamic exception specification.

By default, **unexpected** calls **terminate**, but a program can install its own handler function. In this case, another predefined function can be used: **set_unexpected**. The function **set_unexpected** receives as parameter a pointer to a handler function, which represents the function that will be called in this situation. The prototype of such a handler has the following form:

```
typedef void (*unexpected_handler)();
```

Remark. There are some compilers that do not use the **set_unexpected** function. Instead of **set_unexpected** they use another function, **set_terminate**, for specifying the handler function which will be called.

Example 12.3. The following program (that run under the Windows operating system and it is compiled with a Microsoft C++ compiler) uses the **set_terminate** function for specifying the function that will be called when handling mechanism cannot find any specific handler for an exception.

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

class A { };
class B { };

void g();

void f(int i) throw (A, B) {
  switch(i) {
    case 1: throw A();
    case 2: throw B();
  }
  g();
}

void g() { throw 47; }

void my_terminate() {
  cout << "Exception unknown\n";
  exit(1);
}

int main() {
  set_terminate(my_terminate);
  for(int i = 1; i <=3; i++)
    try {
```

```
      f(i);
    }
    catch(A) {
      cout << "Treating A exception" << endl;
    }
    catch(B) {
      cout << "Treating B exception" << endl;
    }
   return 0;
}
```

12.3.2 Re-throwing exceptions

Another specific situation that can appear is the case when a certain exception is caught in a block, in which is not sufficient information for handling it. So, the exception is re-thrown to a larger context, all information related to the exception remaining unaltered.

Rethrowing an exception is performed by the **`throw`** statement, without any attached expression:

```
throw;
```

Example 12.4. The program from the example 12.2 is modified, in which the exception is re-thrown to a superior context.

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

int main() {
  try {    // Level 0
    try {  // Level 1
      int n;
      cin >> n;
      switch (n) {
      case 1: throw 2;
      case 2: throw "Sequence";
      case 3: throw 12.25;
      }
    }
    catch (char const *mess) {
      cout << "Level 1. char const * : " << mess << endl;
    }
    catch (int k) {
      cout << "level 1. int : " << k << endl;
    }
    catch (...) {
      cout << "level 1. Unknown type\n";
      throw;     // Rethrow the exception
    }
  }
  catch (double d) {
    cout << "Level 0. double : " << d << endl;
```

```
  }
  return 0;
}
```

In the previous example, when the value of *n* is 3, an exception of type **double** it is thrown. It can not be completely handled at level 1, and it is re-thrown to the immediate exterior block (level 0). At this level there is a specific handler, and it displays the value related to the exception (12.25).

12.4 Exceptions, constructors and destructors

The main disadvantage of using the **setjmp** and **longjmp** standard functions is that in this case, the objects that are on the current stack-frame cannot be destroyed when jumping to the exterior function. The exception handling mechanism of the C++ language allows solving this problem.

All the complete created objects in a block where an exception is thrown are destroyed by the compiler in the exception handler where the exception was caught.

Example 12.5. The following program uses the *Object* class for displaying the moment of creation and destruction of objects.

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

class Object {
  string name;
public:
  Object(string n): name(n) {
    cout << "Constructor for object " << n << endl;
  }
  Object(Object const &o): name(o.name + " (copy)")
  {
    cout << "Copy-constructor for object "
           << name << endl;
  }
  ~Object() {
    cout << "Destructor for object " << name << endl;
  }
  void f() {
    Object toThrow("'local obj'");
    cout << "Function f from object " << name << endl;
    throw toThrow;
  }
  void hello() {
     cout << "Hello from the object " << name << endl;
  }
};

int main() {
  Object out("'main obj'");
```

```
  try {
    out.f();
  }
  catch (Object o) {
    cout << " exception treating\n";
    o.hello();
  }
  cout << "After the catch clause\n";
  return 0;
}
```

The program output is:

```
Constructor for object 'main obj'
Constructor for object 'local obj'
Function f from the object 'main obj'
'local obj' (copy)
Copy-constructor for the object 'local obj' (copy) (copy)
Destructor for object 'local obj'
Exception treating
Hello from the object 'local obj' (copy) (copy)
Destructor for object 'local obj' (copy) (copy)
Destructor for object 'local obj' (copy)
After the catch clause
Destructor for object 'main obj'
```

Remarks.

1. The function *f* from the *Object* class generates an exception, so that after the message from 3rd line, it is created by copying an exception object in the **`throw`** statement (message in line 4).
2. Because the **`catch`** clause is acting like a function, it is created (by copying the object generated by the exception) an object of the *Object* class for the *o* parameter (5th line message). The local object is destroyed (line 6) and the *f* function execution ends, the program execution continues in the **`catch`** clause from the **`main`** function.
3. After displaying the message from the *hello* function, the body of the **`catch`** clause ends and the destructor for the *o* object is called. In addition, the object associated to the exception generated by the **`throw`** statement (messages from lines 9 and 10) is also destroyed.
4. After the **`main`** function ends, the object *out* from the **`main`** function is also destroyed.

If is desired as an object exception not be copied again in the **`catch`** clause (passing by value), a *reference* to the exception object can be specified in the place of the respective object.

For example, if in the previous program, the catch clause is modified as follows:

```
catch (Object &o) {
```

the program output is:

```
Constructor for object 'main obj'
Constructor for object 'local obj'
Function f from the object 'main obj'
Copy-constructor for the object 'local obj' (copy)
Destructor for object 'local obj'
Exception treating
Hello from the object 'local obj' (copy)
Destructor for object 'local obj' (copy)
After the catch clause
Destructor for object 'main obj'
```

12.5 Classes and class hierarchies for handling exceptions

For the determination of a **`catch`** clause that will be executed in a sequence of several clauses, the compiler determines the first **`catch`** clause from the sequence that matches the object type thrown by the exception.

If a program can throw several exceptions related to each other, the classes associated to these exceptions can be combined into a class hierarchy, using public inheritance. In this case, the selection of the catch clauses must be treated carefully. Because of the ***upcasting*** mechanism, when several handlers are specified after a **`try`** block, they order is important: handlers that use as parameters objects from the bottom of the class hierarchy must be specified first.

Example 12.6. The following program uses a class hierarchy containing three exception classes.

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

class Advertisement {};
class Error: public Advertisement {};
class FatalError: public Error {};

void f() {
  int n;
  cout << "Error level: ";
  cin >> n;
  switch (n) {
    case 1: throw Advertisement();
    case 2: throw Error();
    case 3: throw FatalError();
  }
}

int main() {
  try {
    f();
  }
  catch(Advertisement) {
    cout << "" << endl;
```

```
  // The next handlers are covered by the precedent
  }
  catch(Error) {
    cout << "" << endl;
  }
  catch(FatalError) {
    cout << "" << endl;
  }
  return 0;
}
```

One can observe regardless of the type of the thrown exception, always the first handler is selected. A correct version of the **`main`** function is the following:

```
int main() {
  try {
    f();
  }
  catch(EroareFatala) {
    cout << " Treating exception FatalError << endl;
  }
  catch(Eroare) {
    cout << " Treating exception Error " << endl;
  }
  catch(Avertisment) {
    cout << " Treating exception Advertisement " << endl;
  }
  return 0;
}
```

A commonly variant for class hierarchies related to exceptions is the use of ***polymorphism*** and ***pointers to the base class***. In this case the base class of the hierarchy is indicated to be an abstract class.

Because in this case pointers are used for sending information related on the current exception, there are certain rules that must be used:

- Each **`catch`** clause has as parameter a pointer to the base class of the class hierarchy,
- Objects related to exceptions are created in the **`try`** block,
- These objects (created in the **`try`** block) must be destroyed explicitly within the exception handlers using **`delete`** operator.

Example 12.7. The following program defines a class hierarchy with an abstract base class. The pure virtual function *ExceptionProcessing* is overridden in all derived classes for processing exceptions.

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

class Exception {
protected:
  char m[20];    // Exception message
public:
```

```
  virtual ~Exception() { }
  virtual void Exception processing() = 0;
friend ostream& operator<<(ostream& os, Exception& e)
  { return os << e.m; }
};

class Advertisement: public Exception {
public:
  Advertisement(char* s) {
    strcpy(m, s);
  }
  void Exception processing() {
    cout << m;
    exit(1);
  }
};

class Error: public Exception {
public:
  Error(char* s) {
    strcpy(m, s);
  }
  void Exception processing() {
    cout << m;
    exit(1);
  }
};

class FatalError: public Exception {
public:
  Fatal Error(char* s) {
    strcpy(m, s);
  }
  void Exception processing() {
    cout << m;
    exit(1);
  }
};

void f() {
  int n;
  cout << "Error level: ";
  cin >> n;
  switch (n) {
    Exception* e;
    case 1:
      e = new Advertisement("Advertisement !");
      throw e;
    case 2:
      e = new Error("Error !!");
      throw e;
    case 3:
      e = new Fatal Error("Error fatal !!!");
      throw e;
  }
}
```

```
int main() {
  try {
    f();
  }
  catch(Exception* e) {
    e->Exception processing();
     delete e;
  }
  return 0;
}
```

C++ programs may use predefined classes of the standard library related to exceptions. This class hierarchy has a base class called **exception**. The public function **what** of the class **exception** can be used for displaying a text describing the error.

From the class **exception** there are derived two other classes: **logic_error** for the errors detected at compilation phase, and **runtime_error**, for errors detected at runtime.

The main classes derived from **logic_error** are:

- **domain_error** – reports violation of a precondition
- **invalid_argument** – invalid argument for a function
- **length_error** – an object with a length greater than NPOS (the maximum value that can be represented)
- **out_of_range** – outside of the range
- **bad_cast** – an erroneous **dynamic_cast** operation

The main classes derived from **runtime_error** are:

- **range_error** – a post-condition violation
- **overflow_error** – wrong arithmetic operation (overflow)
- **bad_alloc** – wrong memory allocation

12.6 Overloading *new* and *delete* operators

Overloading of these operators is described in this chapter, because these functions, generally, use exceptions. The initial versions of the **new** operator return the **NULL** pointer when memory allocation has failed. The current version uses exceptions for handling these cases, even if most of the C++ compilers support both styles.

The operators described in Chapter 9 can be overloaded for classes defined by the programmers. The operators **new** and **delete** can be overloaded in several ways:

- for user defined classes,
- at global level,
- for arrays (**new[]** and **delete[]**).

Overloading the operators **new** and **delete** must be careful performed, because the semantics of these operators must be preserved.

12.6.1 Handling exceptions when memory allocation fails

The main task of the **new** operator is to determine a free memory zone (with an appropriate length) and to return its starting address. As a consequence, such an operator must have at least one parameter of **size_t** type and it returns a **void*** type.

The modern versions of the C++ compilers use exceptions when they cannot allocate the required memory zone. In this case it is called a specific function, named ***handler of the allocation error***. This handler can throw an exception of **bad_alloc** type, or the exception can be thrown by the operator itself.

The main structure of a **new** operator can have the following form, although in reality is can be more complicated:

```
void operator new(size_t size) {
  // allocation of a 'size' bytes of memory
  if ( /* successful allocation */ )
    return ( /* pointer to the start zone address */ );
  set_new_handler(handler);
  (*handler)();
  throw bad_alloc();
}
```

In the above code *handler* represents a pointer to the handler function that returns *void*. The predefined function **set_new_handler** allows setting the function which will be the error allocation handler.

In general, the goal of the allocation handler is to collect the memory zones which are not used in the program (*garbage collection*), but this is a difficult operation.

Usually, the allocation handler frees a certain memory zone and displays a warning message. For this, it is used a certain memory zone, which is dynamic allocated at the start of the program, and this memory can be accessed and freed by the handler.

Example 12.8. The next program uses the **set_new_handler** function, for setting a specific handler for allocation errors.

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

// Memory block used for displaying an error message
char *buff = new char[128];

void my_new_handler() {
  // It frees the block 'buff'
```

```
  delete buff;
  printf( "Allocation failed\n" );
}

int main() {
  // New handler declaration
  set_new_handler(my_new_handler);
  // Pointer to a block memory
  int* p;
  for (double k=0; ; k++)
    p = new int[1024];
  // ...
}
```

12.6.2 Overloading global operators *new* and *delete*

The **new** and **delete** operators can be overloaded at global level (outside of any classes). In this case they override the operators with the same name from the standard namespace **std**, so that they cannot be used. For this reason, they must carefully implemented..

Usually, overloading global operators **new** and **delete** is justified when the memory allocation method used by the standard **new** and **delete** operators is not satisfactory for certain applications, especially for real time applications, or in the case when the memory fragmentation can appear.

For memory allocation the standard allocation functions (**malloc**, **calloc**, **realloc**) can be used, and for freeing memory, the **free** function. The **delete** operator has a **void*** type parameter and it deallocates the previous allocated memory using the pointer specified as parameter.

When these operators are used for the creation/destroying of some objects, in addition, these operators call constructors/destructors for respective classes. This call cannot be controlled by the program; it is managed directly by the compiler.

Example 12.9. The following programs uses overloading of global new and delete operators.

```
#include <cstdio>
#include <cstdlib>
using namespace std;

void* operator new(size_t sz) {
  printf("Allocation %d bytes\n", sz);
  void* m = malloc(sz);
  if(!m) puts("Allocation error\n");
  return m;
}

void operator delete(void* m) {
  puts("Free");
```

```
  free(m);
}

class A {
  int i[100];
public:
  S() { puts("Constructor A"); }
  ~S() { puts("Destructor A"); }
};

int main() {
  puts("Allocation/free int");
  int* p = new int(47);
  delete p;
  puts("Allocation/free A");
  A* a = new A;
  delete s;
  puts("Allocation/free A[5]");
  A* sa = new A[5];
  delete []sa;
  return 0;
}
```

The functions **printf** and **puts** were used because at the creation of the objects **cin**, **cout** and **cerr**, the operators **new** and **delete** are used.

12.6.3 Overloading operators *new* and *delete* for classes

When operators **new** and **delete** are overloaded for a specific class, they perform a subsequent call for an appropriate constructor/destructor of the corresponding class. Because the actions from these operators do not have access to the related object (the constructor is called after the specific operations of the **new** operator are performed, and the actions of the destructor are performed in the reverse order), the function **operator new** and **operator delete** must be static functions, even they are not explicit declared.

When operators **new** and **delete** are overloaded for classes, they may use global operators in order to allocate memory for data members. However, there are cases when calling global operators is not necessary. For example, one can use a specific scheme for memory allocation, not in the *heap* zone, but in the zone of *static data*. When the memory is allocated in the *heap* zone, using exceptions is indicated, both in the overloaded **new** operator, and in the functions that use this operator.

Exemplul 13.10. The following programs use overloading new and delete operators for a class, using the heap zone memory.

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

#define dim 10
```

```
class X {
  int n;
public:
  X(int a = 0): n(a) { }
  static void * operator new(size_t) throw(bad_alloc);
  static void operator delete(void*);
  void Print() const { cout << n << endl; }
};

void* X::operator new(size_t) throw(bad_alloc) {
  void *memory;
  cout << "Operator new\n";
  try {    // Memory allocation
    memory = ::operator new(sizeof(X));
  }
  catch (bad_alloc&) {  // Rethrowing exception
    throw;
  }
  return memory;
}

void X::operator delete(void* memory) {
  cout << "Operator delete\n";
  if (memory == 0) {
    return;
  }
  ::delete(memory);
}

int main() {
  X* pv[dim];
  int k;
  try {
    for (k=0; k<dim; k++) {
      pv[k] = new X(k);
    }
  }
  catch(bad_alloc) {
    cerr << "Allocation error\n";
  }
  for (k=0; k<dim; k++) {
    pv[k]->Print();
  }
  for (k=0; k<dim; k++) {
    delete pv[k];
  }
  return 0;
}
```

Remark. The value of the parameter **`size_t`** of the new operator is initialized by the compiler.

For a class can exist several overloaded functions for operators **`new`** and **`delete`**. In this case they can have a different number of parameters, but always the first

parameter has the type **size_t** and its value is initialized by the compiler. The other parameters can be used for realizing some specific operations.

The syntax for calling these operators is quite different. In the case of the operator **new**, all parameters, starting to the second parameter, are specified between the keyword **new** and the name of the corresponding class. For example, if for a class X, an operator new was defined, having the following form:

```
static void* operator new(size_t, int, int)
      throw(bad_alloc);
```

then, calling this operator can have the following form:

```
X* p = new (5, 3) X;
```

In the case of operator **delete**, to be able to specify the variant of the function **delete** that will be called, one can use its explicit call. For example, for an operator of the form:

```
static void operator new(void*, int, int);
```

can be called by using the following syntax:

```
p->X::~X(5, 3);
```

13.6.4. Overloading operators *new[]* and *delete[]*

Even in the case when **new** and **delete** have been overloaded for a specific class, if an array of objects of this class must be allocated, then the compiler calls global operators. For example, if for the class *X* from previous example, an array of the following form is declared:

```
X v[dim];
```

one can observe that the operators **new** and **delete**, overloaded for the class *X*, are not called (messages specified in these operators are not displayed).

In these cases it is necessary to overload operator for memory allocation and deallocation for arrays: **new[]** and **delete[]**.

For example, for the following class *Object*, two operators **new[]** and two operators **delete[]** have been overloaded.

```
class Object {
  int n;
public:
  void* operator new[](size_t size);
  void* operator new[](size_t size, int extra);
  void operator delete[](void *p);
  void operator delete[](void *p, int extra);
};
```

The first variant of **new[]** represents the main form, which returns a **void*** pointed, and it receive a **size_t** value. This value represents the number of memory bytes that

have to be allocated for the desired number of objects (instances of the Object class). A usual implementation of this operator uses the global operator **new**, as in the following example:

```
void *Object::operator new[](size_t size) {
  return ::new Object[size/sizeof(Object)];
}
```

As in the case of operator **new**, also in this case additional parameters can be specified. For example, the second variant of the operator **new[]** of the class *Object* uses an additionally parameter for initialization of elements.

```
void *Object::operator new[](size_t size, int extra) {
  unsigned n = size / sizeof(Object);
  Object* op = ::new Object[n];
  for (unsigned idx = 0; idx < n; idx++) {
    op[idx].value = extra;
  }
  return op;
}
```

The following declaration performs, in addition, the initialization of the array components:

```
Object* ov = new(120) Object[15];
```

The first variant of the operator **delete[]** from the class *Object* represents the main form of this operator. It receives as parameter, the address of the array of objects, before allocated with **Object::new[]**. A usual implementation of this operator uses the global operator **delete[]**, as in the following example:

```
void Object::operator delete[](void *p) {
  ::delete[] reinterpret_cast<Object *>(p);
}
```

The second variant of the operator **delete[]** allows to specify some additional parameters. In this case, when calling **delete[]**, the list of the parameters is specified after brackets. For example:

```
delete[] (new Object[5], 100);
```