Chapter 56. Boost.Exception

The library Boost.Exception provides a new exception type, `boost::exception`, that lets you add data to an exception after it has been thrown. This type is defined in `boost/exception/exception.hpp`. Because Boost.Exception spreads its classes and functions over multiple header files, the following examples access the master header file `boost/exception/all.hpp` to avoid including header files one by one.

Boost.Exception supports the mechanism from the C++11 standard that transports an exception from one thread to another. `boost::exception_ptr` is similar to `std::exception_ptr`. However, Boost.Exception isn't a full replacement for the header file `exception` from the standard library. For example, Boost.Exception is missing support for nested exceptions of type `std::nested_exception`.

Note

To compile the examples in this chapter with Visual C++ 2013, remove the keyword `noexcept`. This version of the Microsoft compiler doesn't support `noexcept` yet.

Example 56.1. Using `boost::exception`

```
#include <boost/exception/all.hpp>
#include <exception>
#include <new>
#include <string>
#include <algorithm>
#include <limits>
#include <iostream>

typedef boost::error_info<struct tag_errmsg, std::string> errmsg_info;

struct allocation_failed : public boost::exception, public std::exception
{
  const char *what() const noexcept { return "allocation failed"; }
};

char *allocate_memory(std::size_t size)
{
  char *c = new (std::nothrow) char[size];
  if (!c)
    throw allocation_failed{};
  return c;
}

char *write_lots_of_zeros()
{
  try
  {
    char *c = allocate_memory(std::numeric_limits<std::size_t>::max());
    std::fill_n(c, std::numeric_limits<std::size_t>::max(), 0);
    return c;
  }
  catch (boost::exception &e)
  {
    e << errmsg_info{"writing lots of zeros failed"};
    throw;
  }
}
```

```
int main()
{
  try
  {
    char *c = write_lots_of_zeros();
    delete[] c;
  }
  catch (boost::exception &e)
  {
    std::cerr << boost::diagnostic_information(e);
  }
}
```

Example 56.1 calls the function `write_lots_of_zeros()`, which in turn calls `allocate_memory()`. `allocate_memory()` allocates memory dynamically. The function passes `std::nothrow` to `new` and checks whether the return value is 0. If memory allocation fails, an exception of type `allocation_failed` is thrown. `allocation_failed` replaces the exception `std::bad_alloc` thrown by default if `new` fails to allocate memory.

`write_lots_of_zeros()` calls `allocate_memory()` to try and allocate a memory block with the greatest possible size. This is done with the help of `max()` from `std::numeric_limits`. The example intentionally tries to allocate that much memory to make the allocation fail.

`allocation_failed` is derived from `boost::exception` and `std::exception`. Deriving the class from `std::exception` is not necessary. `allocation_failed` could have also been derived from a class from a different class hierarchy in order to embed it in an existing framework. While Example 56.1 uses the class hierarchy defined by the standard, deriving `allocation_failed` solely from `boost::exception` would have been sufficient.

If an exception of type `allocation_failed` is caught, `allocate_memory()` must be the origin of the exception, since it is the only function that throws exceptions of this type. In programs that have many functions calling `allocate_memory()`, knowing the type of the exception is no longer sufficient to debug the program effectively. In those cases, it would help to know which function tried to allocate more memory than `allocate_memory()` could provide.

The challenge is that `allocate_memory()` does not have any additional information, such as the caller name, to add to the exception. `allocate_memory()` can't enrich the exception. This can only be done in the calling context.

With Boost.Exception, data can be added to an exception at any time. You just need to define a type based on `boost::error_info` for each bit of data you need to add.

`boost::error_info` is a template that expects two parameters. The first parameter is a *tag* that uniquely identifies the newly created type. This is typically a structure with a unique name. The second parameter refers to the type of the value stored inside the exception. Example 56.1 defines a new type, `errmsg_info` – uniquely identifiable via the structure `tag_errmsg` – that stores a string of type `std::string`.

In the `catch` handler of `write_lots_of_zeros()`, `errmsg_info` is used to create an object that is initialized with the string "writing lots of zeros failed". This object is then added to the exception of type `boost::exception` using `operator<<`. Then the exception is re-thrown.

Now, the exception doesn't just denote a failed memory allocation. It also says that the memory allocation failed when the program tried to write lots of zeros in the function `write_lots_of_zeros()`. Knowing which function called `allocate_memory()` makes debugging larger programs easier.

To retrieve all available data from an exception, the function `boost::diagnostic_information()` can be called in the `catch` handler of `main()`. `boost::diagnostic_information()` calls the member function `what()` for each exception passed to it and accesses all of the additional data stored inside the exception. `boost::diagnostic_information()` returns a string of type `std::string`, which, for example, can be written to standard error.

When compiled with Visual C++ 2013, Example 56.1 will display the following message:

```
Throw location unknown (consider using BOOST_THROW_EXCEPTION)
Dynamic exception type: struct allocation_failed
std::exception::what: allocation failed
[struct tag_errmsg *] = writing lots of zeros failed
```

The message contains the type of the exception, the error message retrieved from `what()`, and the description, including the name of the structure.

`boost::diagnostic_information()` checks at run time whether or not a given exception is derived from `std::exception`. `what()` will only be called if that is the case.

The name of the function that threw the exception of type `allocation_failed` is unknown.

Boost.Exception provides a macro to throw an exception that contains not only the name of the function, but also additional data such as the file name and the line number.

Example 56.2. More data with `BOOST_THROW_EXCEPTION`

```
#include <boost/exception/all.hpp>
#include <exception>
#include <new>
#include <string>
#include <algorithm>
#include <limits>
#include <iostream>

typedef boost::error_info<struct tag_errmsg, std::string> errmsg_info;

struct allocation_failed : public std::exception
{
  const char *what() const noexcept { return "allocation failed"; }
};

char *allocate_memory(std::size_t size)
{
  char *c = new (std::nothrow) char[size];
  if (!c)
    BOOST_THROW_EXCEPTION(allocation_failed{});
  return c;
}

char *write_lots_of_zeros()
{
  try
```

```
  {
    char *c = allocate_memory(std::numeric_limits<std::size_t>::max());
    std::fill_n(c, std::numeric_limits<std::size_t>::max(), 0);
    return c;
  }
  catch (boost::exception &e)
  {
    e << errmsg_info{"writing lots of zeros failed"};
    throw;
  }
}

int main()
{
  try
  {
    char *c = write_lots_of_zeros();
    delete[] c;
  }
  catch (boost::exception &e)
  {
    std::cerr << boost::diagnostic_information(e);
  }
}
```

Using the macro `BOOST_THROW_EXCEPTION` instead of `throw`, data such as function name, file name, and line number are automatically added to the exception. But this only works if the compiler supports macros for the additional data. While macros such as `__FILE__` and `__LINE__` have been standardized since C++98, the macro `__func__`, which gets the name of the current function, only became standard with C++11. Because many compilers provided such a macro before C++11, `BOOST_THROW_EXCEPTION` tries to identify the underlying compiler and use the corresponding macro if it exists.

Compiled with Visual C++ 2013, Example 56.2 displays the following message:

```
main.cpp(20): Throw in function char *__cdecl allocate_memory(unsigned int)
Dynamic exception type: class boost::exception_detail::clone_impl<struct boos
std::exception::what: allocation failed
[struct tag_errmsg *] = writing lots of zeros failed
```

In Example 56.2, `allocation_failed` is no longer derived from `boost::exception`. `BOOST_THROW_EXCEPTION` accesses the function `boost::enable_error_info()`, which identifies whether or not an exception is derived from `boost::exception`. If not, it creates a new exception type derived from the specified type and `boost::exception`. This is why the message shown above contains a different exception type than `allocation_failed`.

Example 56.3. Selectively accessing data with `boost::get_error_info()`

```
#include <boost/exception/all.hpp>
#include <exception>
#include <new>
#include <string>
#include <algorithm>
#include <limits>
#include <iostream>

typedef boost::error_info<struct tag_errmsg, std::string> errmsg_info;

struct allocation_failed : public std::exception
{
```

```
  const char *what() const noexcept { return "allocation failed"; }
};

char *allocate_memory(std::size_t size)
{
  char *c = new (std::nothrow) char[size];
  if (!c)
    BOOST_THROW_EXCEPTION(allocation_failed{});
  return c;
}

char *write_lots_of_zeros()
{
  try
  {
    char *c = allocate_memory(std::numeric_limits<std::size_t>::max());
    std::fill_n(c, std::numeric_limits<std::size_t>::max(), 0);
    return c;
  }
  catch (boost::exception &e)
  {
    e << errmsg_info{"writing lots of zeros failed"};
    throw;
  }
}

int main()
{
  try
  {
    char *c = write_lots_of_zeros();
    delete[] c;
  }
  catch (boost::exception &e)
  {
    std::cerr << *boost::get_error_info<errmsg_info>(e);
  }
}
```

Example 56.3 does not use `boost::diagnostic_information()`, it uses `boost::get_error_info()` to directly access the error message of type `errmsg_info`. Because `boost::get_error_info()` returns a smart pointer of type `boost::shared_ptr`, `operator*` is used to fetch the error message. If the parameter passed to `boost::get_error_info()` is not of type `boost::exception`, a null pointer is returned. If the macro `BOOST_THROW_EXCEPTION` is always used to throw an exception, the exception will always be derived from `boost::exception` – there is no need to check the returned smart pointer for null in that case.