Advanced Concepts:

21. Exceptions

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in our program by transferring control to special functions called *handlers*.

To catch exceptions we must place a portion of code under exception inspection. This is done by enclosing that portion of code in a *try block*. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

A exception is thrown by using the throw keyword from inside the try block. Exception handlers are declared with the keyword `catch`, which must be placed immediately after the try block:

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

int main () {
  try
  {
    throw 20;
  }
  catch (int e)
  {
    cout << "An exception
occurred. Exception Nr. " << e <<
endl;
  }
  return 0;
}
```

```
An exception occurred. Exception
Nr. 20
```

The code under exception handling is enclosed in a `try` block. In this example this code simply throws an exception:

```
throw 20;
```

A throw expression accepts one parameter (in this case the integer value 20), which is passed as an argument

to the exception handler.

The exception handler is declared with the `catch` keyword. As you can see, it follows immediately the closing brace of the `try` block. The catch format is similar to a regular function that always has at least one parameter. The type of this parameter is very important, since the type of the argument passed by the throw expression is checked against it, and only in the case they match, the exception is caught.

We can chain multiple handlers (catch expressions), each one with a different parameter type. Only the handler that matches its type with the argument specified in the throw statement is executed.

If we use an ellipsis (`...`) as the parameter of `catch`, that handler will catch any exception no matter what the type of the `throw` exception is. This can be used as a default handler that catches all exceptions not caught by other handlers if it is specified at last:

```
try {
  // code here
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

In this case the last handler would catch any exception thrown with any parameter that is neither an `int` nor a `char`.

After an exception has been handled the program execution resumes after the `try-catch` block, not after the `throw` statement!.

It is also possible to nest `try-catch` blocks within more external `try` blocks. In these cases, we have the possibility that an internal `catch` block forwards the exception to its external level. This is done with the expression `throw;` with no arguments. For example:

```
try {
  try {
      // code here
  }
  catch (int n) {
      throw;
  }
}
catch (...) {
  cout << "Exception occurred";
}
```

Exception specifications

When declaring a function we can limit the exception type it might directly or indirectly throw by appending a `throw` suffix to the function declaration:

```
float myfunction (char param) throw (int);
```

This declares a function called `myfunction` which takes one agument of type `char` and returns an element of type `float`. The only exception that this function might throw is an exception of type `int`. If it throws an exception with a different type, either directly or indirectly, it cannot be caught by a regular `int`-type handler.

If this `throw` specifier is left empty with no type, this means the function is not allowed to throw exceptions. Functions with no `throw` specifier (regular functions) are allowed to throw exceptions with any type:

```
int myfunction (int param) throw();

    // no exceptions allowed
int myfunction (int param);

    // all exceptions allowed
```

Standard exceptions

The C++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions. It is called `exception` and is defined in the `<exception>` header file under the `namespace std`. This class has the usual default and copy constructors, operators and destructors, plus an additional virtual member function called `what` that returns a null-terminated character sequence (`char *`) and that can be overwritten in derived classes to contain some sort of description of the exception.

```
// standard exceptions
#include <iostream>
#include <exception>
using namespace std;

class myexception: public
exception
{
  virtual const char* what() const
throw()
  {
    return "My exception
happened";
  }
} myex;

int main () {
  try
  {
    throw myex;
```

```
  }
  catch (exception& e)
  {
    cout << e.what() << endl;
  }
  return 0;
}
```

```
My exception happened.
```

We have placed a handler that catches exception objects by reference (notice the ampersand & after the type), therefore this catches also classes derived from `exception`, like our `myex` object of class `myexception`.

All exceptions thrown by components of the C++ Standard library throw exceptions derived from this `std::exception` class. These are:

exception	description
bad_alloc	thrown by new on allocation failure
bad_cast	thrown by dynamic_cast when fails with a referenced type
bad_exception	thrown when an exception type doesn't match any catch
bad_typeid	thrown by typeid
ios_base::failure	thrown by functions in the iostream library

For example, if we use the operator `new` without `(nothrow)` and the memory cannot be allocated, an exception of type `bad_alloc` is thrown:

```
try
{
  int * myarray= new int[1000];
}
catch (bad_alloc&)
{
  cout << "Error allocating memory." << endl;
}
```

It is recommended to include all dynamic memory allocations within a try block that catches this type of exception to perform a clean action instead of an abnormal program termination, which is what happens when this type of exception is thrown and not caught. If you want to force a `bad_alloc` exception to see it in action, you can try to allocate a huge array; On my system, trying to allocate 1 billion `int`s threw a `bad_alloc` exception.

Because `bad_alloc` is derived from the standard base class `exception`, we can handle that same exception by catching references to the `exception` class:

```
// bad_alloc standard exception
#include <iostream>
#include <exception>
using namespace std;
```

```
int main () {
  try
  {
    int* myarray= new int[1000];
  }
  catch (exception& e)
  {
    cout << "Standard exception: "
<< e.what() << endl;
  }
  return 0;
}
```