C++ Namespaces, Exceptions

CSci 588: Data Structures, Algorithms and Software Design

http://www.cplusplus.com/doc/tutorial/namespaces/
http://www.cplusplus.com/doc/tutorial/exceptions/
http://www.cplusplus.com/doc/tutorial/typecasting/

Namespaces

Namespaces can group classes, objects and functions under a name.

This helps prevent code included from multiple sources from conflicting.

For example, what if you include code from two separate libraries, each which have a max function or their own implementation of a string class?

Namespaces

Namespaces are specified similar to a class — in a sense they're a class with all members static.

```
namespace <name> {
    <fields / classes / functions>
}
```

Using Namespaces

You've already used namespaces before, you can either use:

using namespace <name>;

To use everything within the namespace without having to qualify it (using the :: operator). Or you can specify what namespace something comes from:

<name>::<field / class / function>

Just like:

std::cout << "Hello World!" << std::endl;

Uses cout and endl from the std namespace.

Using Namespaces

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

namespace first
{
  int x = 5;
}

namespace second
{
  double x = 3.1416;
}

int main () {
  {
    using namespace first;
    cout << x << endl;
  }
  {
    using namespace second;
    cout << x << endl;
  }
  return 0;
}
```

You can use:

using namespace <name>;

Anywhere within your code (it doesn't have to go at the top of the file).

This will allow you to use all the fields, classes and functions within that namespace within the scope the using statement was used.

‘Overlapping’ Namespaces

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

namespace first
{
  int x = 5;
}

namespace second
{
  double x = 3.1416;
}

int main () {
   using namespace first;
   cout << x << endl;
   cout << second::x << endl;
   return 0;
}
```

You can use a similarly named field/class/ function from a different namespace by qualifying it.

Nested Namespaces

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

namespace first
{
    namespace second
    {
        double x = 3.1416;
    }
}


int main () {
    cout << first::second::x << endl;
    return 0;
}
```

You can also nest your namespaces (boost and opencv do this frequently).

Aliasing Namespaces

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

namespace first {
    double x = 3.1416;
}

namespace second = first;


int main () {
    cout << second::x << endl;
    return 0;
}
```

You can also rename a namespace.

Exceptions

Sometimes problems happen in our programs due to improper input, etc.

In the olden days of programming, a common programming style was to have functions return error codes (and have the actual return values be things passed by reference). For example:

```
int convert_to_int(char* my_str, int &my_int) {
  …
  if (!is_int(my_str)) return 1; //error
  …
  my_int = converted_value;

  return 0; //success
}
```

Exceptions

However, this makes for some messy code, and makes it so code calling the function had to handle any errors:

```
int convert_to_int(char* my_str, int &my_int) {
  …
  if (!is_int(my_str)) return 1; //error
  …

  my_str = …;

  return 0; //success
}

…

int retval = convert_to_int(something, new_int);
if (retval == 1) {
  //handle error 1
} else if (retval == 2) (
  // handle error 2
} else {
  //success, we can actually do something.
}
```

Exceptions

Exceptions provide a better way to handle issues that occur within functions, as well as a way to delegate farther up the call stack any problems that might occur.

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

int main () {
  try {
    throw 20;
  } catch (int e) {
    cout << "An error occurred: " << e << endl;
  }
  return 0;
}
```

Exceptions

You can throw exceptions from within functions.

They also can be of any type.

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

int convert_to_int(char* my_str) throws (int) {
  if (0 == strcmp(my_str, "")) {
    throw 30;
  }
  return 10;
}

int main (int argc, char **argv) {
  try {
    convert_to_int(argv[1]);
  } catch (int e) {
    cout << "An error occurred: " << e << endl;
  }
  return 0;
}
```

Exceptions

A throw statement immediately exits the function and goes into the catch statements from the function that called it.

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

int convert_to_int(char* my_str) throws (int) {
  if (0 == strcmp(my_str, "")) {
    throw 30;
  }

  return 10;
}

int main(int argc, char **argv) {
  int result = 0;
  try {
    result = convert_to_int(argv[0]);
  } catch (int e) {
    cout << "An error occurred: " << e << endl;
  }
  //This will print out 0.
  cout << “result is: “ << result << endl;
  return 0;
}
```

Exceptions

If there is no catch statement in a function, it will throw the exception from functions it calls instead of progressing as well.

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

int convert_to_int2(char* my_str) throws (int) {
  throw 30;

  return 10;
}

int convert_to_int(char* my_str) throws (int) {
  int result = 10;
  result = convert_to_int2(my_str);

  return result;
}

int main(int argc, char **argv) {
  int result = 0;
  try {
    result = convert_to_int(argv[0]);
  } catch (int e) {
    cout << "An error occurred: " << e << endl;
  }
  //This will print out 0.
  cout << “result is: “ << result << endl;
  return 0;
}
```

Exceptions

You can throw multiple types of exceptions, which can help you determine what kind of error occurred, especially if you've made classes for your different exception types.

If you catch(…) it will be used for any exception not previous specified by the catch blocks before it.

```
class MyException1 {
  …
}

class MyException2 {
  …
}

…

try {
  // code here
}
catch (MyException1 e) { cout << “exception 1"; }
catch (MyException2 e) { cout << “exception 2"; }
catch (...) { cout << "default exception"; }
```

Exception Specifications

You can specify what exceptions a function can throw.

```
//This function can throw an exception of any type.
int f0(…);

//This function can throw an exception of type int.
int f1(…) throws (int);

//This function can throw an exception
//of type int or char.
int f2(…) throws (int, char);

//This function cannot throw an exception.
int f3(…) throws ();
```

Standard Exceptions

If you #include <exception> you can use the standard exception class:

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

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

int main () {
  try {
    throw new MyException();
  } catch (MyException *e) {
    cout << e.what() << endl;
    delete e;
  }
  return 0;
}
```

Standard Exceptions

There are also default set of exceptions that can be thrown:

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

Standard Exceptions

For example, an exception of type bad_alloc is thrown when there is some problem allocating memory:

```
try {
  int *myarray = new int[1000];

} catch (bad_alloc& e) {
  cout << "Error allocating memory:" << e.what() << endl;
}
```

Standard Exceptions

All these default exceptions are derived from the standard exception class, so we can catch them using the exception type.

Note that when an exception is caught, c++ will go down the catch list and try and match the thrown exception to the type specified. So just like you can assign an instance of a child class to a variable with its parent's type, a child exception will match to its parent's type.

```
try {
  int *myarray = new int[1000];

} catch (exception& e) {
  cout << "Exception: " << e.what() << endl;
}
```

A Helpful Exception Class

The C and C++ give some automatically defined variables, which can be very helpful for making your own exception class:

```
class MyException : public exception {
   private:
      int line;
      char* file;
   public:
      MyException(int l, char* f) : line(l), file(f) {
      }

      virtual const char* what const throw() {
          ostringstream oss;
          oss << “Exception thrown in file “ << file
              << “ on line “ << line;
          return oss.c_str();
      }
}

throw new MyException(__LINE__, __FILE__);
```