14. Error Handling

22. Handling Exceptions

Filesystem (1)

```
// Since C++17
// <filesystem>

std::filesystem::directory_iterator
// member: const std::filesystem::directory_entry& operator*() const;

std::filesystem::directory_entry
// member:     const std::filesystem::path& path() const noexcept;

std::filesystem::path
// member:     path filename() const;
//             path stem() const;
//             std::string string() const;
//             std::wstring wstring() const;
```

Filesystem (2)

```
#include <iostream>
#include <filesystem>
#include <string>
#include <io.h>
#include <fcntl.h>

int main() {
    constexpr char cp_utf16le[] = ".1200";
    setlocale(LC_ALL, cp_utf16le);
    _setmode(_fileno(stdout), _O_WTEXT);    _setmode(_fileno(stdin), _O_WTEXT);

    std::wstring folder;
    std::wcout << L"Input folder: ";
    std::wcin >> folder;
    std::wcout << L"Folder: " << folder << std::endl;

    int i = 0;
    for (const auto& entry : std::filesystem::directory_iterator(folder)) {
        if (entry.is_directory()) std::wcout << L"[Folder]" << L", ";
        std::wcout << entry.path().stem().wstring() << L", ";
        std::wcout << entry.path().filename().wstring() << L", ";
        std::wcout << entry.path().wstring() << std::endl;
    }
}
```

Lambda Function (1)

A lambda expression allows us to define an anonymous function inside another function.

```
[captureList](parameterList)mutable->returnType {body} // mutable
[captureList](parameterList)->returnType {body}        // const
[captureList](parameterList) {body} // return type or void
[captureList]{body} // no argument

//• mutable allows body to modify the captured objects by copy
//• captureList is a comma-separated list of zero or more captures
// [], [=], [&], [this], [a, &b], [=, &a] [a, b, &c], [&, a],
// [this] → reference
//• parameterList is a comma-separated list of parameters
//• returnType is the type of the result the function returns
```

Lambda Function (2)

```
#include <iostream>
#include <vector>
#include <algorithm>
int main() {
   std::cout << [](int x, int y) -> int {return x + y; } (3, 2)
      << std::endl;

   auto F1 = [](auto a, auto b, auto c) {return a + b + c; };
   std::cout << F1(10, 20, 30) << std::endl;

   std::vector<int> v = { 1, 2, 3, 4, 5 };
   std::for_each(v.begin(), v.end(),
    [](int element) { std::cout << element << std::endl; });
}
// for_each: Applies the given function object f to the result of
// dereferencing every iterator in the range [first, last), in order.
// template<class InputIt, class UnaryFunction>
// UnaryFunction for_each(InputIt first,InputIt last,UnaryFunction f);
// until  C++20
```

Lambda Function (3)

```
#include <iostream>
#include <vector>
#include <algorithm>
int main() {
   int sum = 0;
   std::vector<int> vec{5, 22, 6, -3, 8, 4};
   std::for_each(std::begin(vec), std::end(vec),
      [&sum](int x) { sum += x; });

   std::cout << "The sum is " << sum << '\n';
}
```

Lambda Function (4)

```
#include <iostream>
int main() {
   int x = 1, y = 2;
   auto print0 = [=] { std::cout << x << " , " << y << std::endl;};
   auto print1 = [=] () mutable { std::cout << ++x << " , " << ++y
      << std::endl;};
   auto print2 = [&] { std::cout << x++ << " , " << y++ << std::endl;};
   auto print3 = [=, &y] {std::cout << x << " , " << y++ << std::endl;};

   print0();    // 1, 2
   std::cout << "after: " << x << " , " << y << std::endl;  // 1, 2
   x = 1; y = 2;        print1();       // 2, 3
   std::cout << "after: " << x << " , " << y << std::endl;  // 1, 2
   x = 1; y = 2;        print2();       // 1, 2
   std::cout << "after: " << x << " , " << y << std::endl;  // 2, 3
   x = 1; y = 2;        print3();       // 1, 2
   std::cout << "after: " << x << " , " << y << std::endl;  // 1, 3
   return 0;
}
```

Lambda Function (5)

```
#include <iostream>
#include <functional>
// std::function - general polymorphic function wrapper <Rtype(Args)>
std::function<double(double)> derivative
   (std::function<double(double)> f, double h) {
   return [f, h](double x) { return (f(x + h) - f(x)) / h; };
}
double fun1(double x) {  return 3*x*x + 2*x +5; }
double fun2(double x) {  return 3*x + 5; }

int main() {
   double h = 0.00001;
   auto der0 = derivative(fun1, h);
   std::cout << der0(5.) << std::endl;

   auto der1 = derivative(fun2, h);
   std::cout << der1(5.) << std::endl;
}
```

Namespaces (1)

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
namespace vec_utils {
   int max(const std::vector<int>& vec) {
      auto p = std::begin(vec);      int m = *p++;
      while (p != std::end(vec)) {
         if (*p > m) m = *p;
         p++;
      }
      int count = 0;
      for (auto elem : vec)  if (elem == m) count++;
      return count;
   }
}
int main() {
   std::cout << max({ 1, 2, 3, 4, 5 }) << std::endl;
   std::cout << vec_utils::max({ 1, 2, 3, 4, 5 }) << std::endl;
}
```

Namespaces (2)

```
namespace utils {
   namespace graphics {
      namespace math {
         double f(double v) {
            /* Details omitted . . . */
         }
      }
   }
}
```

Exception Motivation

```
// The at method works just like operator[], except at does check
// the vector's bounds. If i >= v.size() or i < 0, the expression
// v.at(i) represents an exceptional situation, and we say
// the vector's at method throws, or raises, an exception.


v.at(i) = 4;

if (0 <= i && i < v.size()) // Ensure i is in range
   v[i] = 4;
else
   …
```

Exception Examples (1)

```
#include <iostream>
#include <vector>
int main() {
   std::vector<double> nums{ 1.0, 2.0, 3.0 };
   int input;
   std::cout << "Enter an index: ";
   std::cin >> input;

   try {
      std::cout << nums.at(input) << '\n';
   }
   catch (std::exception& e) {
      std::cout << e.what() << '\n';
   }
}
```

Exception Examples (2)

```cpp
#include <iostream>
#include <vector>
int main() {
   std::vector<double> nums { 1.0, 2.0, 3.0 };
   int input;
   while (true) {
      std::cout << "Enter an index: ";
      std::cin >> input;
      try {
         std::cout << nums.at(input) << '\n';
         break; // Printed successfully, so break out of loop
      }
      catch (std::exception&) {
         std::cout << "Index is out of range. Please try again.\n";
      }
   }
}
```

Exception Examples (3)

```cpp
#include <iostream>
#include <vector>
int main() {
   double n1, n2, n;
   std::cin >> n1 >> n2;

   try {
      if(n2 == 0)throw n2; // exception
      n = n1/n2;
   }
   catch(double &e) {
      n = 0;
      std::cout << "n2 = " << e << std::endl;
      std::cout << "n <= " << n << std::endl;
   }

   std::cout << "n = " << n << std::endl;
}
```

Exception Examples (4)

```
#include <iostream>
int main() {
   int input = 0, sum = 0;                   // set exceptions mask
   std::cin.exceptions(std::istream::badbit | std::istream::failbit);
   std::cout << "Please enter integers to sum, 999 ends list: ";
   while (input != 999) {
      try {
         std::cin >> input;
         if (input != 999) sum += input;
      }
      catch (std::exception& e) {
        std::cout << "****Non-integer input detected\n";
        std::cin.clear(); //std::cout << e.what() << '\n';
        std::cin.ignore(
           std::numeric_limits<std::streamsize>::max(),'\n');
      }
   }
   std::cout << "Sum = " << sum << '\n';
}
```

Propagation Exception

```
#include <iostream>

void F(int n){
   if(n == 0) throw n;
   std::cout << "F: " << n << std::endl;
}

int main() {
   int n;
   std::cin >> n;

   try {
      F(n);
   }
   catch (int &e) {
      std::cout << "Error " << e << std::endl;
   }
}
```

Custom Exception

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
class FileNotFoundException : public std::exception {
   std::string message;
public:
   FileNotFoundException(const std::string& fname):
      message("File \"" + fname + "\" not found") {}
   const char *what() const { return message.c_str(); }
};
std::vector<int> load_vector(const std::string& filename) {
   std::ifstream fin(filename);
   if (fin.good()) { std::vector<int> result; // …
                     return result;}
   else throw FileNotFoundException(filename);
}
int main() {
   try {
      std::vector<int> numbers = load_vector("values.data");
      // …
   }
   catch (std::exception& e) { std::cout << e.what() << '\n'; }
}
```

```
virtual const char* what() const noexcept; (since C++11)
```

Catching Multiple Exceptions (1)

```
class Exception1 : public std::exception {
 // …
};
class Exception2 : public std::exception {
 // …
};


if(/*…*/) throw Exception1();
else if(/*…*/) throw Exception2();


catch (std::exception &e) {/*…*/}

catch (Exception1 &e1) {/*…*/}
catch (Exception2 &e2) {/*…*/}
```

Catching Multiple Exceptions (2)

```
catch (std::out_of_range&) {
   std::cout << "Index provided is out of range\n";
}
catch (std::invalid_argument&) {
   std::cout << "Index provided is not an integer\n";
}
catch (...) {// Catches any exceptions not caught by more specific
   std::cout << "Unknown error\n";
}
```