Generic Programming, Exception Handling, and Multiple Threads in C++

ESE2025

Generic Programming

- the term “generic programming” refers to a way of coding which is not dependent on any particular type;
- for example, the *sort* function available in the C++ algorithm library, is able to sort any container, as long as start() and end() are specified. For example,

```
#include <vector>
#include <list>
#include <algorithm

int main() {
…
        std::vector<int> my_integers;
        std::list<string> my_strings;
…
        std::sort (my_integers.begin(), my_integers.end()); // sort the integers
        std::sort (my_strings.begin(), my_strings.end());   // sort the strings
...
        return 0;
}
```

- C++ has features such containers, iterators, and algorithms which are tools to help us program more “generically”; why is this useful?

C++ containers...

Until now, we have only used the C++ std::vector container. But C++ has several containers to choose from:

- sequential containers:
  - std::vector : indexed, elements can be accessed instantly (via index)
  - std::list : linked-list like, efficient insertion or deletion anywhere
  - std::deque (pronounced “deck”, stands for “double-ended queue”)

    : indexed, a queue with efficient insert/delete available at either head or tail
- sequential-container-like:
  - std::string : (almost) like std::vector<char>
- associative containers: think (<key>, <value>) pairs
  - std::map : associative array; elements retrieved by key, efficient lookup & retrieval
  - others (not covered here: std::set, std::multimap, std::multiset)

iterators

every container offers an iterator mechanism that makes coding more generic…

e.g.,

```
int      load_arr[50]; // storage for our integers
…

std::vector<int> integer_data;
std::vector<int>::iterator iter;
…

iter = integer_data.begin();
size_t i = 0;
while (i != 50)
{
         *iter.push_back() = load_arr[i];
         ++i;
}

...
```

Multiple Threads in C++

```
// thread example
#include <iostream>       // std::cout
#include <thread>         // std::thread

void foo()
{
  // do stuff...
}

void bar(int x)
{
  // do stuff...
}

int main()
{
  std::thread first (foo);     // spawn new thread that calls foo()
  std::thread second (bar,0);  // spawn new thread that calls bar(0)

  std::cout << "main, foo and bar now execute concurrently...\n";

  // synchronize threads:
  first.join();                // pauses until first finishes
  second.join();               // pauses until second finishes

  std::cout << "foo and bar completed.\n";

  return 0;
}
```

Exceptions Revisited

next time