C++ Continued: Templated Functions, Stack Unwinding, and More Threads (critical section and semaphores)

ESE 2025

Template Definitions

- the following is taken largely from *C++ Primer*, Fourth Ed., by Lippman et al.
- template definitions offer another means of generic programming, that is, allowing us to write code that is “type universal”
- “... imaging that we want to write a function to compare two values and indicate whether the first is less than, equal to, or greater than the second ... [We could] define several overloaded functions:”

```
int compare (const string &v1, const string &v2)
{
        if (v1 < v2) return -1;
        if (v1 > v2) return 1;
        return 0;
}
```

```
int compare (const double &v1, const double &v2)
{
        if (v1 < v2) return -1;
        if (v1 > v2) return 1;
        return 0;
}
```

Template Definitions Cont'd

- "Having to repeat the body of a function for each type we compare is tedious and error-prone. More importantly, we need to know *in advance* all the types that might ever want to compare."
- "Rather than defining a new function for each type, we can define a single **function template**."

```
template <typename T>
int compare (const T &v1, const T &v2)
{
        if (v1 < v2) return -1;
        if (v1 > v2) return 1;
        return 0;
}
```

Class Templates

- consider an example of defining a Queue class; what if we want a queue of strings or a queue of integers? We should be able to define a queue **regardless of the type** of the elements in the queue...

```
template <class Type> class Queue
{
        public:
                Queue();              // default constructor
                Type &front();        // return element from head of Queue
                const Type &front() const;     // front will not change data members of object
                void push (const Type &);      // add element to back of Queue
                void pop();                    // remove element from head of Queue
                bool empty()  const;           // true if no elements in the Queue
        private:
                // ...
}
```

Class Templates Cont’d

- to use the class template, we must specify the type parameters (which was not needed in the function template):

```
Queue<int> qi;     // Queue that holds ints

Queue< vector<double> > qc; // Queue that holds vectors of doubles

Queue<string> qs; // Queue that holds strings
```

RECALL: Class Definitions and Declarations

```
class Sales_item {
public:
        // operations on Sales_item objects
        double avg_price () const;
        bool same_isbn (const Sales_item &rhs) const
                { return isbn == rhs.isbn; }
        // default constructor needed to initialize
        // members of built-in type:
        Sales_item(): units_sold(0), revenue(0.0) { }
private:
        std::string isbn;
        unsigned units_sold;
        double revenue;
};
```

- we also need any functions not defined within the class definition:

```
double Sales_item::avg_price() const
{
        if (units_sold)
                return revenue/units_sold;
        else
                return 0;
}
```

- members of a class can be either data, functions or type definitions

Exceptions again...

- *exceptions* are run-time anomalies, such as running out of memory or encountering unexpected input; think of exceptions in C++ as involving both **error detection** and **error handling**.

- in C++, exception handling involves **throw expressions** (error-detecting code indicates that an error has occurred; a throw raises an exceptional condition), **try blocks** (which the error-handing code uses to deal with the exception; recall that a try block contains **catch** clauses), and a set of **exception classes** (used to pass information about an error between a throw and an associated catch).

Exceptions Cont’d

- recall how a throw and catch might work together:

```
try {
        // ...
        throw runtime_error(“Data must refer to the same ISBN code”);
        // ...
} catch (runtime_error err) {
        cout   << err.what()
                << “\n Try Again? Enter y or n” << endl;
        char c;
        cin >> c;
        if (cin && c == ‘n’)
                // do something
}
```

Data must refer to the same ISBN code
Try Again? Enter y or n

- the **exception** header defines the most general kind of exception class (of the same name); it communicates only that an exception has occurred but no additional information
- the **stdexcept** header defines several general-purpose exception classes
- you can also define your own exception-derived class:

```
class myownexception: public std::runtime_error {
        public:        explicit myownexception(const std::string &s):
                                        std::runtime_error(s) {  }
};
```

Exceptions: Stack Unwinding

- “when an **exception** is thrown, execution of the current function is suspended and the search begins for a matching **catch** clause ...
- the search starts by checking whether the **throw** itself is located inside a **try** block …
- if so, the catch clauses associated with this try are examined for a match, and if, so, the exception gets handled ...
- if no catch is found, the current function is exited--- its memory freed and local objects are destroyed--- and the search continues within the *calling* function …
- this process continues, up the the chain of nested function calls until a catch clause for the exception is found!
- as soon as the the exception is handled, execution continues immediately after the last catch clause...

Back to Threads...

- critical sections
  - portions of thread execution in which a context switch (to another thread) is not allowed
  - example:

```
void print_thread_id (int id)
{
        std::unique_lock<std::mutex> lck (mtx,std::defer_lock);

        // critical section (exclusive access to std::cout signaled by locking lck):
        lck.lock();
        std::cout << "thread #" << id << '\n';
        lck.unlock();
}
```

- semaphores
  - similar to a mutex, but allow more than just one thread “in the bathroom”
  - thus semaphores have counters or queues of fixed sizes, that keep track of how many threads have been admitted
  - in C++, no explicit semaphore type (other than a mutex), but you can create your own

thread condition variables, via an example

```cpp
#include <iostream>
#include <string>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;

void worker_thread()
{
    // Wait until main() sends data
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, []{return ready;});

    // after the wait, we own the lock.
    std::cout << "Worker thread is processing data\n";
    data += " after processing";

    // Send data back to main()
    processed = true;
    std::cout << "Worker thread signals data processing completed\n";

    // Manual unlocking is done before notifying, to avoid waking up
    // the waiting thread only to block again (see notify_one for details)
    lk.unlock();
    cv.notify_one();
}
```

```cpp
int main()
{
    std::thread worker(worker_thread);

    data = "Example data";
    // send data to the worker thread
    {
        std::lock_guard<std::mutex> lk(m);
        ready = true;
        std::cout << "main() signals data ready for processing\n";
    }
    cv.notify_one();

    // wait for the worker
    {
        std::unique_lock<std::mutex> lk(m);
        cv.wait(lk, []{return processed;});
    }
    std::cout << "Back in main(), data = " << data << '\n';

    worker.join();
}
```