

# **CS3520 Programming in C++ Threads**

**Ken Baclawski  
Fall 2016**

# Outline

- Threads
  - Shared Data
  - Exclusive Locks
  - Producer-Consumer
- Midterm Review
  - Strings and Statements
  - Control Structures
  - Vectors
  - Classes and Functions
  - Iterators
  - Library Algorithms
  - Maps and Sets

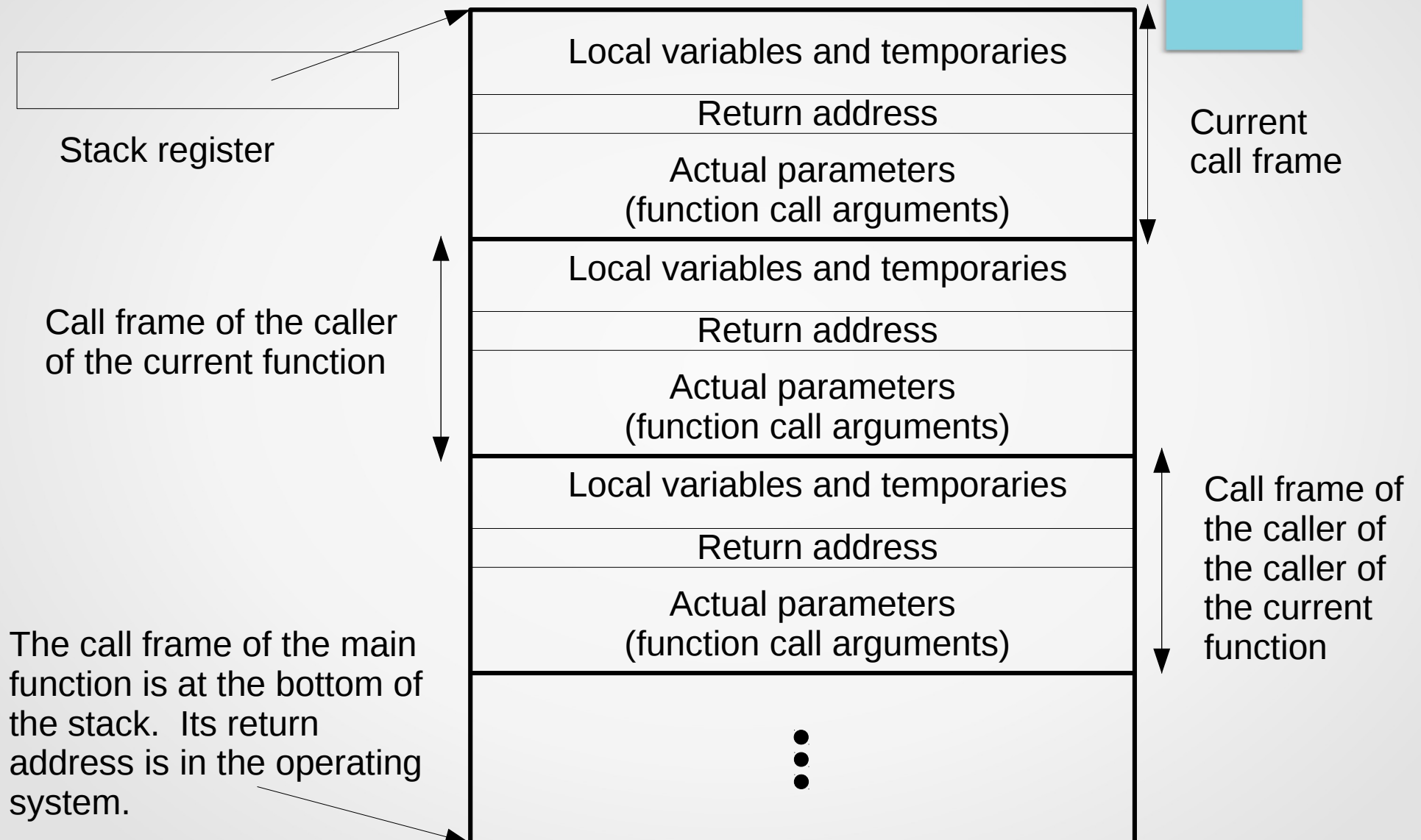
# Process

- A *process* is an instance of a program that is running on a computer
- Each process has its own independent address space
  - Processes act like independent machines
  - Communication between two processes is complicated
- A typical Unix/Linux machine will have hundreds of processes running at the same time
  - The top command will display the current processes that are the most active

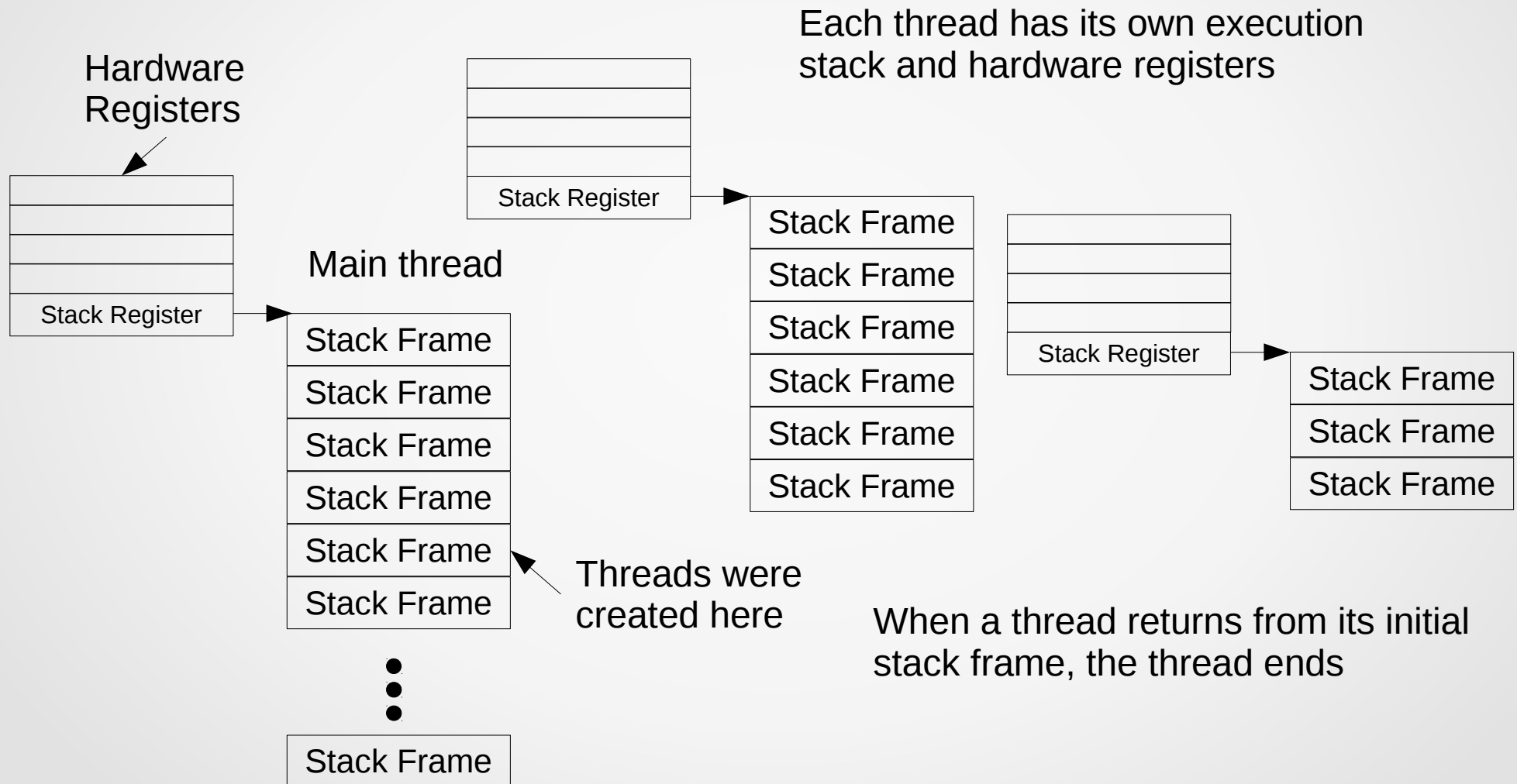
# Thread

- A *thread* is an execution sequence
  - In the same address space as the creating process
  - Has its own execution stack and registers
  - Uses far fewer resources than a process
- Each thread has its own independent local variables
- Communication between threads in a process is relatively simple compared with interprocess communication
  - However, it is still necessary to communicate properly

# Execution Stack



# Thread Execution Stacks



# Overloading Function Call Syntax

- One can even overload the function call syntax
- The operator is `operator()`
- For example, if a method is declared as:

```
void operator()(const string& input);
```

Then the method can be called like this:

```
object("Hello");
```

where `object` is an object of the class.

The full syntax for this call is:

```
object.operator()("Hello");
```

# Creating threads

- Declaring a thread and initializing it immediately starts a new thread
- The first argument is an object
- The method that is executed is `operator()`
- The arguments are passed to the method and the method called in a new thread

```
thread t(object, arg1,  
         arg2, ...);
```

is the same as this statement

```
object(arg1, arg2, ...);
```

but running in its own thread



# Enabling Multithreading

- Threads must be *enabled*
- This requires the `-pthread` option on the command line for compilation

```
g++ -Wall -std=c++11 *.cpp -o main -pthread
```

# Ending threads

- Threads run independently
  - Must terminate on their own
  - A thread cannot be forced to stop non-cooperatively
- The creating thread can wait for a thread to end or can detach the thread
- If an undetached thread is destructed, then the *entire process* is terminated

A creating thread waits for a thread to finish by calling the `join` method

```
t.join();
```

A creating thread can detach a thread with the `detach` method

```
t.detach();
```

# Shared Data

- Data that is accessed by more than one thread is called *shared data*.
- Accessing shared data incorrectly can result in corrupted data and is very difficult to debug.
- The only reliable technique for avoiding shared data errors is to use careful design of shared data.

# Lost Update Error

Two uncoordinated threads are attempting to deposit \$100 in a bank account. Both of these execution traces could occur:

Thread 1	Thread 2	Balance
		100
Read balance		100
Add 100		100
Write new value		200
	Read balance	200
	Add 100	200
	Write new value	300

Thread 1	Thread 2	Balance
		100
Read balance		100
	Read balance	100
Add 100		100
	Add 100	100
Write new value		200
	Write new value	200

The balance is shared data. The shared data is not being accessed correctly. To access the shared data correctly, it must be *locked* before any accesses.

# Locking

- A *lock* is a mechanism for synchronizing access to shared data
  - A lock is always on a specific object
  - Locks on different objects have no effect on each other
- An object that can be locked is called a `mutex` in C++
  - The name comes from “mutual exclusion”
  - However, the term “mutex” is also used for non-exclusive
  - It is better to regard a mutex as a *lockable object*
- Declare a lockable object like this:

```
std::mutex lockableObject_;
```

# Kinds of Lock

- Exclusive locks
  - `lock_guard`
    - Can only be locked once
    - Slightly more efficient
  - `unique_lock`
    - Can be unlocked and relocked
    - More flexible
    - Required for some uses such as notifiers
- Shared locks
  - `shared_lock` (since C++14)

# Lock Management

- A `mutex` can be locked and unlocked directly
  - `lock()` acquires the lock or waits if it is not available
  - `try_lock()` acquires the lock if available and does not wait
  - `unlock()` releases the lock
- These methods must *never* be used!
- Failing to unlock a locked object is a difficult error to find and correct
  - It does not manifest itself in an obvious way
  - Difficult to ensure that the object is always unlocked
- `lock_guard` and `unique_lock` ensure that this will never happen
  - The lock can be explicitly released
  - If it has not been released, the lock is released when the lock goes out of scope

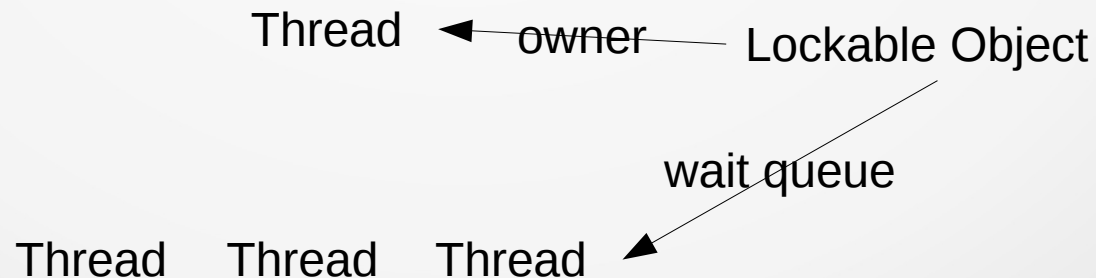
# Using locks

```
{  
    // Lock the object by constructing a lock.  
  
    lock_guard<mutex> lock(lockableObject_);  
  
    // We now have exclusive access to the contents.  
  
    contents_.push_back(text);  
  
    // When the lock goes out of scope, the lock is  
    // released.  
}
```



# Lock implementation

- A lockable object has a wait queue
- Threads are placed in the wait queue if there is a conflicting lock
- If the thread is at the head of the queue and the lock is released, then the thread is given the lock





# The dumpster package

# Requirements

- A dumpster is a shared container
  - One can dump a string in the dumpster
  - One cannot take the string out
  - The current contents of the dumpster can be observed
- An action object encapsulates the execution code and associated data for a thread
- The main program creates three action objects and four threads
  - It waits for the threads to finish and then prints a conclusion

# Dumpster Header

```
#ifndef DUMPSTER_DUMPSTER_H
#define DUMPSTER_DUMPSTER_H

#include <string>
#include <mutex> ← Library for locks
#include <vector>

namespace dumpster {

/**
 * A dumpster is a thread-safe container that
 * checks in strings but never releases them. The
 * collection of strings dumped in the dumpster is
 * a vector, a copy of which can be retrieved.
 *
 * @author Ken Baclawski
 */
class Dumpster {
```

# Dumpster Header

```
public:
    /**
     * Check a string into the dumpster.
     */
    void dump(** The string to be dumped into the dumpster. */
              const std::string& object) noexcept;

    /**
     * Get the strings that have been dumped into
     * the dumpster.
     * @return A copy of the list of strings in the
     * dumpster.
     */
    std::vector<std::string> getContents() noexcept;
```

Cannot be const because it will be locking the lockable object

# Dumpster Header

```
private:
    /**
     * The list of objects that were dumped in the
     * dumpster.
     */
    std::vector<std::string> contents_;

    /**
     * The lockable object used for setting locks.
     */
    std::mutex lockableObject_;
};

}

#endif
```

← A mutex is an object that can be locked

# Dumpster Source

```
#include <string>
#include <iostream>
#include <vector>
#include "Dumpster.h"


using namespace std;
using namespace dumpster;

void Dumpster::dump(const string& text) noexcept {
    // Lock the object by constructing a lock.

    lock_guard<mutex> lock(lockableObject_);

    // We now have exclusive access to the contents.
    contents_.push_back(text);


    // When the lock goes out of scope, the lock is released.
}
```



This is the "critical section"

# Dumpster Source

```
vector<string> Dumpster::getContents() noexcept {  
    // Lock the object by constructing a lock.  
  
    lock_guard<mutex> lock(lockableObject_);  
  
    // Show the current contents.  
  
    cout << "The current contents of the dumpster is:";  
    for (const string& text : contents_) {  
        cout << " \"" << text << "\"";  
    }  
    cout << endl;  
  
    // This copies the list and returns the copy.  
  
    return contents_;  
  
    // When the lock goes out of scope, the lock is released.  
}
```



This is the  
“critical  
section”



# Action Header

```
#ifndef DUMPSTER_ACTION_H
#define DUMPSTER_ACTION_H

#include <string>
#include <vector>
#include "Dumpster.h"

namespace dumpster {

/**
 * An action object for a thread.  An action
 * object specifies the execution code and
 * associated data for a thread.
 *
 * @author Ken Baclawski
 */
class Action {
```

# Action Header

```
public:
    /**
     * Construct an action object.
     */
    Action(** The label of the action. */
           const std::string& label,
           /** The dumpster. */
           Dumpster& dumpster);

    /**
     * The procedure that this action runs. It
     * dumps its label in the dumpster, and
     * retrieves a copy of the current contents of
     * the dumpster.
     */
    void operator()() noexcept;
```

The dumpster  
will be modified  
so it must be a  
reference

In Java this class  
would be a  
Runnable class

# Action Header

```
private:
    /**
     * The label of the action.
     */
    const std::string label_;

    /**
     * The dumpster.
     */
    Dumpster& dumpster_;

    /**
     * The dumpster contents.
     */
    std::vector<std::string> contents_;
};

#endif
```

Unshared data

Shared data

# Action Source

```
#include <string>
#include "Action.h"

using namespace std;
using namespace dumpster;

Action::Action(const string& label, Dumpster& dumpster)
    : label_(label), dumpster_(dumpster) {}

void Action::operator>()() noexcept {

    // Dump the action label in the dumpster.

    dumpster_.dump(label_);

    // Retrieve the dumpster contents.

    contents_ = dumpster_.getContents();
}
```

# Main

```
#include <string>
#include <iostream>
#include <stdexcept>
#include <thread>
#include "Dumpster.h"
#include "Action.h"

using namespace std;
using namespace dumpster;

/**
 * @namespace dumpster This package illustrates
 * the how to create and run threads.
 *
 * @author Ken Baclawski
 */
```

Thread library

# Main

```
/**
 * Thread test program.  Four threads are started,
 * each of which dumps a string in the dumpster
 * and retrieves the current contents of the
 * dumpster.  When all of the threads have
 * finished, a conclusion is printed.
 */
int main() {

    // Construct the dumpster.

    Dumpster dumpster;
```

# Main

```
// Construct three action objects.  
  
Action a1("First", dumpster);  
Action a2("Second", dumpster);  
Action a3("Third", dumpster);  
  
// Start up the threads.  
  
thread t1(a1);  
thread t2(a2);  
thread t3(a3);  
  
// The last thread uses the first action object  
// again. This does not affect the first action  
// object because the thread constructor copies  
// the action argument.  
  
thread t4(a1);
```

# Main

```
// Wait until the threads have finished.
```

```
try {  
    t1.join();  
    t2.join();  
    t3.join();  
    t4.join();  
} catch (const exception& e) {  
    cout << "The main thread could not wait because of "  
        << e.what() << endl;  
}
```

The word "join" means "wait for the thread to finish."

```
// Concluding remarks
```

```
cout << "All the threads have finished" << endl;
```

```
// Return normal status
```

```
return 0;
```

```
}
```



# Output of Main

- The four threads are constructed and started
- Threads t1 and t4 dump their labels first
- Then t3 dumps its label
- Finally t2 dumps its label
- The main thread waits for the threads to finish and then prints a conclusion
- Running the program again will probably produce different output

```
The current contents of the dumpster is: "First"  
The current contents of the dumpster is: "First" "First"  
The current contents of the dumpster is: "First" "First" "Third"  
The current contents of the dumpster is: "First" "First" "Third" "Second"  
All the threads have finished
```

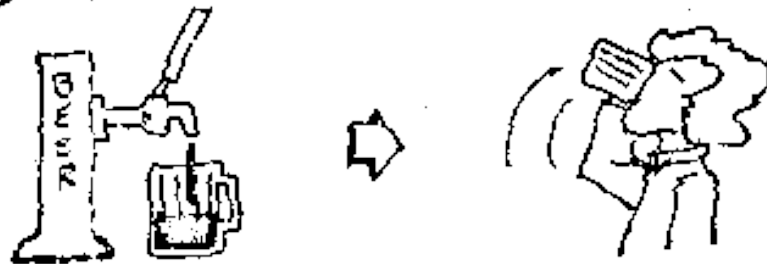


# Producer-Consumer Design Pattern

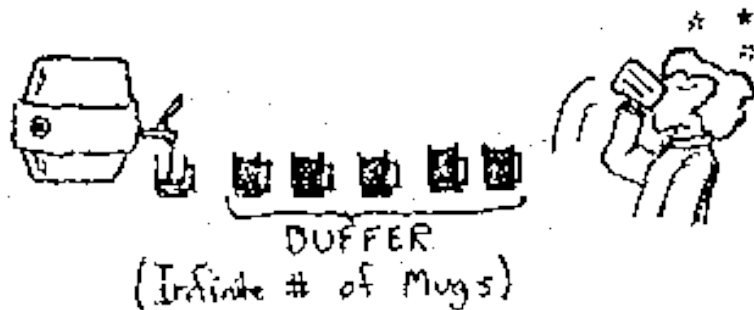
# A GRAPHIC EXAMPLE OF THE PRODUCER/CONSUMER PROBLEM

Michael Vignaux

① PRODUCER CONSUMER



②



③ PROBLEM -

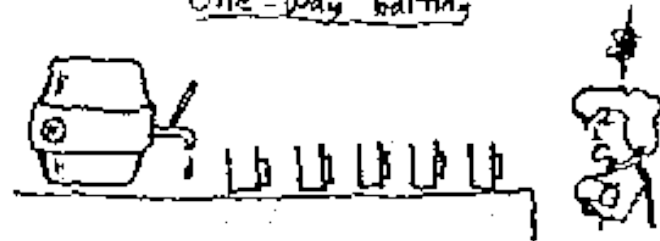


Consumer takes from buffer before producer is done adding to it - trouble!  
This is solved by \_\_\_\_\_

(fill in the blank)

④

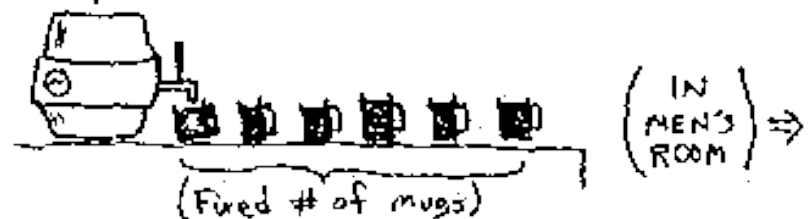
One-way halting



The consumer must wait for producer to produce before it can consume...

⑤

BOUNDED BUFFER

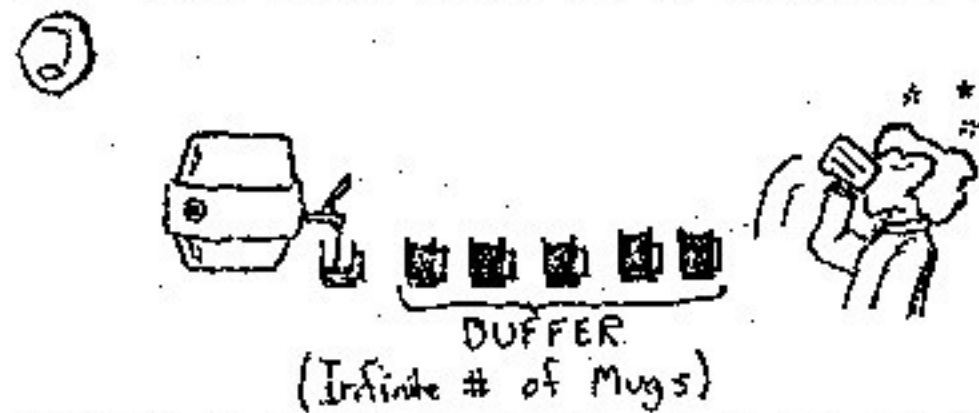


If the consumer is busy (can't consume), the producer must wait, if the buffer is full, for the consumer to start consuming again. The processes are now \_\_\_\_\_  
(fill in)

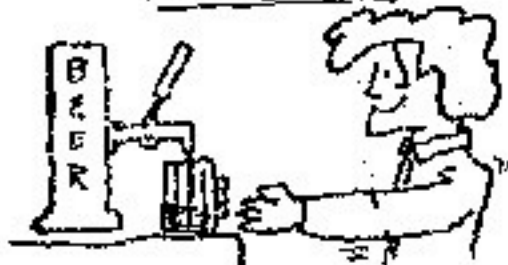
# A GRAPHIC EXAMPLE OF THE PRODUCER/CONSUMER PROBLEM

Michael Vignaux

① PRODUCER CONSUMER



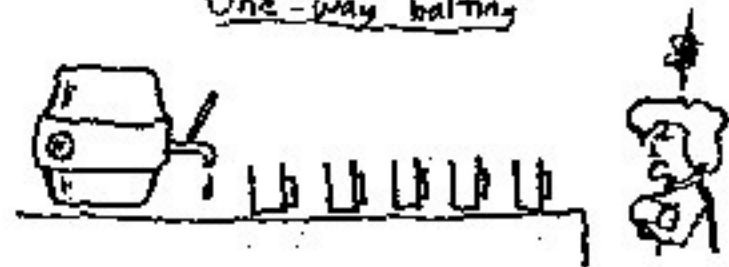
③ PROBLEM -



Consumer takes from buffer before producer is done adding to it - trouble!  
This is solved by \_\_\_\_\_

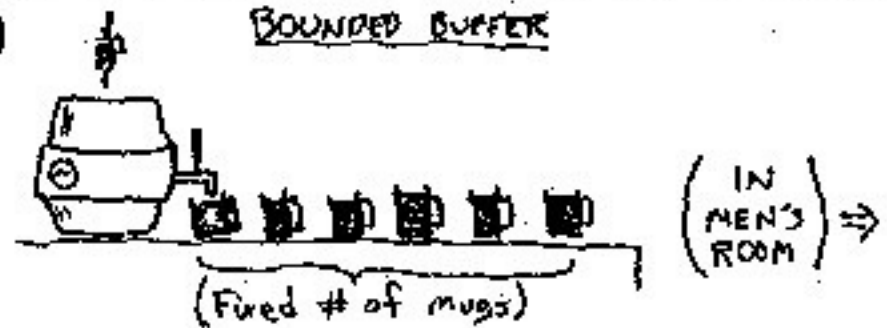
(fill in the blank)

④ One-way halting



The consumer must wait for producer to produce before it can consume...

⑤ BOUNDED BUFFER



If the consumer is busy (can't consume), the producer must wait, if the buffer is full, for the consumer to start consuming again. The processes are now \_\_\_\_\_  
(fill in)

# Producer-Consumer Design Pattern

- Very popular and versatile technique for sharing data among threads
- A container called a *buffer* is shared among threads
- Objects in the buffer are transferred from one thread to another
  - The objects might be tasks or reports other kinds of object
- A thread can *produce* and *consume* objects
  - Produce adds another object to the buffer
  - Consume removes an object from the buffer
- The buffer is either unbounded or bounded (limited size)

# Producer-Consumer Communication

- How can a consumer find out that an object is available to be consumed?
- Periodic polling is either very inefficient or has poor response time
- Directly sending a message also does not work
  - The consumer would still need to use polling
  - There will be many potential consumers
  - Every producer would be responsible for maintaining a list of potential consumers and would have to notify all of them whether they are waiting or not

# Producer-Consumer Communication

- The solution is to use a *notifier*
- The name for a notifier in C++ is `condition_variable`
- Advantages
  - Very efficient: no polling required
  - Very responsive: consumers are immediately notified



# The shared1 package



# Requirements

- A shared stack is a type-safe container
  - One can *produce* an object which is stored in the container
  - One can *consume* an object taken from the container
  - If the container is empty, the consumer waits for an object to be produced by some other thread
- The action object either produces or consumes a string
- The main program creates two action objects and four threads
  - It waits for the threads to finish and then prints a conclusion

# Action Header

```
#ifndef SHARED1_ACTION_H
#define SHARED1_ACTION_H

#include <string>
#include "SharedStack.h"

namespace shared1 {

/**
 * An action object for a thread.  An action
 * object specifies the execution code and
 * associated data for a thread.
 *
 * @author Ken Baclawski
 */
class Action {
```

Differences with the  
dumpster Action  
class are in boldface

# Action Header

```
public:
    /**
     * Construct an action object.
     */
    Action(** What the action should do.
            It should be produce or consume. */
           const std::string& activity,
           /** The shared stack. */
           SharedStack& stack);

    /**
     * The procedure that this action runs. It
     * performs either a produce or a consume
     * operation.
     */
    void operator()() noexcept;
```

# Action Header

```
private:
    /**
     * The activity to be performed.
     */
    const std::string activity_; ← Unshared data

    /**
     * The shared stack.
     */
    SharedStack& stack_; ← Shared data
};

}

#endif
```

# Action Source

```
#include <string>
#include "Action.h"

using namespace std;
using namespace shared1;

Action::Action(const string& activity, SharedStack& stack)
    : activity_(activity), stack_(stack) {}

void Action::operator>()() noexcept {

    // Perform the requested activity.

    if (activity_ == "produce") {
        stack_.produce("item");
    } else if (activity_ == "consume") {
        stack_.consume();
    }
}
```

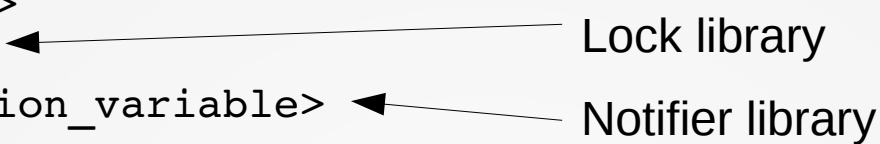
# SharedStack Header

```
#ifndef SHARED1_SHAREDSTACK_H
#define SHARED1_SHAREDSTACK_H

#include <string>
#include <mutex>
#include <condition_variable>
#include <stack>

namespace shared1 {

/**
 * A shared stack is a thread-safe stack that
 * pushes and pops strings into and out of a
 * stack. It differs from an ordinary stack not
 * only because it is thread-safe but also because
 * the pop operation never fails. If the stack is
 * empty, a thread that is calling the pop method
 * will wait until a string is pushed onto the
 * stack.
 *
 * @author Ken Baclawski
 */
class SharedStack {
```



Lock library

Notifier library

# SharedStack Header

```
public:
    /**
     * Push a string onto the shared stack.
     */
    void produce(** The string to be pushed onto the stack. */
               const std::string& text) noexcept;

    /**
     * Pop the top string off the stack if there is one.
     * Otherwise, wait until there is a string to be popped.
     *
     * @return The string on the top of the stack
     * that was popped.
     */
    const std::string& consume() noexcept;
```

# SharedStack Header

```
private:
    /**
     * The stack of strings.
     */
    std::stack<std::string> stack_;

    /**
     * The lockable object used for setting locks.
     */
    std::mutex lockableObject_;

    /**
     * The notifier for the producer-consumer
     * design pattern.
     */
    std::condition_variable notifier_;
};
#endif
```




# SharedStack Source

```
#include <string>
#include <stack>
#include <mutex>
#include <condition_variable>
#include <iostream>
#include "SharedStack.h"
```

```
using namespace std;
using namespace shared1;
```

These are redundant because they are in SharedStack.h



# SharedStack Source

```
void SharedStack::produce(const string& text) noexcept {
```

```
    // Lock the object.
```

```
    unique_lock<mutex> lock(lockableObject_);
```

```
    // We now have exclusive access to the stack.
```

```
    stack_.push(text);
```

```
    // Report the size of the stack.
```

```
    cout << "The stack has been pushed and now has "
          << stack_.size() << " item(s)" << endl;
```

```
    // Release the lock.
```

```
    lock.unlock();
```

```
    // Notify a waiting thread that the stack is nonempty.
    // This must be done after the lock is released.
```

```
    notifier_.notify_one();
```

```
}
```

A lock\_guard could have been used here

This is the  
"critical section"

The only threads that  
could be waiting are  
consumers

# SharedStack Source

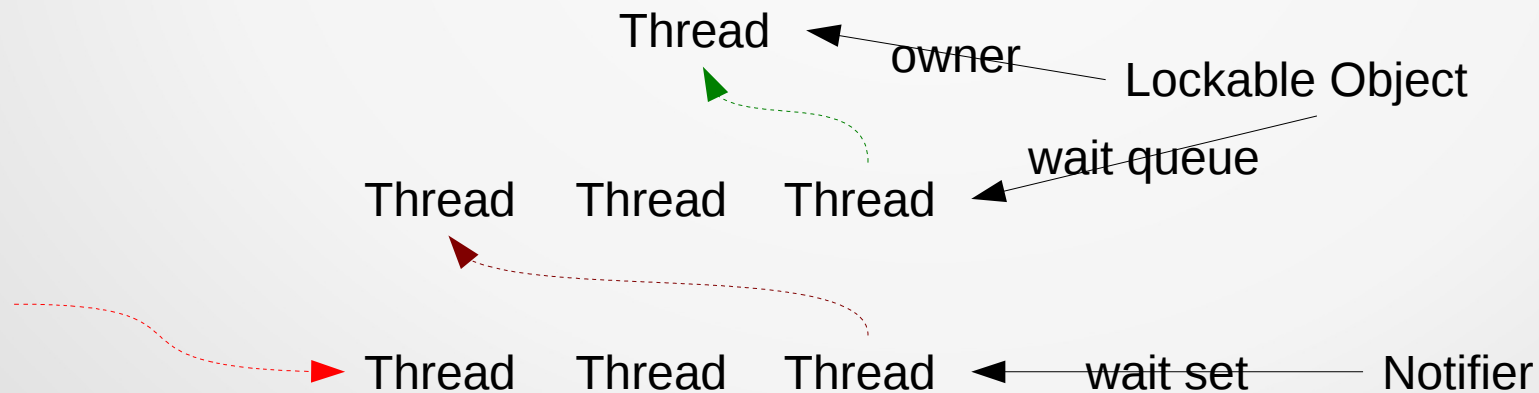
```
#const string& SharedStack::consume() noexcept {  
    // Lock the object.  
    unique_lock<mutex> lock(lockableObject_);  
    // We now have exclusive access to the stack.  
    while (stack_.empty()) {  
        notifier_.wait(lock);  
    }  
    // The stack is nonempty, so it can be popped.  
    const string& top = stack_.top();  
    stack_.pop();  
    // Report the size of the stack.  
    cout << "The stack has been popped and now has "  
        << stack_.size() << " item(s)" << endl;  
    // The lock is released when it goes out of scope.  
    return top;  
}
```

What happens here is discussed on the next slide

The “critical section” is in two parts.

# How a notifier works

- The call to wait(lock) causes
  - The lock is released
  - The thread waits until it is notified by another thread
  - The thread is added to the wait queue of the lockable object
  - When the thread obtains the lock it starts running again
- The notifier wait set is not necessarily a queue



# Spurious notifications

- After the consumer thread wakes up, it must check whether there is a string in the stack
  - A thread can be woken up for other reasons
  - This is called a *spurious wakeup*
- The best way to deal with spurious wakeup is to check the required condition in a while loop

# Main

```
#include <string>
#include <iostream>
#include <stdexcept>
#include <thread>
#include <chrono> ← Time library is explained later
#include "SharedStack.h"
#include "Action.h"

using namespace std;
using namespace shared1;

/**
 * @namespace shared1 This package illustrates the
 * producer-consumer design pattern with an
 * unbounded container.
 *
 * @author Ken Baclawski
 */
```

# Main

```
/**
 * Producer-consumer test program.  Various
 * threads are created for pushing and popping the
 * shared stack.
 */
int main() {

    // Construct the shared stack.

    SharedStack sharedStack;

    // Construct two action objects.

    Action produce("produce", sharedStack);
    Action consume("consume", sharedStack);
```

# Main

```
// Start up the threads and pause.  
  
thread t1(consume);  
thread t2(consume);  
thread t3(produce);  
this_thread::sleep_for(chrono::seconds(1));  
  
// Report that we are about to go to sleep for a  
// longer time.  
  
cout << "Going to sleep for a while..." << endl;  
this_thread::sleep_for(chrono::seconds(3));  
  
// Now create one more thread.  
  
thread t4(produce);
```

Pause for one second. This helps prevent the output of the action objects from getting mixed with the output of the threads



# Chrono namespace

- This is a namespace within `std` that includes a wide variety of time-related functions
- A `duration` is a span of time
  - `seconds` is the special case of duration in which the unit of time is seconds
  - `hours`, `minutes`, `milliseconds`, ... are other special cases
  - You can create your own special case
- A `time_point` is an instant of time

# this\_thread namespace

- This is a namespace within `std` that includes some functions that access the thread that calls the functions
- `sleep_for` pauses for a duration of time
- `sleep_until` pauses until a specified point in time
- `yield` allows another thread to run
- `get_id` is the thread identifier

# The rest of Main

```
// Wait until the threads have finished.

try {
    t1.join();
    t2.join();
    t3.join();
    t4.join();
} catch (const exception& e) {
    cout << "The main thread could not wait because of "
         << e.what() << endl;
}

// Concluding remarks

cout << "All the threads have finished" << endl;

// Return normal status

return 0;
}
```

# Output of Main

- Threads t1 and t2 start first, but both want to consume, so they must wait
- Thread t3 produces
- Either t1 or t2 consumes what t3 produced and the other one continues to wait
- The program sleeps for a while
- Thread t4 produces
- Whichever of t1 or t2 is still waiting now consumes
- The main program waits for all of the threads to finish and then prints the conclusion

```
An item has been produced and there are now 1 item(s)
An item has been consumed and there are now 0 item(s)
Going to sleep for a while...
An item has been produced and there are now 1 item(s)
An item has been consumed and there are now 0 item(s)
All the threads have finished
```



# The shared2 package

# Requirements

- Same as the shared1 package but now the stack has a limited size
- Unlike the unbounded case, both producers and consumers may have to wait
  - The consumer may wait for an object in the buffer
  - The producer may wait for available space in the buffer
- The following will show only the differences between shared1 and shared2

# SharedStack Header

- There is now a constructor that specifies the bound on how many strings can be stored in the stack
- There is a data member with the bound specified in the constructor

```
/**
 * Construct a shared stack with a specified maximum size.
 */
SharedStack(unsigned int bound);
...
private:
...
/**
 * The bound on the number of strings allowed in
 * the stack.
 */
const std::stack<std::string>::size_type bound_ = 0;
```

# SharedStack Source

```
void SharedStack::produce(const string& text) noexcept {  
  
    // Lock the object.  
  
    unique_lock<mutex> lock(lockableObject_);  
  
    // We now have exclusive access to the stack.  
    // Wait until the stack has room.  
  
    while (stack_.size() >= bound_) {  
        notifier_.wait(lock);  
    }  
    stack_.push(text);  
  
    // Report the size of the stack.  
  
    cout << "An item has been produced and there are now "  
        << stack_.size() << " item(s)" << endl;
```



# SharedStack Source

```
// Report the size of the stack.

cout << "An item has been produced and there are now "
      << stack_.size() << " item(s)" << endl;

// Release the lock.

lock.unlock();

// Notify all waiting threads that the stack is
// nonempty. This must be done after the lock
// is released.

notifier_.notify_all();
}
```

# SharedStack Source

- The `consume` method is the same except for using `notify_all` rather than `notify_one`
- The `notify_all` method wakes up all of the waiting threads
  - They now compete to be able to lock the object and either produce or consume
  - The while loop is essential, as nearly all of the notified threads will have to continue waiting
- Why not use `notify_one`?
  - In theory, both producers and consumers could be waiting



# **Review for MidTerm Exam**

# Logistics

- Open book/open notes
- 1.5 hour time limit
- Late penalty: 1 point/minute
- Laptops may be used only with prior permission
- Paper on which to write your answers will be distributed
  - You can use your own paper if you prefer

# Outline

- Strings and Statements
- Control Structures
- Vectors
- Classes and Functions
- Iterators
- Library Algorithms
- Exam Problems
  - Sample Question
  - Sample Solution
- Missing code may include:
  - C++ code
  - doxygen comments
  - ordinary comments

# Strings and Statements

- Documentation
  - doxygen
  - ordinary comments
- Includes
- Variables
- Standard I/O streams
- Const variables
- Statements
- Strings
  - Constructors
  - Operators
- Style requirements

# Control Structures

- Conditionals and loops
- Blocks
- Brackets vs the `at` method
- String `resize` method
- The `stringstream` class
- The `char` type is a byte not a character
- Style requirements
- Loop invariants
  - not on exams

# Vectors

- Vector class
  - Constructing
  - Methods
- Brackets vs the `at` method
- Sorting vectors
- Stack class
- Use cases
  - Not on exams



# Classes and Functions

- Object vs value
- Data hiding and encapsulation
- Namespaces
- Method declarations
- Documentation
  - doxygen comments
  - comments in code
- Visibility
- Static members
- Files
  - Header
  - Source
- Throwing and catching exceptions
- Parameter passing
  - By value (copy)
  - By reference
  - By const reference
- Lambda expressions
- Style requirements

# Iterators

- Notion of iterator
- Dereferencing
- Iterator arithmetic
- Container methods
  - Observers
  - Modifiers
  - Iterators
  - Others
- Default constructor
- Booleans and flags
- Changing recursive methods to non-recursive
  - Not on exams

# Library Algorithms

- Inserter iterators
- Predicates
- Naming conventions
- Library algorithm parameters
- Sequential observers
- Sequential modifiers
- Partitioning
- Sorting
- Operations on sorted data
- Heap operations
- Extremes
- Lexicographic ordering
- Enumerations
- Equality operator
- Typedefs
- Pairs and tuples
- Stable sorting and partitioning
- UML diagrams
  - Not on exams
- Complexity
  - Not on exams

# CS3520 MidTerm Exam Sample Question

The MidTerm Exam will have questions in which you are given part of a class and you must fill in the missing parts. The following is one such question.

Here is the header for a class that computes the fibonacci sequence:

```
#ifndef FIBONACCI_H
#define FIBONACCI_H

#include <vector>

namespace fib {

/**
 * Compute the fibonacci sequence, caching known values.
 * The fibonacci sequence is defined recursively on
 * unsigned integers by the following:
 *
 * <ol>
 * <li>fibonacci<sub>0</sub> = 0
 * <li>fibonacci<sub>1</sub> = 1
 * <li>fibonacci<sub>n</sub> = fibonacci<sub>n-1</sub> + fibonacci<sub>n-2</sub>
 * </ol>
 *
 * @author Ken Baclawski
 */
class Fibonacci {
```

```

public:
    /**
     * Compute the fibonacci sequence.
     * @return the entry at the specified position in the fibonacci sequence.
     */
    int fibonacci(** The position in the fibonacci sequence. */
                 unsigned int position);

private:
    /**
     * The saved values of the fibonacci sequence.
     */
    std::vector<int> saved_;
};

}

#endif

```

Here is the source file for the Fibonacci class but with some missing parts. Show what the missing parts should be.

```

#include "Fibonacci.h"

using namespace std;
using namespace fib;

```

```
int Fibonacci::fibonacci(unsigned int position) {  
  
    // Take care of the base cases  
  
    ... Missing Part 1 ...  
  
    // Make sure that the saved list is big enough  
  
    ... Missing Part 2 ...  
  
    // Look up existing value  
    if (saved_.at(position) > 0) {  
  
        ... Missing Part 3 ...  
  
    }  
  
    // Compute the new case and save it.  
  
    int result = fibonacci(position - 1) + fibonacci(position - 2);  
  
    ... Missing Part 4 ...  
}
```

# Next Friday Class

- Finish Threads
- Brief Introduction to Graphics