



**PRIMEDIC™**  
Saves Life. Everywhere.



Multithreading done right?

# Overview

- Threads
- Shared Variables
- Thread local data
- Condition Variables
- Tasks
- Memory Model



# Threads

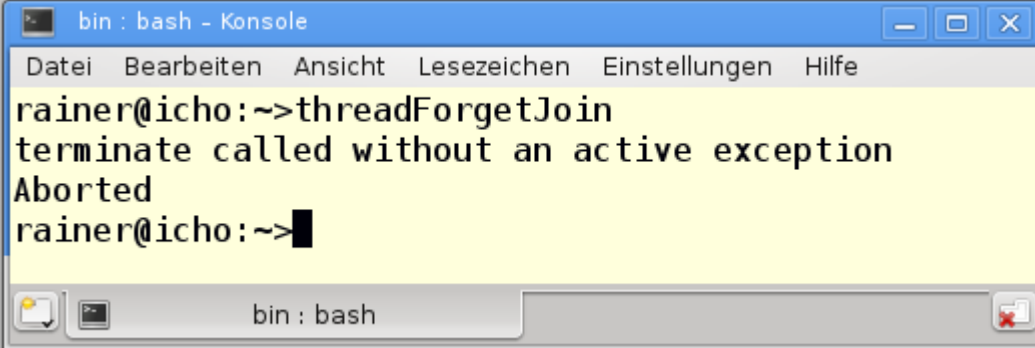


- Needs a work package and run immediately
- The creator has to care of this child by
  - waiting for his child
  - detach from his child
- Accepts the data by copy or by reference

# Problems?

```
#include <iostream>
#include <thread>

int main(){
    thread t( []{cout << this_thread::get_id();} );
}
```

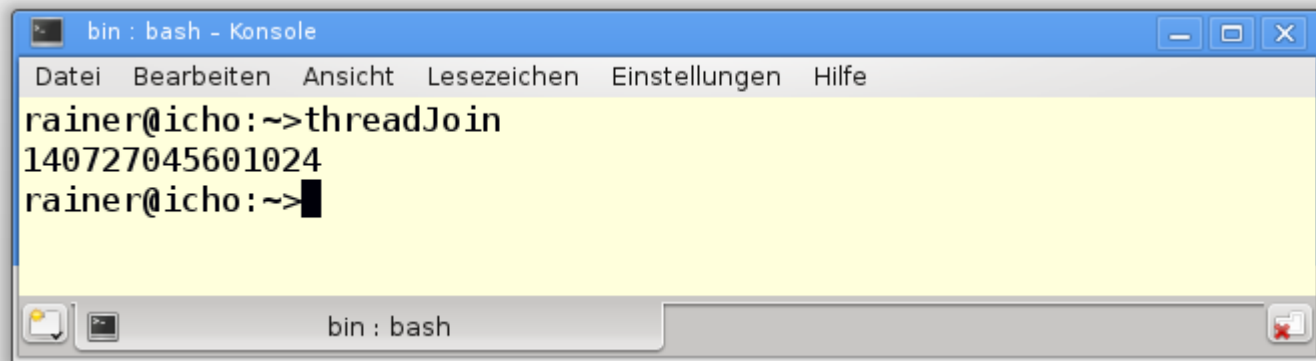
A terminal window titled "bin : bash - Konsole" with a menu bar containing "Datei", "Bearbeiten", "Ansicht", "Lesezeichen", "Einstellungen", and "Hilfe". The terminal content shows the command "threadForgetJoin" being executed, followed by the error message "terminate called without an active exception" and "Aborted". The prompt "rainer@icho:~>" is visible at the end of the line.

```
bin : bash - Konsole
Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe
rainer@icho:~>threadForgetJoin
terminate called without an active exception
Aborted
rainer@icho:~>█
```

# Solution: join or detach

```
#include <iostream>
#include <thread>

int main(){
    thread t( []{cout << this_thread::get_id();} );
    t.join();
    // t.detach();
}
```

A screenshot of a terminal window titled "bin : bash - Konsole". The window has a menu bar with "Datei", "Bearbeiten", "Ansicht", "Lesezeichen", "Einstellungen", and "Hilfe". The terminal content shows the command "threadJoin" being executed, which outputs the thread ID "140727045601024". The prompt "rainer@icho:~>" is visible before and after the command. The terminal window has standard window controls (minimize, maximize, close) in the top right corner and a taskbar at the bottom with a "bin : bash" tab and a close button.

# Problems?

```
#include <iostream>
#include <thread>

int main(){
    thread t( []{cout << this_thread::get_id();} );
    thread t2([]{cout << this_thread::get_id();});

    t= std::move(t2);

    t.join();
    t2.join();
}
```

A terminal window titled "bin : bash - Konsole <2>". The window has a menu bar with "File", "Edit", "View", "Bookmarks", "Settings", and "Help". The terminal output shows the command "threadMove" being executed, followed by the error message "terminate called without an active exception" and "Aborted". The prompt "rainer@linux: ~>" is visible at the end of the line.

```
bin : bash - Konsole <2>
File Edit View Bookmarks Settings Help
rainer@linux: ~>threadMove
terminate called without an active exception
Aborted
rainer@linux: ~>
```

# Solution: join or detach

```
#include <iostream>
#include <thread>

int main(){
    thread t( []{cout << this_thread::get_id();} );
    thread t2([]{std::cout << this_thread::get_id();});

    t.join();
    t= std::move(t2);
    t.join();

    cout << "t2.joinable(): " << t2.joinable();
}
```

A terminal window titled "bin : bash - Konsole <2>". The window contains the following text:

```
File Edit View Bookmarks Settings Help
rainer@linux:~>threadMove
139829891147520
139829882754816

t2.joinable(): false
rainer@linux:~>█
```

The terminal window has a blue title bar and a grey border. The text is displayed on a yellow background. The prompt "rainer@linux:~>" is followed by the command "threadMove". The output shows two thread IDs: "139829891147520" and "139829882754816". Below that, the output "t2.joinable(): false" is shown. The prompt "rainer@linux:~>" is followed by a cursor "█".

# Solution: A. Williams "C++ Concurrency in Action"

```
class scoped_thread{
    std::thread t;
public:
    explicit scoped_thread(std::thread t_):t(std::move(t_)){
        if ( !t.joinable() ) throw std::logic_error("No thread");
    }
    ~scoped_thread(){
        t.join();
    }
    scoped_thread(scoped_thread&)= delete;
    scoped_thread& operator=(scoped_thread const &)= delete;
};

int main(){
    scoped_thread t(std::thread(
        []{std::cout << std::this_thread::get_id();}
    ));
};
```





# Problems?

```
struct Sleeper{
    Sleeper(int& i_):i{i_}{};
    void operator() (int k){
        for (unsigned int j= 0; j <= 5; ++j){
            this_thread::sleep_for(chrono::milliseconds(100));
            i += k;
        }
    }
private:
    int& i;
};

int main(){
    int valSleeper= 1000;
    cout << "valSleeper = " << valSleeper << endl;
    thread t(Sleeper(valSleeper),5);
    t.detach();
    cout << "valSleeper = " << valSleeper << endl;
}
```



# Effect and Solution

```
bin : bash - Konsole
Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe
rainer@icho:~>variableOutOfScope

valSleeper = 1000
valSleeper = 1000

rainer@icho:~>
```



```
thread t(Sleeper(valSleeper),5);
t.join(); // instead of t.detach()
```

```
bin : bash - Konsole
Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe
rainer@icho:~>variableNotOutOfScope

valSleeper = 1000
valSleeper = 1030

rainer@icho:~>
```

# Shared Variables



<http://www.robinage.com/>

- Has to be protected from shared access by
  - Atomic variables
  - Mutexe

# Problems?

```
struct Worker{
    Worker(string n):name(n){};
    void operator() (){
        for (int i= 1; i <= 3; ++i){
            this_thread::sleep_for(chrono::milliseconds(200));
            cout << name << ": " << "Work " << i << " done !!!" << endl;
        }
    }
private:
    string name;
};

...
cout << "Boss: Let's start working.\n\n";
thread herb= thread(Worker("Herb"));
thread andrei= thread(Worker("  Andrei"));
thread scott= thread(Worker("    Scott"));
thread bjarne= thread(Worker("      Bjarne"));
thread andrew= thread(Worker("        Andrew"));
thread david= thread(Worker("          David"));
    // join all Worker
cout << "\n" << "Boss: Let's go home." << endl;
```



# Effect

```
bin : bash - Konsole
Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe
rainer@icho:~>bossAndWorker

Boss: Let's start working.

      Bjarne: Work 1  Andrei done !!!!: Work          David1: Work
Herb: Work 1 done !!!
      Andrew: Work 1 done !!!
1 done !!!
done !!!
      Scott: Work 1 done !!!
      Andrei: Work 2 done !!!
Herb: Work 2 done !!!
      Bjarne: Work 2 done !!!
      Andrew  Scott: Work 2 done !!!
: Work 2 done !!!
      David: Work 2 done !!!
      Andrei: Work 3 done !!!
      Andrew: Work 3 done !!!
      Bjarne: Work 3 done !!!
      Scott: Work 3 done !!!
      David: Work 3 done !!!
Herb: Work 3 done !!!

Boss: Let's go home.

rainer@icho:~>
```



# Solution: Mutex

```
mutex coutMutex;

struct Worker{
    Worker(string n):name(n) {};
    void operator() (){
        for (int i= 1; i <= 3; ++i){
            sleep_for(milliseconds(200));
            coutMutex.lock();
            cout << name << ": " << "Work " <<
            i << " done !!!" << endl;
            coutMutex.unlock();
        }
    }
private:
    string name;
};
```



**std::cout is thread-safe.**

```
bin : bash - Konsole
Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe
rainer@icho:~>bossWorkerMutex

Boss: Let's start working.

Herb: Work 1 done !!!
  Bjarne: Work 1 done !!!
  Scott: Work 1 done !!!
  Andrei: Work 1 done !!!
  Andrew: Work 1 done !!!
  David: Work 1 done !!!
Herb: Work 2 done !!!
  Scott: Work 2 done !!!
  Bjarne: Work 2 done !!!
  David: Work 2 done !!!
  Andrew: Work 2 done !!!
  Andrei: Work 2 done !!!
  Bjarne: Work 3 done !!!
Herb: Work 3 done !!!
  Scott: Work 3 done !!!
  Andrew: Work 3 done !!!
  David: Work 3 done !!!
  Andrei: Work 3 done !!!

Boss: Let's go home.
rainer@icho:~>
```

# Still a problem?

```
void operator() (){
    for (int i= 1; i <= 3; ++i){
        sleep_for(milliseconds(200));
        coutMutex.lock();
        cout << name << ": " << "Work "
        << i << " done !!!" << endl;
        coutMutex.unlock();
    }
}
```

~~Mutex~~ → Lock



```
void operator() (){
    for (int i= 1; i <= 3; ++i)
        sleep_for(milliseconds(200));
    lock_guard<mutex>coutGuard(coutMutex);
    cout << name << ": " << "Work " << i <<
    " done !!!" << endl;
}
}
```

# Problems? and Effect!

```
struct CriticalData{
    mutex mut;
};

void lock(CriticalData& a, CriticalData& b){
    lock_guard<mutex>guard1(a.mut);
    cout << "get the first mutex" << endl;
    this_thread::sleep_for(chrono::milliseconds(1));
    lock_guard<mutex>guard2(b.mut);
    cout << "get the second mutex" << endl;
    // do something with a and b
}

int main(){
    CriticalData c1;
    CriticalData c2;
    thread t1([&]{lock(c1,c2);});
    thread t2([&]{lock(c2,c1);});
    t1.join();
    t2.join();
}
```

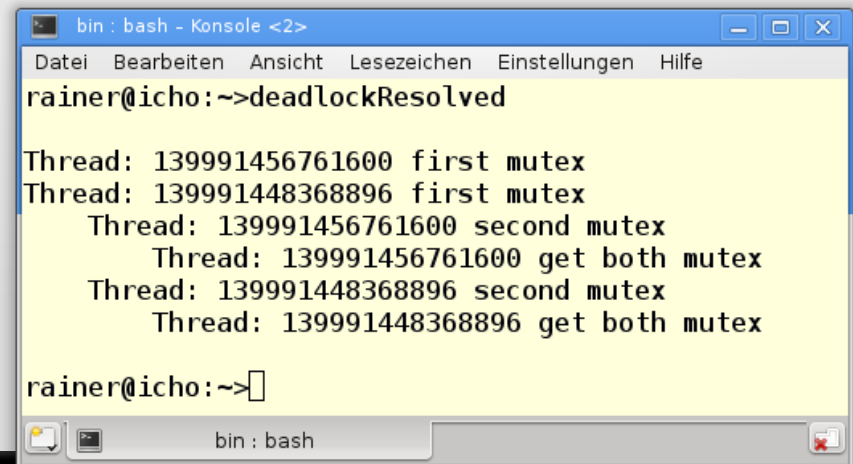
A terminal window titled "bin : bash - Konsole" showing the output of the program. The output is highlighted in yellow and reads: "rainer@icho:~>deadlock", "get the first mutexget the first mutex", and "^C". The prompt "rainer@icho:~>" is followed by a black cursor. The terminal window has a menu bar with "Datei", "Bearbeiten", "Ansicht", "Lesezeichen", "Einstellungen", and "Hilfe". The status bar at the bottom shows "bin : bash" and a red X icon.

```
bin : bash - Konsole
Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe
rainer@icho:~>deadlock
get the first mutexget the first mutex
^C
rainer@icho:~>
```



# Solution: unique\_lock

```
void deadLock(CriticalData& a, CriticalData& b){  
  
    unique_lock<mutex>guard1(a.mut,defer_lock);  
    cout << "Thread: " << get_id() <<" first mutex"<< endl;  
  
    this_thread::sleep_for(chrono::milliseconds(1));  
  
    unique_lock<mutex>guard2(b.mut,defer_lock);  
    cout <<"      Thread: " << get_id() <<" second mutex"<< endl;  
  
    cout <<"          Thread: " << get_id() <<" get both mutex" << endl;  
  
    lock(guard1,guard2);  
  
}
```

A screenshot of a terminal window titled "bin : bash - Konsole <2>". The window shows the output of the program. The first line is "rainer@icho:~>deadlockResolved". This is followed by two lines of output for the first thread: "Thread: 139991456761600 first mutex" and "Thread: 139991456761600 second mutex". Then two lines for the second thread: "Thread: 139991448368896 first mutex" and "Thread: 139991448368896 second mutex". Finally, two lines for both threads: "Thread: 139991456761600 get both mutex" and "Thread: 139991448368896 get both mutex". The prompt "rainer@icho:~>" is visible at the bottom of the terminal.

```
bin : bash - Konsole <2>  
Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe  
rainer@icho:~>deadlockResolved  
  
Thread: 139991456761600 first mutex  
Thread: 139991448368896 first mutex  
      Thread: 139991456761600 second mutex  
      Thread: 139991456761600 get both mutex  
      Thread: 139991448368896 second mutex  
      Thread: 139991448368896 get both mutex  
  
rainer@icho:~>
```

# Solution, but ...

```
mutex myMutex;

class MySingleton{
public:
    static MySingleton& getInstance(){
        lock_guard<mutex> myLock(myMutex);
        if( !instance ) instance= new MySingleton();
        return *instance;
    }
private:
    MySingleton();
    ~MySingleton();
    MySingleton(const MySingleton&)= delete;
    MySingleton& operator=(const MySingleton&)= delete;
    static MySingleton* instance;
};
MySingleton::MySingleton()= default;
MySingleton::~~MySingleton()= default;
MySingleton* MySingleton::instance= nullptr;
...
MySingleton::getInstance();
```



# Problems? Double-Checked Locking Pattern

```
class MySingleton{
    static mutex myMutex;
public:
    static MySingleton& getInstance(){
        if ( !instance ){
            lock_guard<mutex> myLock(myMutex);
            if( !instance ) instance= new MySingleton();
            return *instance;
        }
    }
}
```



`instance= new MySingleton();` is not atomic

1. Allocate memory for MySingleton
2. Create MySingleton object in the memory
3. Refer instance to MySingleton object

**Possible execution order 1, 3 and 2**

# Solution: call\_once, Meyers Singleton or Atomics

## call\_once and once\_flag

```
class MySingleton{
public:
    static MySingleton& getInstance(){
        call_once(initInstanceFlag,
                  &MySingleton::initSingleton);
        return *instance;
    }
private:
    ...
    static once_flag initInstanceFlag;
    static void initSingleton(){
        instance= new MySingleton;
    }
};
...
once_flag MySingleton::initInstanceFlag;
```

## Meyers Singleton

```
class MySingleton{
public:
    static MySingleton& getInstance(){
        static MySingleton instance;
        return instance;
    }
private:
    ...
```

## Atomic variables

# Thread Local Data



- Are exclusively owned by a thread
- Each thread has his own copy of the data
- Behave like static data
  - will be created at the first usage
  - are bound to the lifetime of their thread

# Problems?

```
mutex coutMutex;
thread_local string s("hello from ");

void addThreadLocal(string const& s2){
    s+=s2;
    lock_guard<mutex> guard(coutMutex);
    cout << s << endl;
    cout << "&s: " << &s << endl;
    cout << endl;
}

int main(){
    thread t1(addThreadLocal,"t1");
    thread t2(addThreadLocal,"t2");
    thread t3(addThreadLocal,"t3");
    thread t4(addThreadLocal,"t4");

    t1.join(), t2.join(), t3.join(),t4.join();
}
```



```
bin : bash - Konsole <2>
Datei Bearbeiten Ansicht Lesezeichen Einstellungen >
rainer@icho:~>threadLocal

hello from t4
&s: 0x7f50d3cf16f8

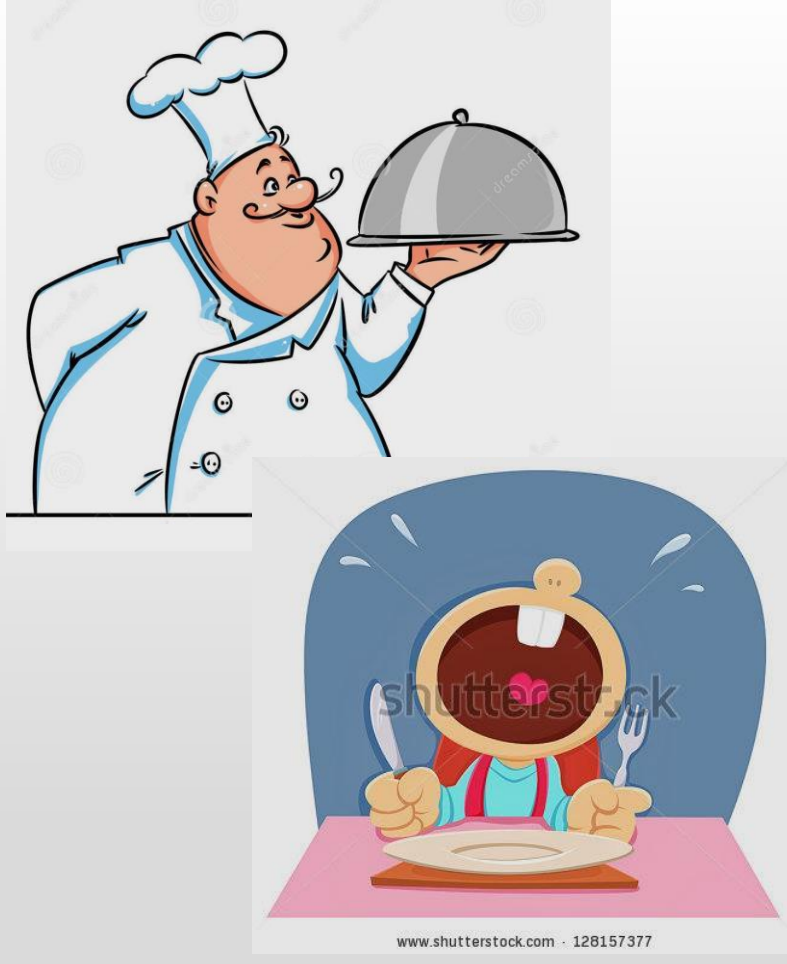
hello from t1
&s: 0x7f50d54f46f8

hello from t3
&s: 0x7f50d44f26f8

hello from t2
&s: 0x7f50d4cf36f8

rainer@icho:~>█
```

# Condition Variables



- Empowers threads to synchronize via notifications
- One threads produces a result, a other thread is still waiting for
- The producer notifies one or more consumers

# Problems?



## Spurious Wakeup

```
mutex mutex_;
condition_variable condVar;

void waitingForWork() {
    unique_lock<mutex> lck(mutex_);
    condVar.wait(lck);
}

void setDataReady() {
    condVar.notify_one();
}

int main() {
    thread t1(waitingForWork);
    thread t2(setDataReady);
    t1.join(),
    t2.join();
}
```

```
bool dataReady= false;

void waitingForWork() {
    unique_lock<mutex> lck(mutex_);
    condVar.wait(lck, []{return dataReady;});
}

void setDataReady() {
    {
        lock_guard<mutex> lck(mutex_);
        dataReady=true;
    }
    condVar.notify_one();
}
```



# Problems?

```
mutex mutex_;
condition_variable condVar;

void waitingForWork() {
    unique_lock<mutex> lck(mutex_);
    condVar.wait(lck, []{return dataReady;});
}

void setDataReady() {
    {
        lock_guard<mutex> lck(mutex_);
        dataReady=true;
    }
    condVar.notify_one();
}

...
thread t1(setDataReady);
// short time delay
thread t2(waitingForWork);

t1.join(), t2.join();
```



## Lost Wakeup

- The `condition_variable` class is a synchronization primitive that can be used to block a thread, or multiple threads **at the same time**, ...

[http://en.cppreference.com/w/cpp/thread/condition\\_variable](http://en.cppreference.com/w/cpp/thread/condition_variable)

# Problem: A lot to do.

```
bin : bash - Konsole <2>
Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe
rainer@icho:~>bossWorker
BOSS: PREPARE YOUR WORK.

    David: Work prepared after 587 milliseconds.
    Bjarne: Work prepared after 804 milliseconds.
    Andrei: Work prepared after 899 milliseconds.
    Scott: Work prepared after 1138 milliseconds.
    Andrew: Work prepared after 1717 milliseconds.
    Herb: Work prepared after 1971 milliseconds.

BOSS: START YOUR WORK.

    Andrew: Work done after 286 milliseconds.
    Scott: Work done after 297 milliseconds.
    Bjarne: Work done after 330 milliseconds.
    Andrei: Work done after 346 milliseconds.
    Herb: Work done after 346 milliseconds.
    David: Work done after 380 milliseconds.

BOSS: GO HOME.

rainer@icho:~>|
```



# Solution: Boss

```
Worker herb("  Herb"), thread herbWork(herb);
...
Worker david("          David"), thread davidWork(david);

cout << "BOSS: PREPARE YOUR WORK.\n " << endl;

preparedCount.store(0);
unique_lock<mutex> preparedUniqueLock( preparedMutex );
worker2BossCondVariable.wait(preparedUniqueLock, []{ return preparedCount == 6; });

startWork= true;  cout << "\nBOSS: START YOUR WORK. \n" << endl;
boss2WorkerCondVariable.notify_all();

doneCount.store(0);
unique_lock<mutex> doneUniqueLock( doneMutex );
worker2BossCondVariable.wait(doneUniqueLock, []{ return doneCount == 6; });

goHome= true;  cout << "\nBOSS: GO HOME. \n" << endl;
boss2WorkerCondVariable.notify_all();

// join all Worker
```



# Solution: Worker

```
struct Worker{
  Worker(string n):name(n){};
  void operator() (){
    int prepareTime= getRandomTime(500,2000);
    this_thread::sleep_for(milliseconds(prepareTime));
    preparedCount++;
    cout << name << ": " << "Work prepared after " << prepareTime << " milliseconds." << endl;
    worker2BossCondVariable.notify_one();
    { // wait for the boss
      unique_lock<mutex> startWorkLock( startWorkMutex );
      boss2WorkerCondVariable.wait( startWorkLock,[] { return startWork;});
    }
    int workTime= getRandomTime(200,400);
    this_tread::sleep_for(milliseconds(workTime));
    doneCount++;
    cout << name << ": " << "Work done after " << workTime << " milliseconds." << endl;
    worker2BossCondVariable.notify_one();
    { // wait for the boss
      unique_lock<mutex> goHomeLock( goHomeMutex );
      boss2WorkerCondVariable.wait( goHomeLock,[] { return goHome;});
    }
  }
  ....
}
```



Use Tasks.



# Tasks



- Are channels between a Sender and a Receiver
- The Producer puts a value in the channel, the Consumer is waiting for
- The Sender is called Promise, the Receiver Future

# Tasks: A little bit of background.

```
int a= 2000;  
int b= 11;
```

- **implicit with `std::async`**

```
future<int> sumFuture= async( [= ] { return a+b; } );  
sumFuture.get(); // 2011
```

- **explicit with `std::future` and `std::promise`**

```
void sum(promise<int>&& intProm, int x, int y) {  
    intProm.set_value(x+y);  
}  
promise<int> sumPromise;  
future<int> futRes= sumPromise.get_future();  
thread sumThread(&sum, move(sumPromise), a, b);  
futRes.get(); // 2011
```



# Problem: A lot to do.

```
bin : bash - Konsole <2>
Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe
rainer@icho:~>bossWorker
BOSS: PREPARE YOUR WORK.

    David: Work prepared after 587 milliseconds.
    Bjarne: Work prepared after 804 milliseconds.
    Andrei: Work prepared after 899 milliseconds.
    Scott: Work prepared after 1138 milliseconds.
    Andrew: Work prepared after 1717 milliseconds.
    Herb: Work prepared after 1971 milliseconds.

BOSS: START YOUR WORK.

    Andrew: Work done after 286 milliseconds.
    Scott: Work done after 297 milliseconds.
    Bjarne: Work done after 330 milliseconds.
    Andrei: Work done after 346 milliseconds.
    Herb: Work done after 346 milliseconds.
    David: Work done after 380 milliseconds.

BOSS: GO HOME.

rainer@icho:~>|
```



# Solution: Boss

```
promise<void> startWorkProm;  
promise<void> goHomeProm;  
  
shared_future<void> startWorkFut= startWorkProm.get_future();  
shared_future<void> goHomeFut= goHomeProm.get_future();  
  
promise<void> herbPrepared;  
future<void> waitForHerbPrepared= herbPrepared.get_future();  
promise<void> herbDone;  
future<void> waitForHerbDone= herbDone.get_future();  
Worker herb("  Herb");  
thread herbWork(herb,move(herbPrepared),startWorkFut,move(herbDone),goHomeFut);  
...  
  
cout << "BOSS: PREPARE YOUR WORK.\n " << endl;  
waitForHerbPrepared.wait(), waitForScottPrepared.wait(), waitFor ...  
cout << "\nBOSS: START YOUR WORK. \n" << endl;  
startWorkProm.set_value();  
waitForHerbDone.wait(), waitForScottDone.wait(), waitFor ...  
cout << "\nBOSS: GO HOME. \n" << endl;  
goHomeProm.set_value();  
  
// join all Worker
```





# Solution: Worker

```
struct Worker{
    Worker(string n):name(n){};
    void operator() (promise<void>&& preparedWork, shared_future<void> boss2WorkerStartWork,
                    promise<void>&& doneWork, shared_future<void>boss2WorkerGoHome ){
        int prepareTime= getRandomTime(500,2000);
        this_thread::sleep_for(milliseconds(prepareTime));
        preparedWork.set_value();
        cout << name << ": " << "Work prepared after " << prepareTime << " milliseconds." << endl;

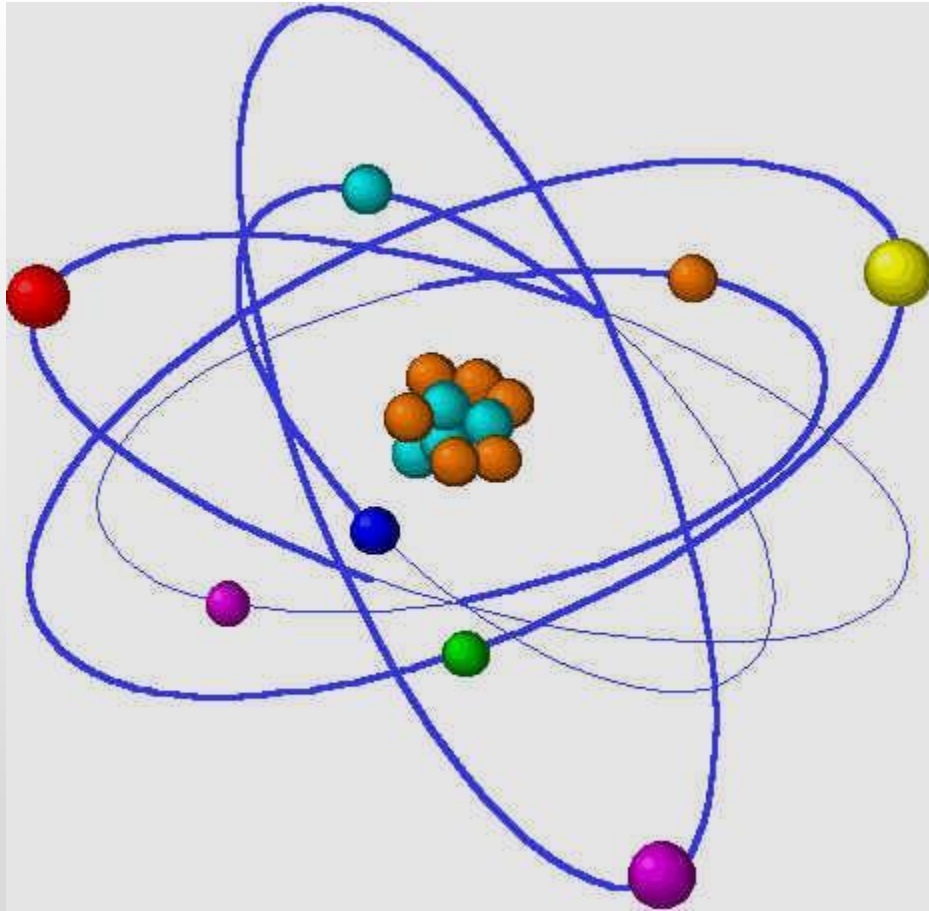
        boss2WorkerStartWork.wait();

        int workTime= getRandomTime(200,400);
        this_thread::sleep_for(milliseconds(workTime));
        doneWork.set_value();
        cout << name << ": " << "Work done after " << workTime << " milliseconds." << endl;

        boss2WorkerGoHome.wait();
    }
private:
    string name;
};
```



# Memory Model



- The Memory Model deals with
  - atomic operations
  - partial ordering of operations
  - visible effect of operations

# Problems?

```
int x= 0;
int y= 0;

void writing() {
    x= 2000;
    y= 11;
}

void reading() {
    cout << "y: " << y << " ";
    cout << "x: " << x << endl;
}

int main() {
    thread thread1(writing);
    thread thread2(reading);
    thread1.join();
    thread2.join();
};
```



y	x	Yes
0	0	<input type="checkbox"/>
11	0	<input type="checkbox"/>
0	2000	<input type="checkbox"/>
11	2000	<input type="checkbox"/>

# Solution: Mutex

```
int x= 0;
int y= 0;
mutex mut;

void writing(){
    lock_guard<mutex> guard(mut);
    x= 2000;
    y= 11;
}

void reading(){
    lock_guard<mutex> guard(mut)
    cout << "y: " << y << " ";
    cout << "x: " << x << endl;
}
...

thread thread1(writing);
thread thread2(reading);
```



y	x	Yes
0	0	<input type="checkbox"/>
11	0	<input type="checkbox"/>
0	2000	<input type="checkbox"/>
11	2000	<input type="checkbox"/>

# Solution: Atomic

```
atomic<int> x= 0;  
atomic<int> y= 0;
```

```
void writing(){  
    x.store(2000);  
    y.store(11);  
}
```

```
void reading(){  
    cout << y.load() << " ";  
    cout << x.load() << endl;  
}
```

...

```
thread thread1(writing);  
thread thread2(reading);
```



y	x	Yes
0	0	<input type="checkbox"/>
11	0	<input type="checkbox"/>
0	2000	<input type="checkbox"/>
11	2000	<input type="checkbox"/>

# Solution: Atomic with acquire release semantic

```
atomic<int> x= 0;  
atomic<int> y= 0;
```

```
void writing() {  
    x.store(2000, memory_order_relaxed);  
    y.store(11, memory_order_release);  
}
```

```
void reading() {  
    cout << y.load(memory_order_acquire) << " ";  
    cout << x.load(memory_order_relaxed) << endl;  
}
```

...

```
thread thread1(writing);  
thread thread2(reading);
```



y	x	Yes
0	0	<input type="checkbox"/>
11	0	<input type="checkbox"/>
0	2000	<input type="checkbox"/>
11	2000	<input type="checkbox"/>

# Solution: Atomic with sychronization of non atomics

```
int x= 0;
atomic<int> y= 0;

void writing(){
    x= 2000;
    y.store(11), memory_order_release);
}

void reading(){
    cout << y.load(memory_order_acquire) << " ";
    cout << x << endl;
}

...

thread thread1(writing);
thread thread2(reading);
```



y	x	Yes
0	0	<input type="checkbox"/>
11	0	<input type="checkbox"/>
0	2000	<input type="checkbox"/>
11	2000	<input type="checkbox"/>

# Solution: Atomic with relaxed semantics

```
atomic<int> x= 0;  
atomic<int> y= 0;
```

```
void writing(){  
    x.store(2000,memory_order_relaxed);  
    y.store(11), memory_order_relaxed);  
}
```

```
void reading(){  
    cout << y.load(memory_order_relaxed) << " ";  
    cout << x.load(memory_order_relaxed) << endl;  
}
```

...

```
thread thread1(writing);  
thread thread2(reading);
```





<b>y</b>	<b>x</b>	<b>Yes</b>
0	0	<input type="checkbox"/>
11	0	<input type="checkbox"/>
0	2000	<input type="checkbox"/>
11	2000	<input type="checkbox"/>



# Solution: Singleton with Atomics

## Sequentially Consistent

## Acquire-Release

```
class MySingleton{
    static mutex myMutex;
    static atomic<MySingleton*> instance;
public:
    static MySingleton* getInstance(){
        MySingleton* sin= instance.load();  instance.load(memory_order_acquire);
        if ( !sin ){
            lock_guard<mutex> myLock(myMutex);
            if( !sin ){
                sin= new MySingleton();
                instance.store(sin);  instance.store(sin,memory_order_release);
            }
        }
        return sin;
    }
    ...
}
```

# THE SOLUTION : CppMem

## CppMem: Interactive C/C++ memory model

Model  
 standard  preferred  release\_acquire  tot  relaxed\_only

Program  
examples/MP\_message\_passing | MP+na\_rel+acq\_na.c

C  Execution

```
MP+na_rel+acq_na
// Message Passing, of data held in non-atomic x,
// with release/acquire synchronisation on y.
// Question: is the read of x required to see the new data value 1
// rather than the initial state value 0?
int main() {
  int x=0; atomic_int y=0;
  {{{ { x=2000;
        y.store(11,memory_order_release); }
      ||| { r1=y.load(memory_order_acquire);
            r2=x; } }}}
  return 0;
}
```

run reset help 8 executions; 2 consistent, only 1 race free

## Execution candidate no. 1 of 8

previous consistent previous candidate next candidate next consistent 1 goto

### Model Predicates

- consistent\_race\_free\_execution = false
- consistent\_execution = true
  - assumptions = true
  - well\_formed\_threads = true
  - well\_formed\_rf = true
  - locks\_only\_consistent\_locks = true
  - locks\_only\_consistent\_lo = true
  - consistent\_mo = true
  - sc\_accesses\_consistent\_sc = true
  - sc\_fenced\_sc\_fences\_heeded = true
  - consistent\_hb = true
  - consistent\_rf = true
  - det\_read = true
  - consistent\_non\_atomic\_rf = true
  - consistent\_atomic\_rf = true
  - coherent\_memory\_use = true
  - rmw\_atomicity = true
  - sc\_accesses\_sc\_reads\_restricted = true
- unsequenced\_races are absent  
data\_races are present  
indeterminate\_reads are absent  
locks\_only\_bad\_mutexes are absent



[CppMem](#)

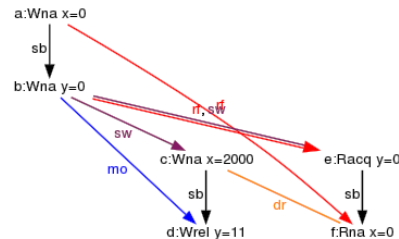
### Computed executions

#### Display Relations

- sb  asw  dd  cd
- rf  mo  sc  lo
- hb  vse  ithb  sw  rs  hrs  dob  cad
- unsequenced\_races  data\_races

#### Display Layout

- dot  neato\_par  neato\_par\_init  neato\_downwards
- tex
- edit display options



Files: out.exc, out.dot, out.dsp, out.tex





**PRIMEDIC™**

Saves Life. Everywhere.

# Thank you for your attention

**Rainer Grimm**  
Metrax GmbH  
[www.primedic.com](http://www.primedic.com)

Phone +49 (0)741 257-0  
Rainer.Grimm@primedic.com