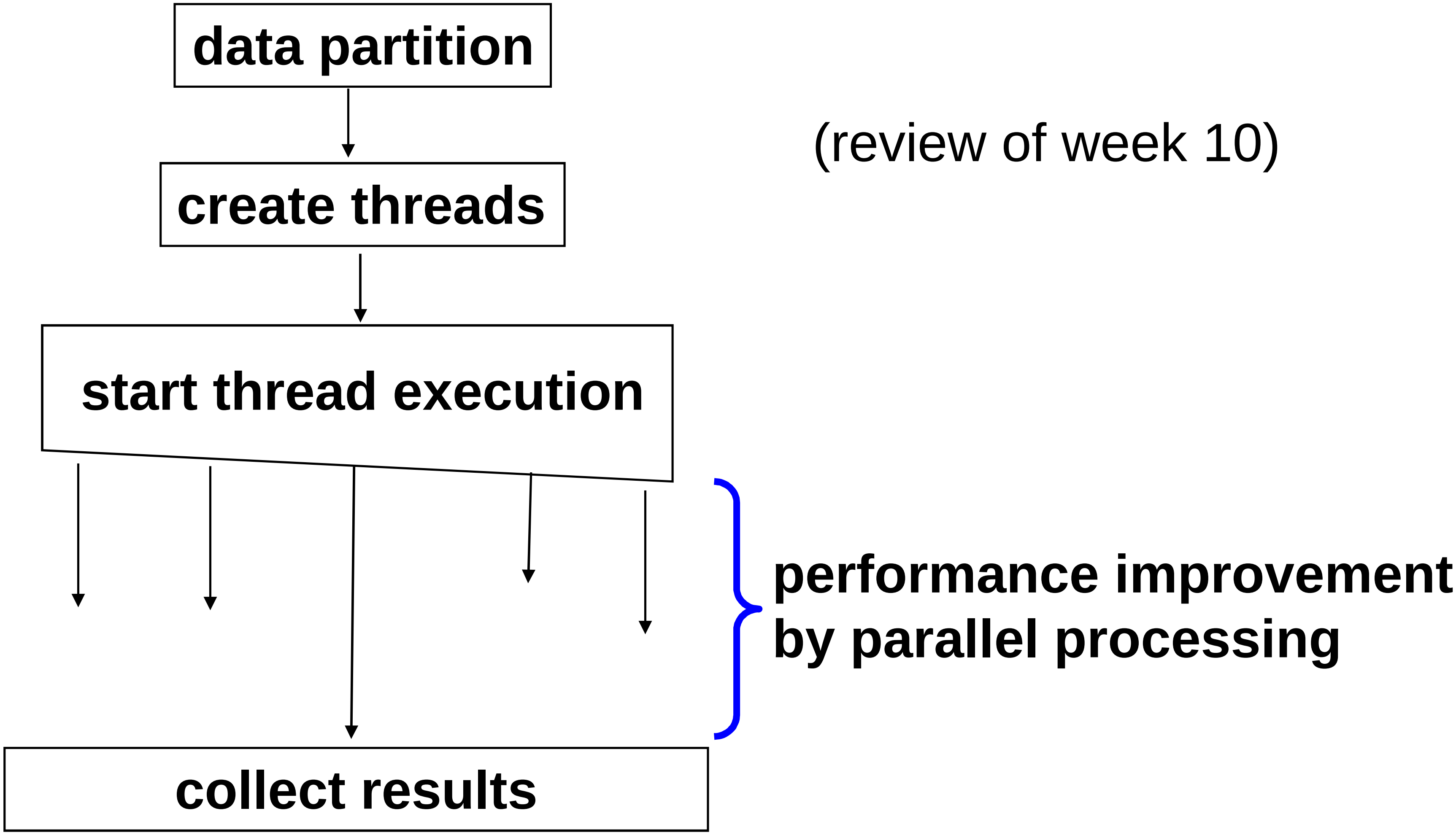
ECE 462
Object-Oriented Programming using C++ and Java

Reuse Threads

Yung-Hsiang Lu
yunglu@purdue.edu

Structure of Parallel Programs



data partition

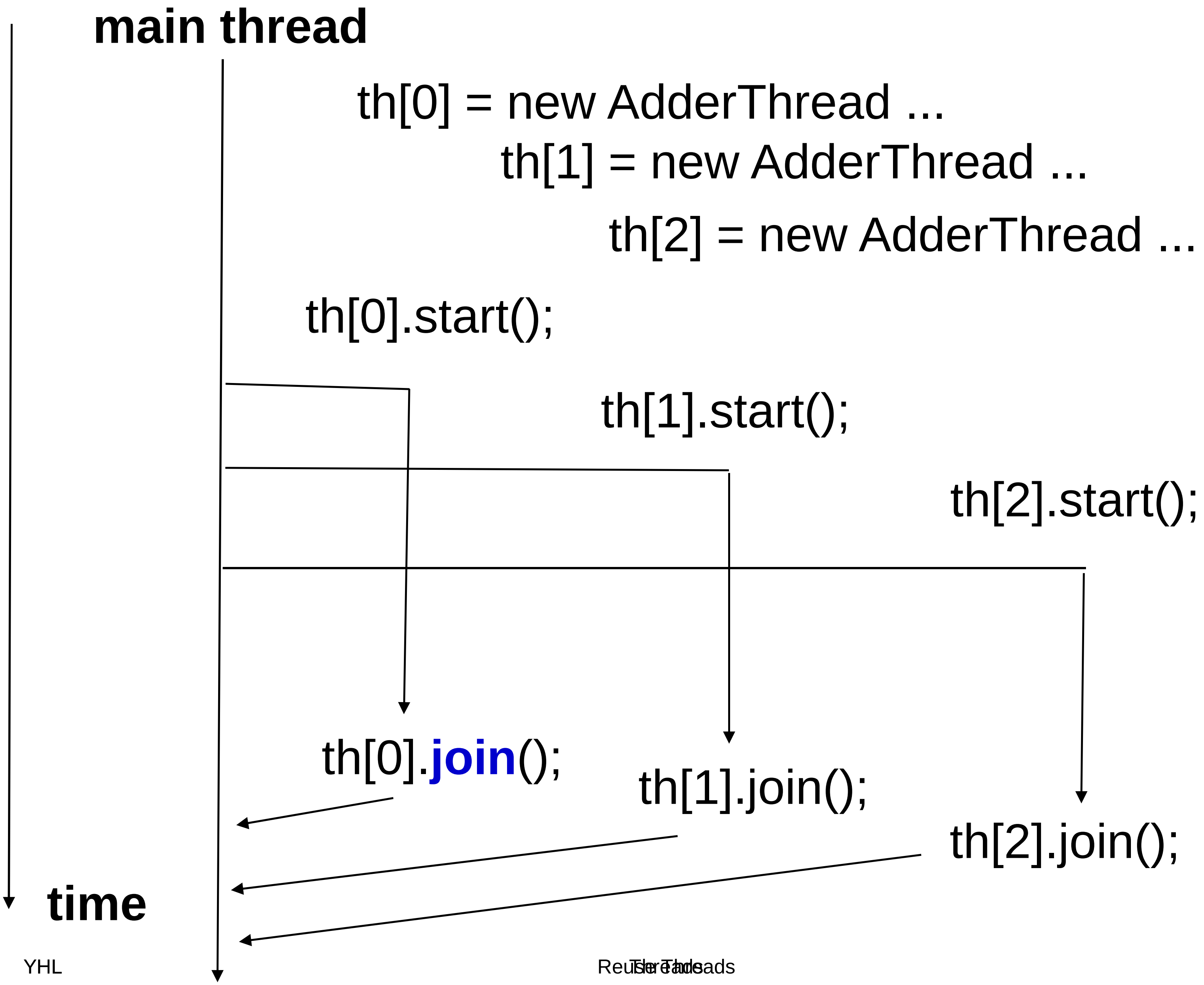
create threads

start thread execution

collect results

(review of week 10)

performance improvement by parallel processing



main thread

th[0] = new AdderThread ...

th[1] = new AdderThread ...

th[2] = new AdderThread ...

th[0].start();

th[1].start();

th[2].start();

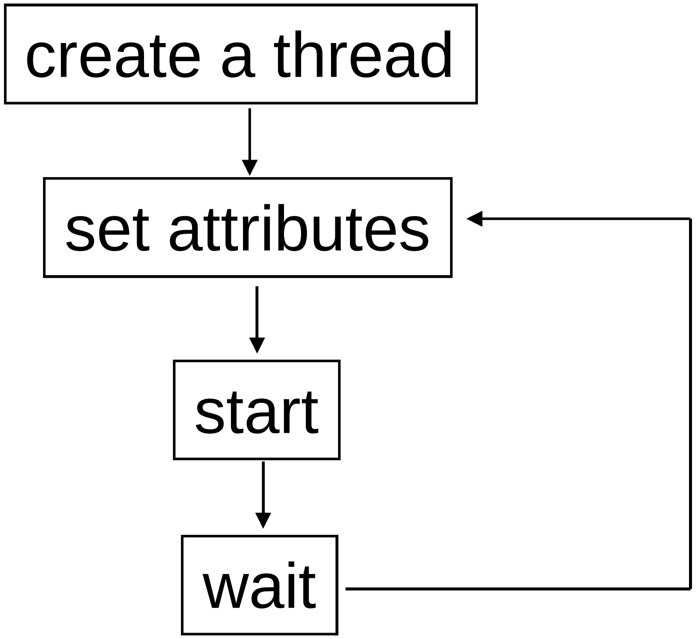
th[0].**join**();

th[1].join();

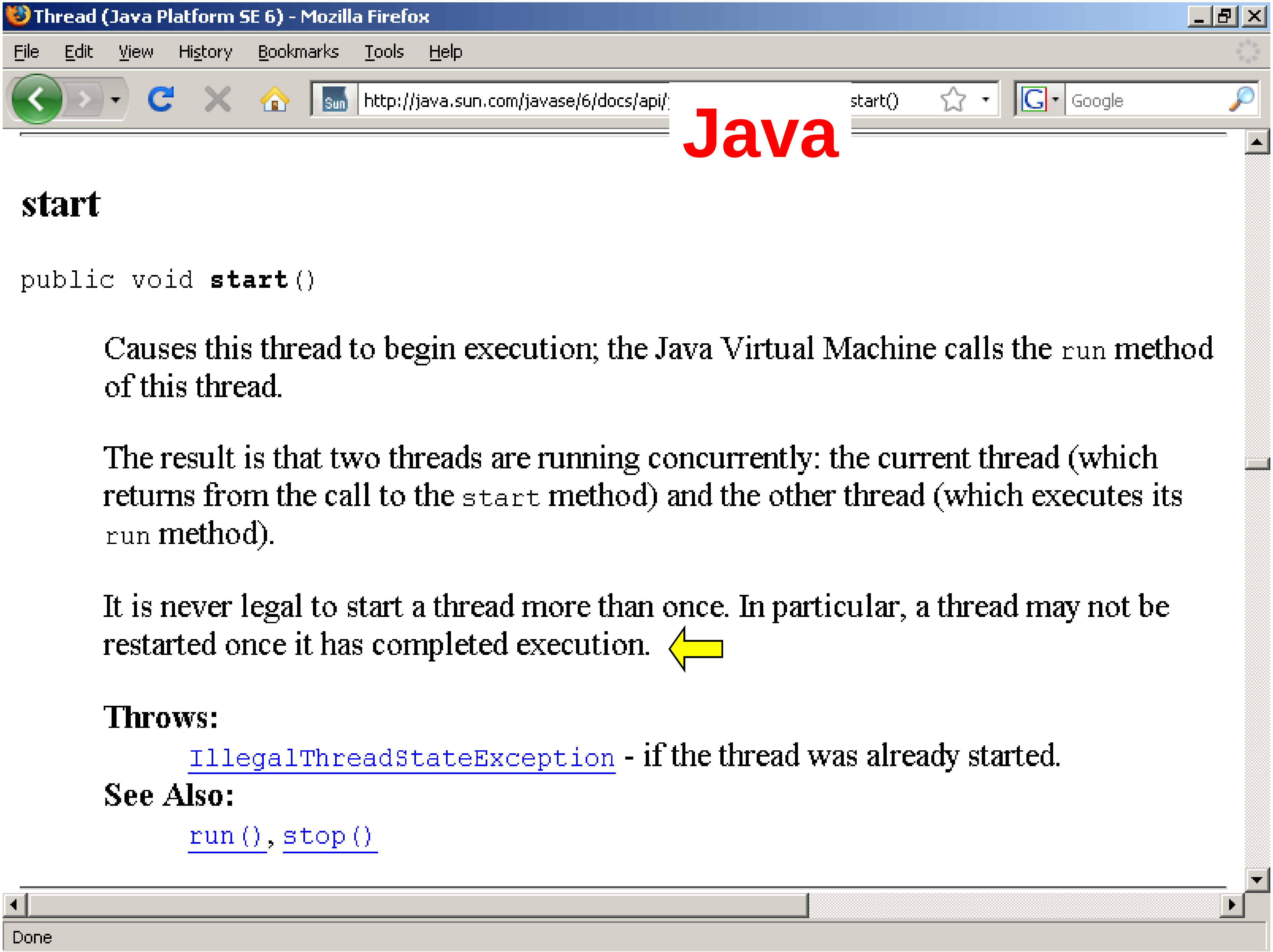
th[2].join();

time

Reuse Threads



Java


Thread (Java Platform SE 6) - Mozilla Firefox
File
Edit
View
History
Bookmarks
Tools
Help
http://java.sun.com/javase/6/docs/api/
start()
Google
Done

start

```
public void start()
```

Causes this thread to begin execution; the Java Virtual Machine calls the `run` method of this thread.

The result is that two threads are running concurrently: the current thread (which returns from the call to the `start` method) and the other thread (which executes its `run` method).

It is never legal to start a thread more than once. In particular, a thread may not be restarted once it has completed execution.

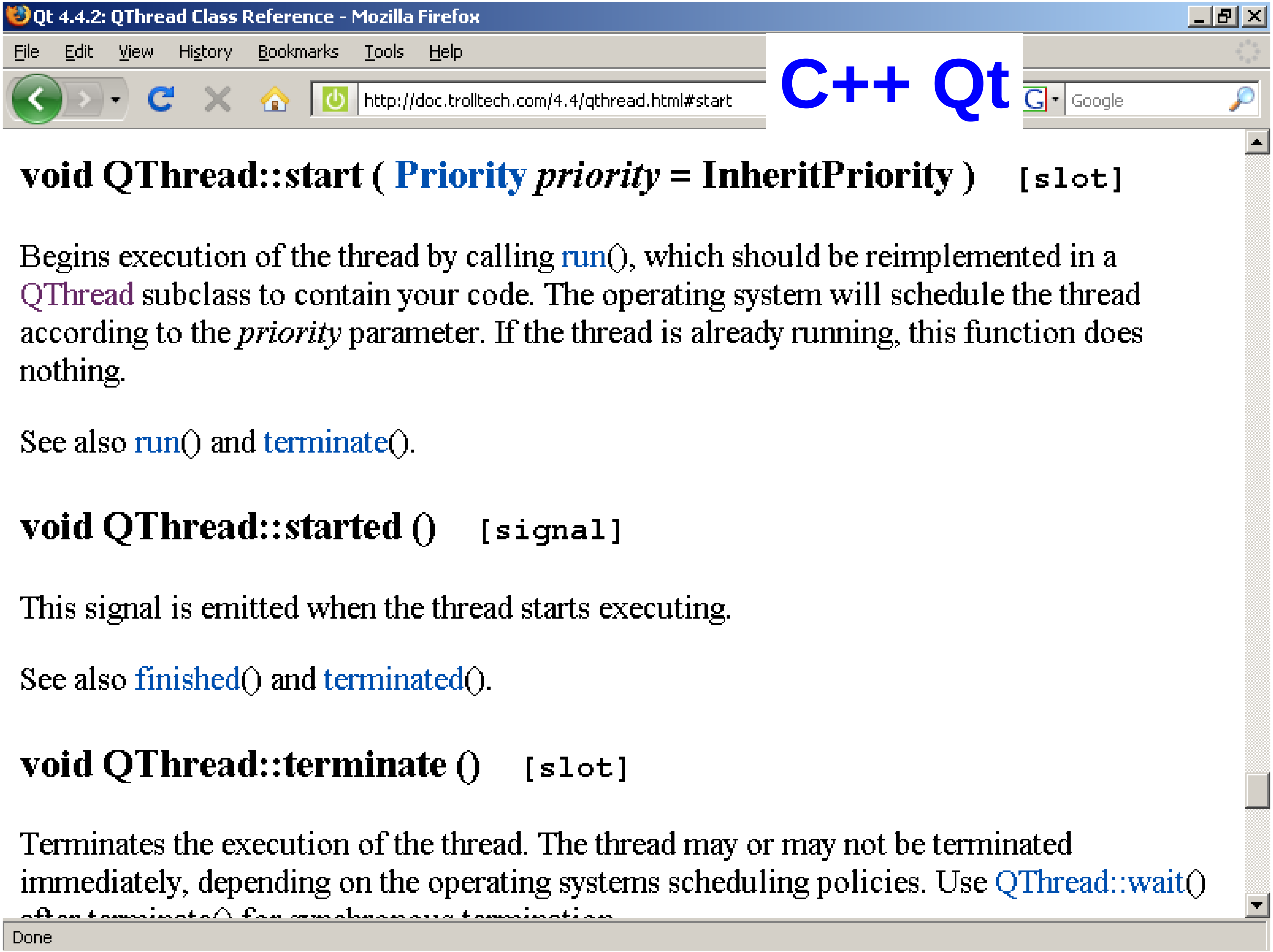
Throws:

`IllegalThreadStateException` - if the thread was already started.

See Also:

`run()`, `stop()`

C++ Qt


Qt 4.4.2: QThread Class Reference - Mozilla Firefox
File
Edit
View
History
Bookmarks
Tools
Help
http://doc.trolltech.com/4.4/qthread.html#start
Google
Done

void QThread::start (Priority *priority* = InheritPriority) `[slot]`

Begins execution of the thread by calling run(), which should be reimplemented in a QThread subclass to contain your code. The operating system will schedule the thread according to the *priority* parameter. If the thread is already running, this function does nothing.

See also run() and terminate().

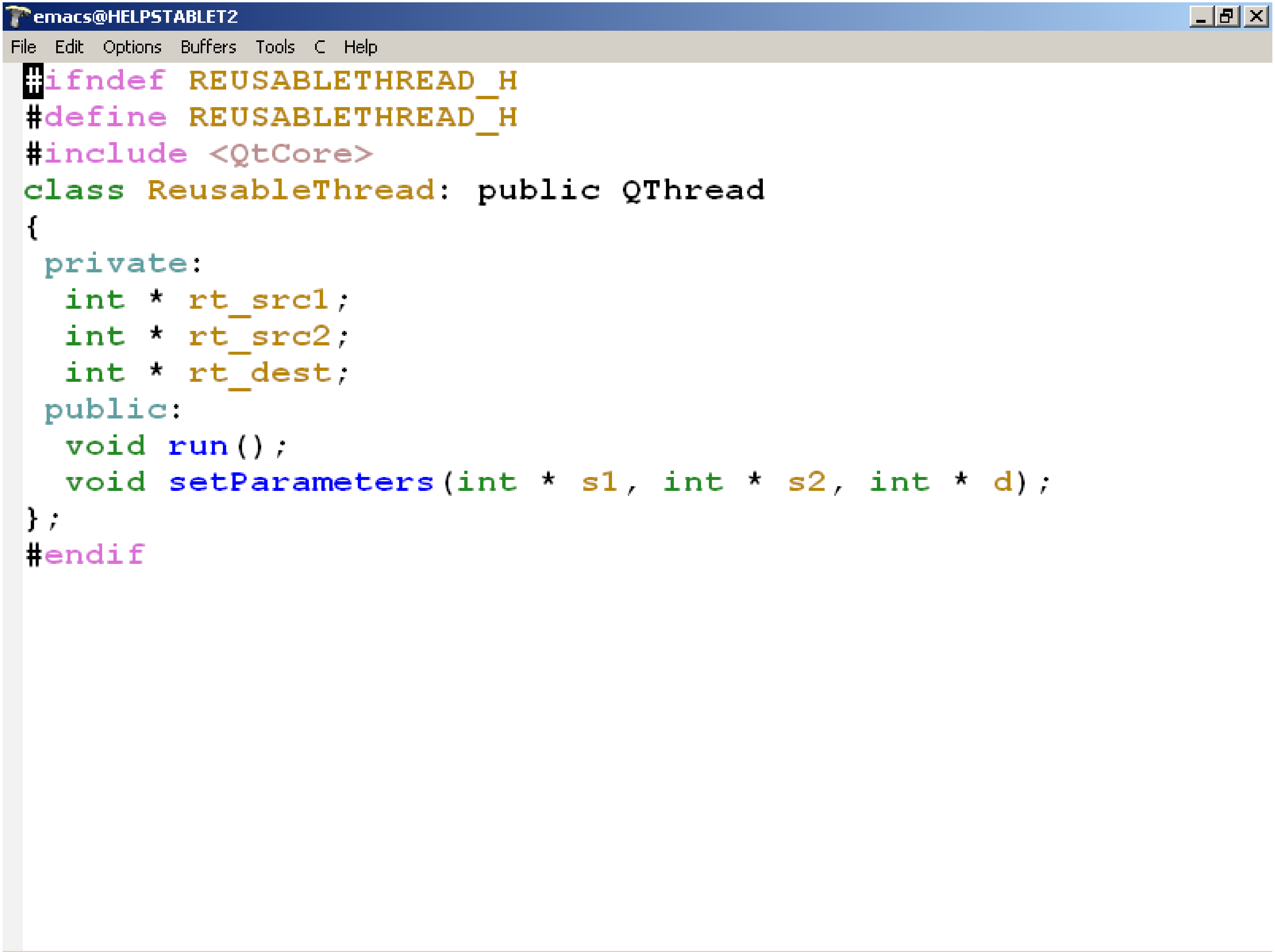
void QThread::started () `[signal]`

This signal is emitted when the thread starts executing.

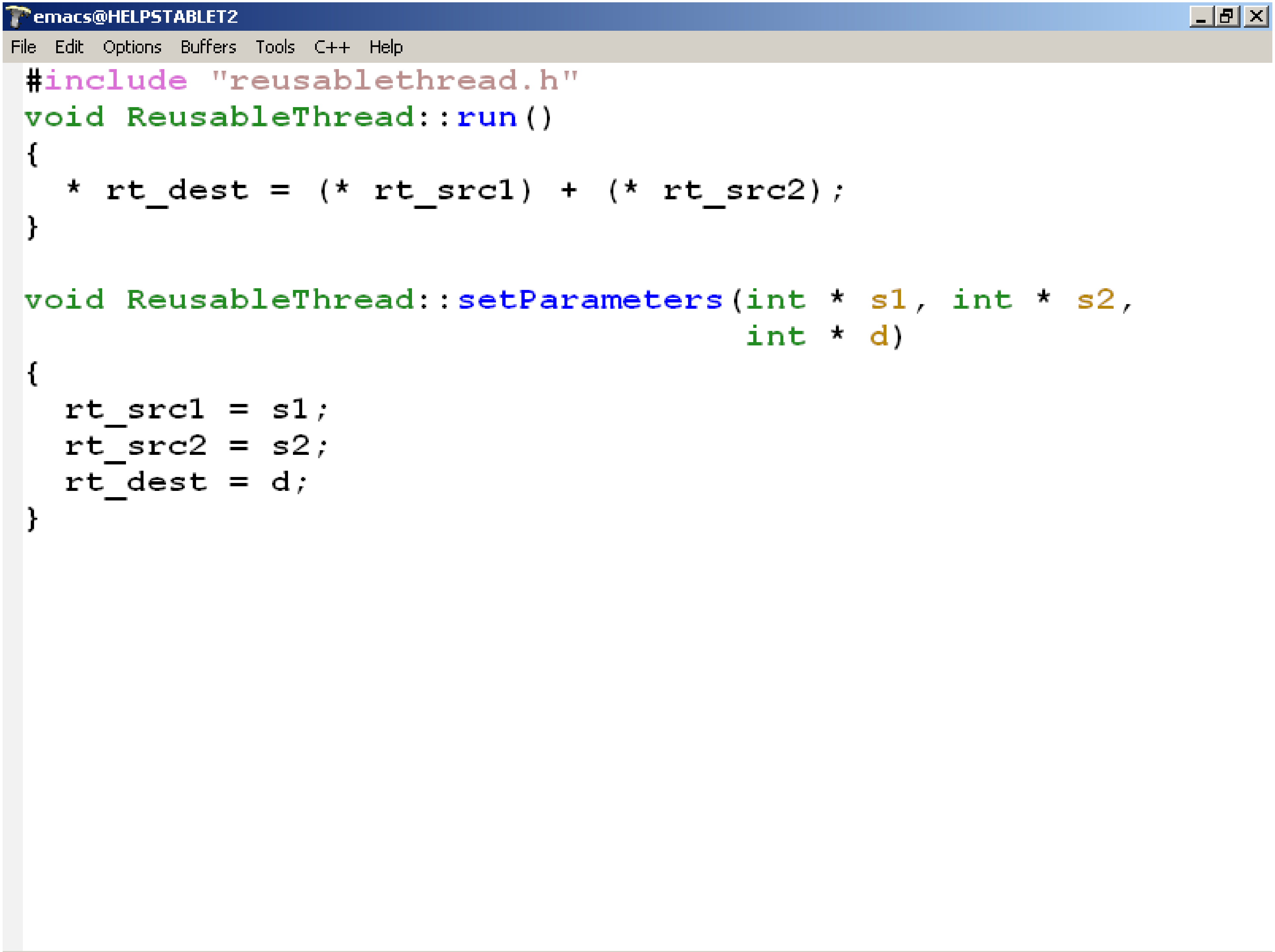
See also finished() and terminated().

void QThread::terminate () `[slot]`

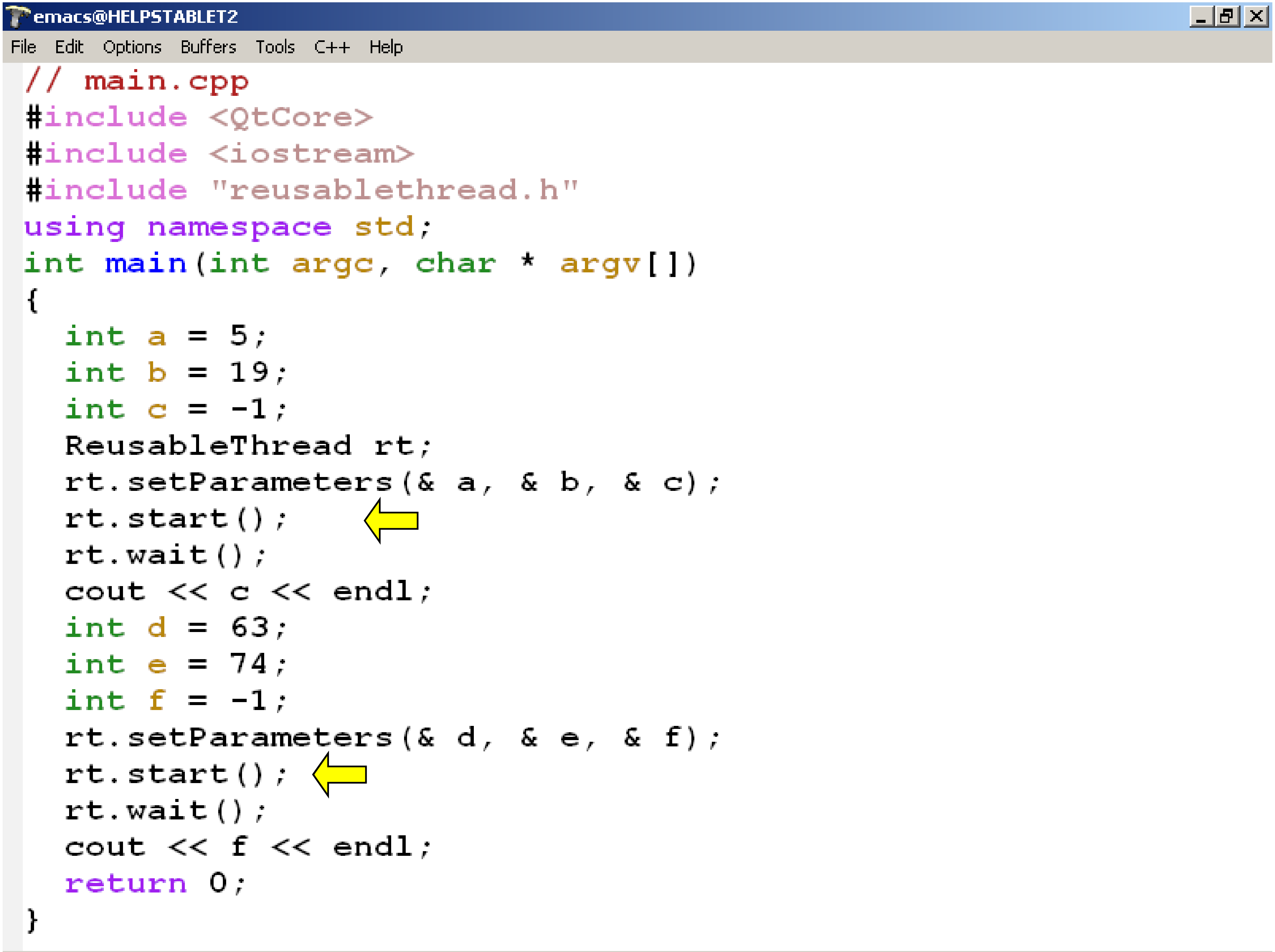
Terminates the execution of the thread. The thread may or may not be terminated immediately, depending on the operating systems scheduling policies. Use QThread::wait() after terminate() for synchronous termination.



```
#ifndef REUSABLETHREAD_H
#define REUSABLETHREAD_H
#include <QtCore>
class ReusableThread: public QThread
{
 private:
  int * rt_src1;
  int * rt_src2;
  int * rt_dest;
 public:
  void run();
  void setParameters(int * s1, int * s2, int * d);
};
#endif
```



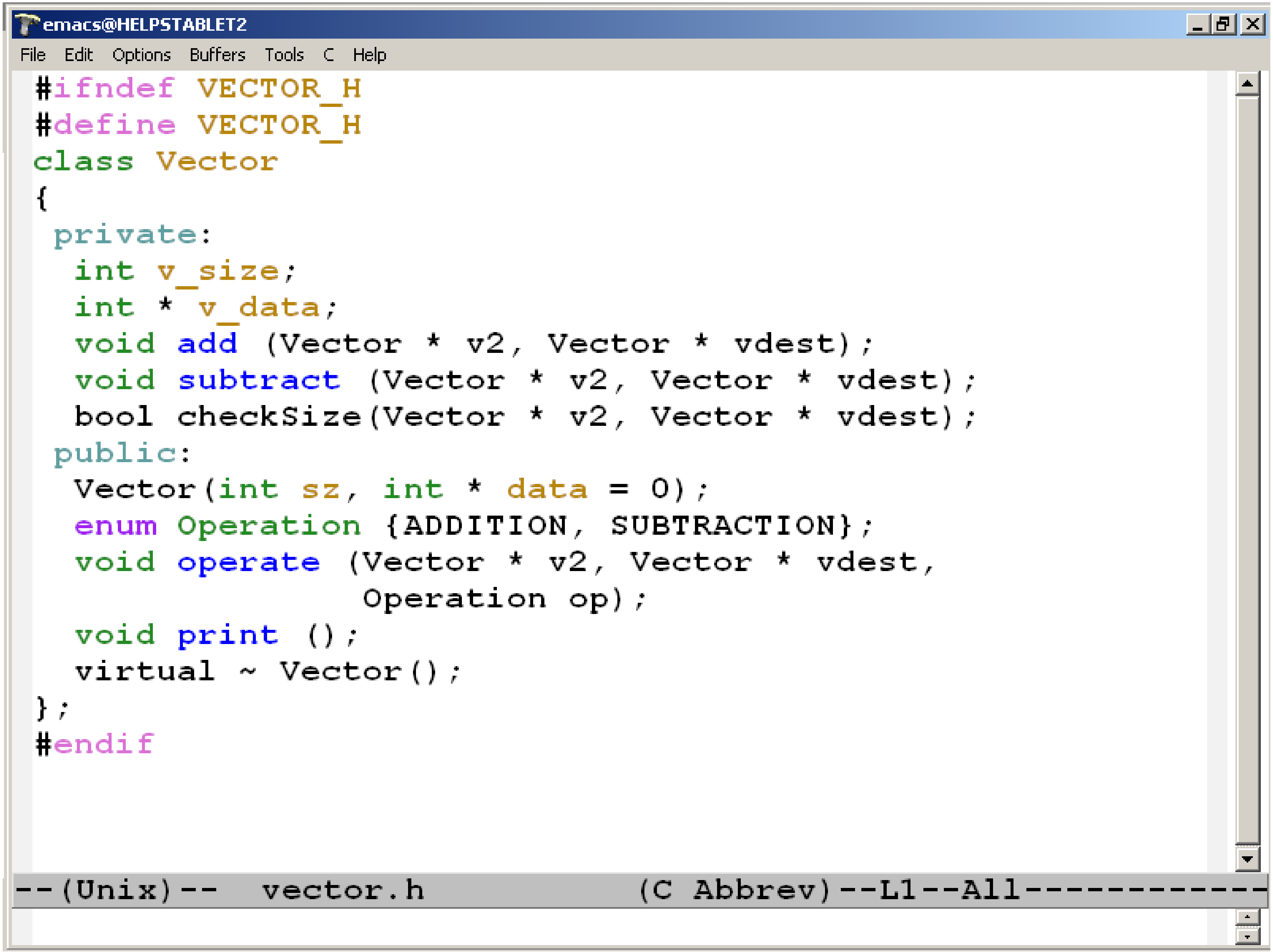
```
#include "reusablethread.h"
void ReusableThread::run()
{
  * rt_dest = (* rt_src1) + (* rt_src2);
}

void ReusableThread::setParameters(int * s1, int * s2,
                                   int * d)
{
  rt_src1 = s1;
  rt_src2 = s2;
  rt_dest = d;
}
```



```cpp
// main.cpp
#include <QtCore>
#include <iostream>
#include "reusablethread.h"
using namespace std;
int main(int argc, char * argv[])
{
  int a = 5;
  int b = 19;
  int c = -1;
  ReusableThread rt;
  rt.setParameters(& a, & b, & c);
  rt.start();
  rt.wait();
  cout << c << endl;
  int d = 63;
  int e = 74;
  int f = -1;
  rt.setParameters(& d, & e, & f);
  rt.start();
  rt.wait();
  cout << f << endl;
  return 0;
}
```

Make (Qt) Threads Reusable

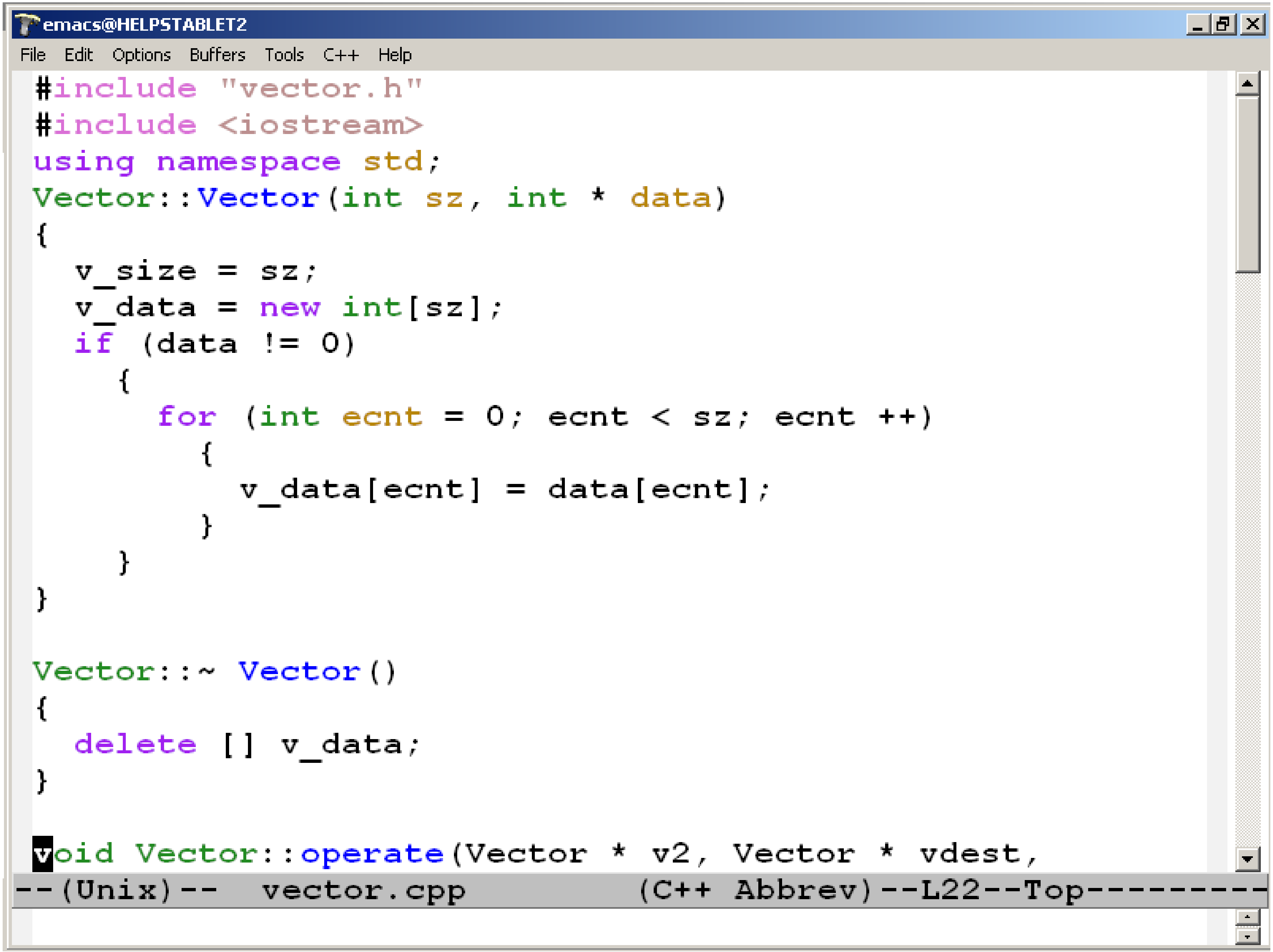
- remove the “intelligence” in threads
  - keep knowledge within the operands
  - implement of the real work inside the objects that need computation
- use threads for computation only. Threads do not need to know what to do. They just supply processor cycles.



emacs@HELPSTABLET2
File Edit Options Buffers Tools C Help

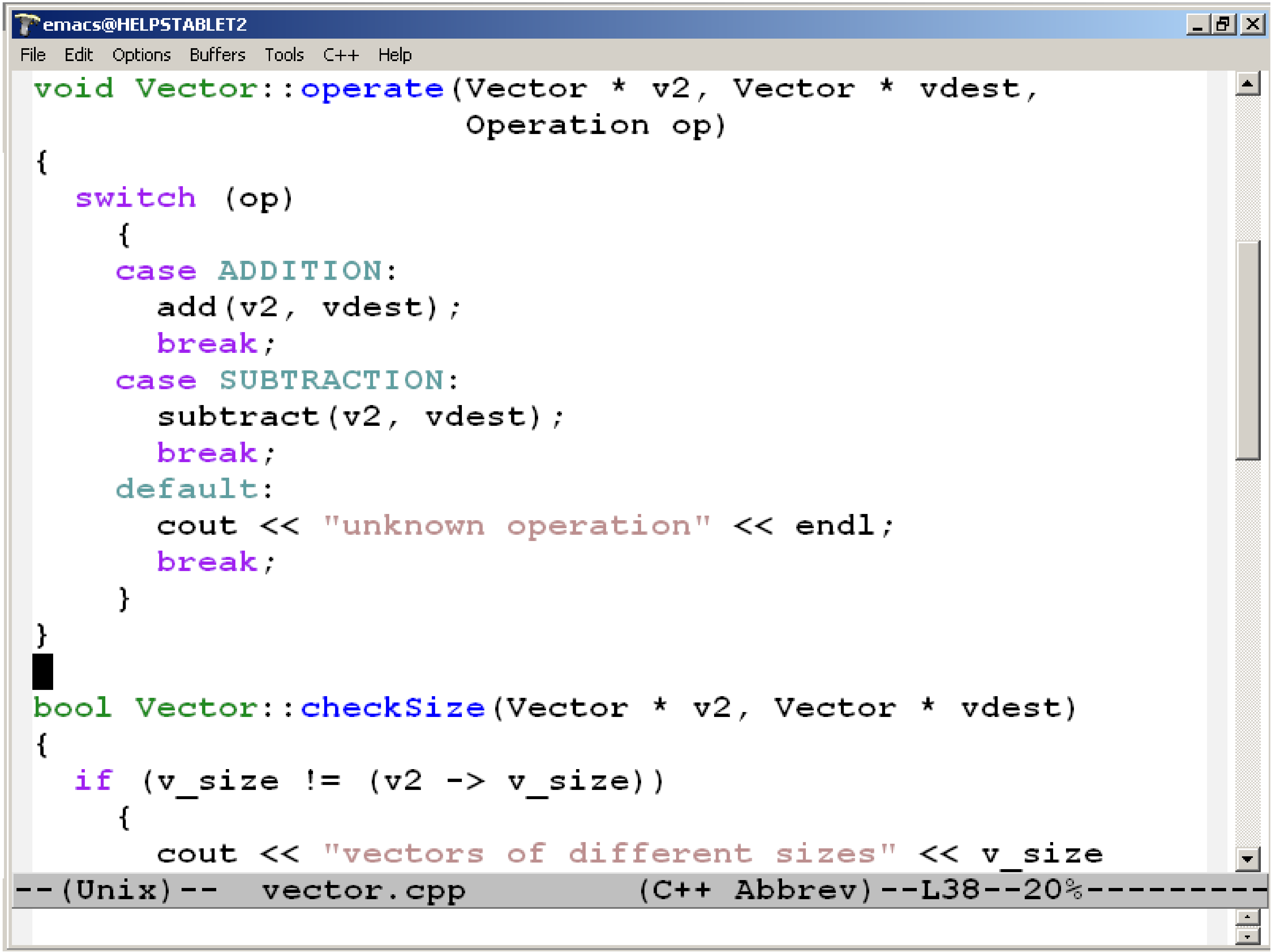
```
#ifndef VECTOR_H
#define VECTOR_H
class Vector
{
 private:
  int v_size;
  int * v_data;
  void add (Vector * v2, Vector * vdest);
  void subtract (Vector * v2, Vector * vdest);
  bool checkSize(Vector * v2, Vector * vdest);
 public:
  Vector(int sz, int * data = 0);
  enum Operation {ADDITION, SUBTRACTION};
  void operate (Vector * v2, Vector * vdest,
                Operation op);
  void print ();
  virtual ~ Vector();
};
#endif
```

--(Unix)--   vector.h   (C Abbrev)--L1--All------------


emacs@HELPSTABLET2
File
Edit
Options
Buffers
Tools
C++
Help

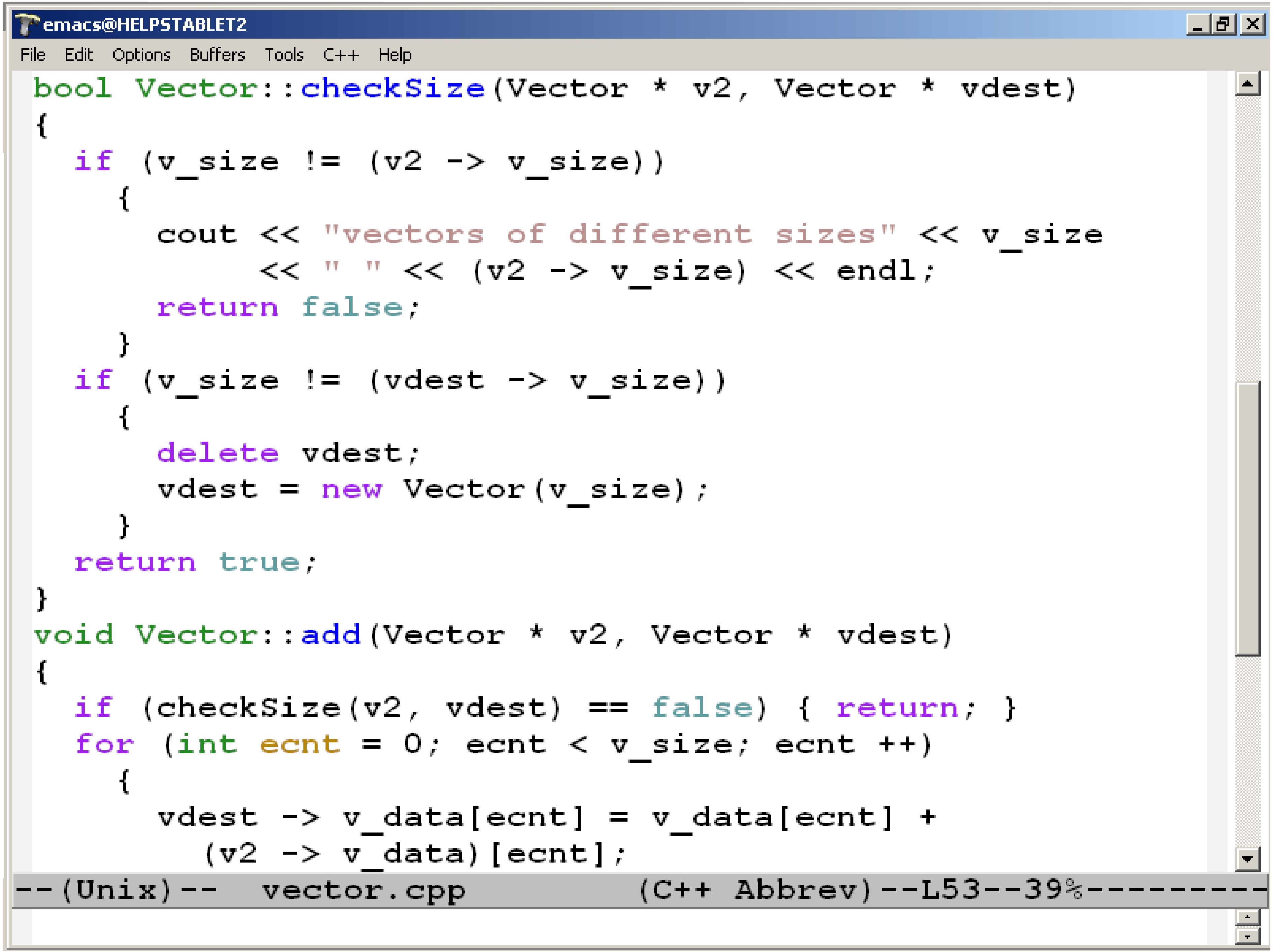
```
#include "vector.h"
#include <iostream>
using namespace std;
Vector::Vector(int sz, int * data)
{
  v_size = sz;
  v_data = new int[sz];
  if (data != 0)
    {
      for (int ecnt = 0; ecnt < sz; ecnt ++)
        {
          v_data[ecnt] = data[ecnt];
        }
    }
}

Vector::~ Vector()
{
  delete [] v_data;
}

void Vector::operate(Vector * v2, Vector * vdest,
```

--(Unix)--   vector.cpp   (C++ Abbrev)--L22--Top---------

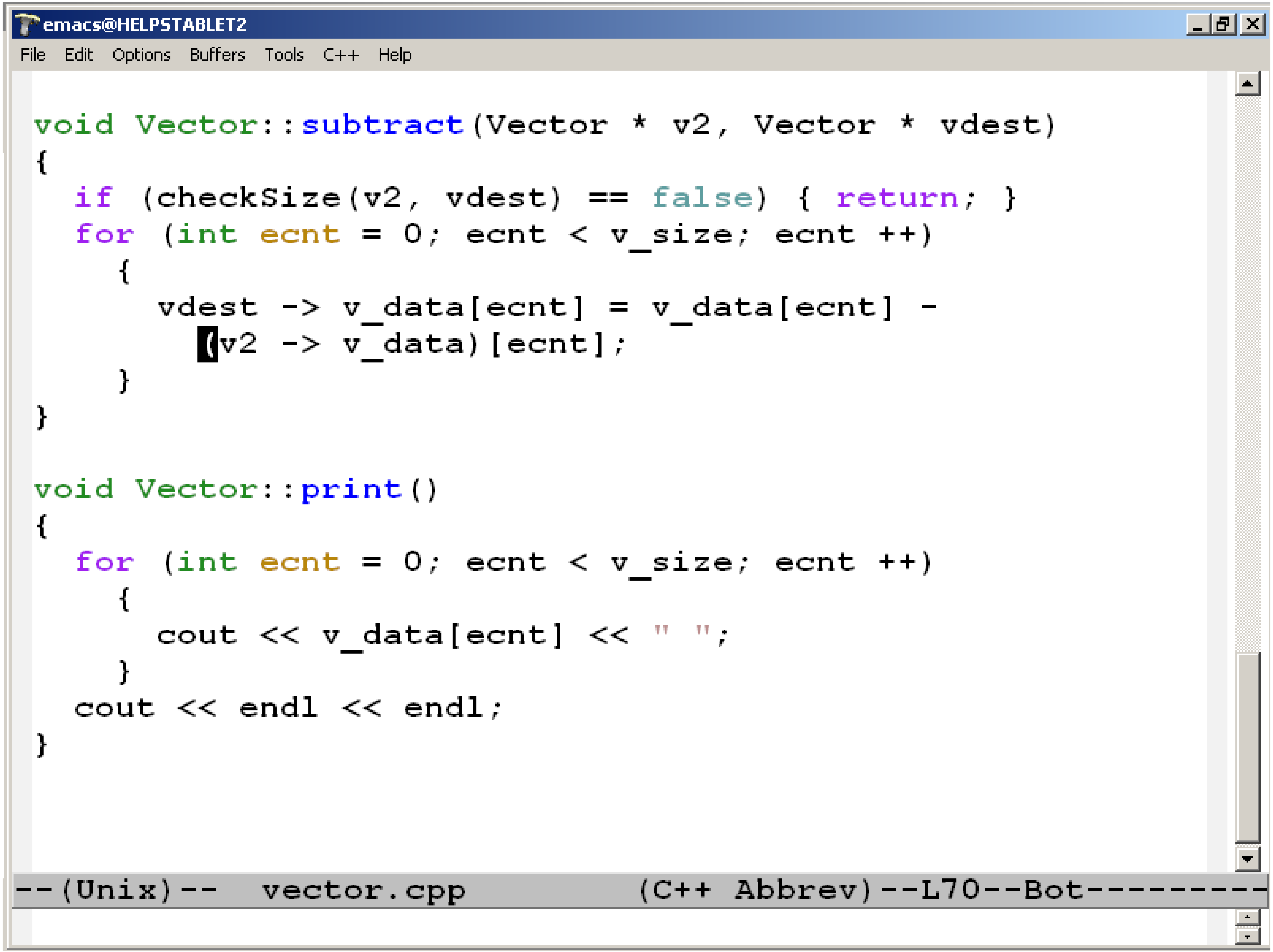


```
void Vector::operate(Vector * v2, Vector * vdest,
                     Operation op)
{
  switch (op)
    {
    case ADDITION:
      add(v2, vdest);
      break;
    case SUBTRACTION:
      subtract(v2, vdest);
      break;
    default:
      cout << "unknown operation" << endl;
      break;
    }
}

bool Vector::checkSize(Vector * v2, Vector * vdest)
{
  if (v_size != (v2 -> v_size))
    {
      cout << "vectors of different sizes" << v_size
```

```
--(Unix)--  vector.cpp        (C++ Abbrev)--L38--20%---------
```

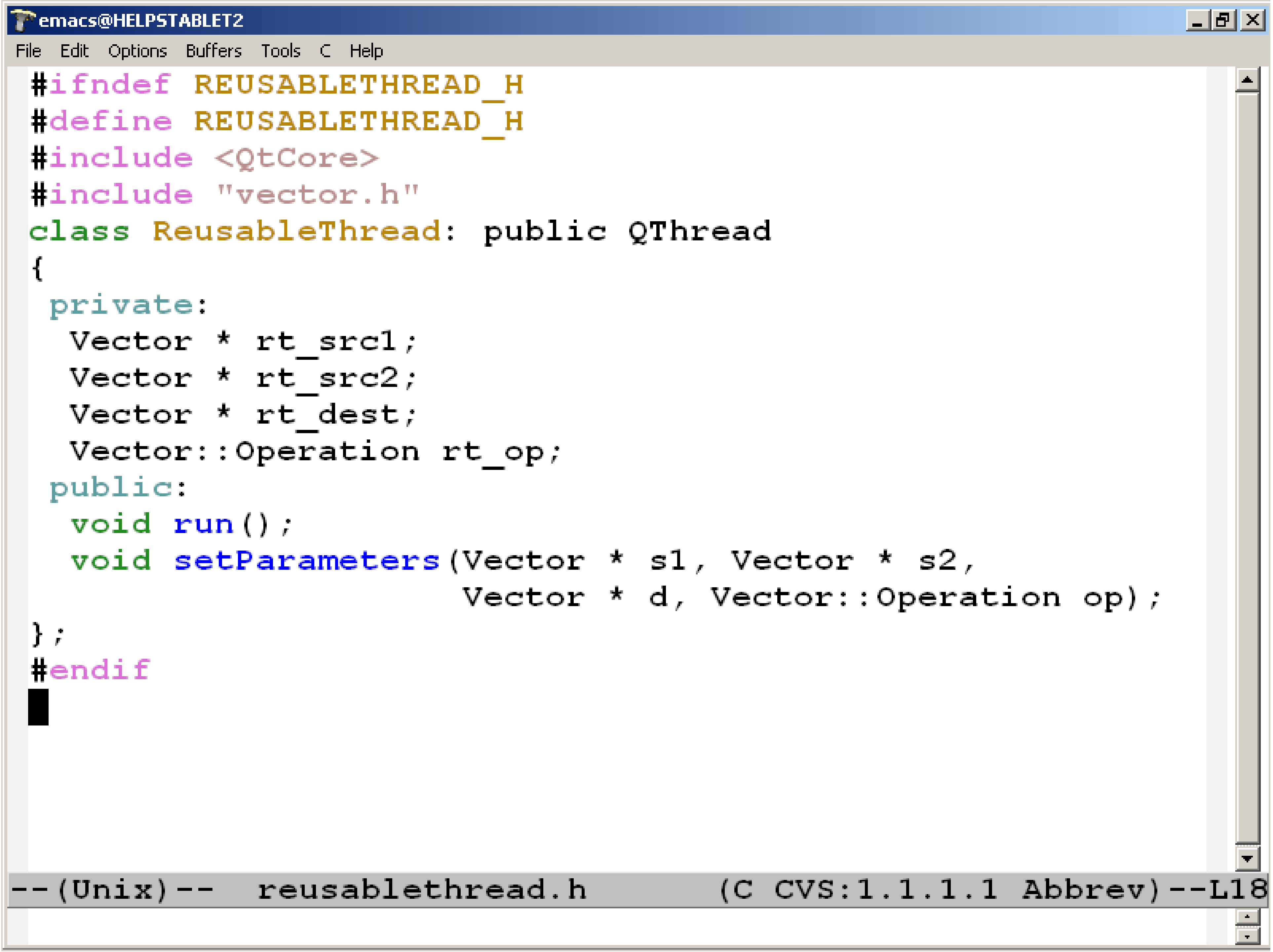

emacs@HELPSTABLET2
File
Edit
Options
Buffers
Tools
C++
Help
--(Unix)--   vector.cpp   (C++ Abbrev)--L53--39%---------

```
bool Vector::checkSize(Vector * v2, Vector * vdest)
{
  if (v_size != (v2 -> v_size))
    {
      cout << "vectors of different sizes" << v_size
           << " " << (v2 -> v_size) << endl;
      return false;
    }
  if (v_size != (vdest -> v_size))
    {
      delete vdest;
      vdest = new Vector(v_size);
    }
  return true;
}
void Vector::add(Vector * v2, Vector * vdest)
{
  if (checkSize(v2, vdest) == false) { return; }
  for (int ecnt = 0; ecnt < v_size; ecnt ++)
    {
      vdest -> v_data[ecnt] = v_data[ecnt] +
        (v2 -> v_data)[ecnt];
```



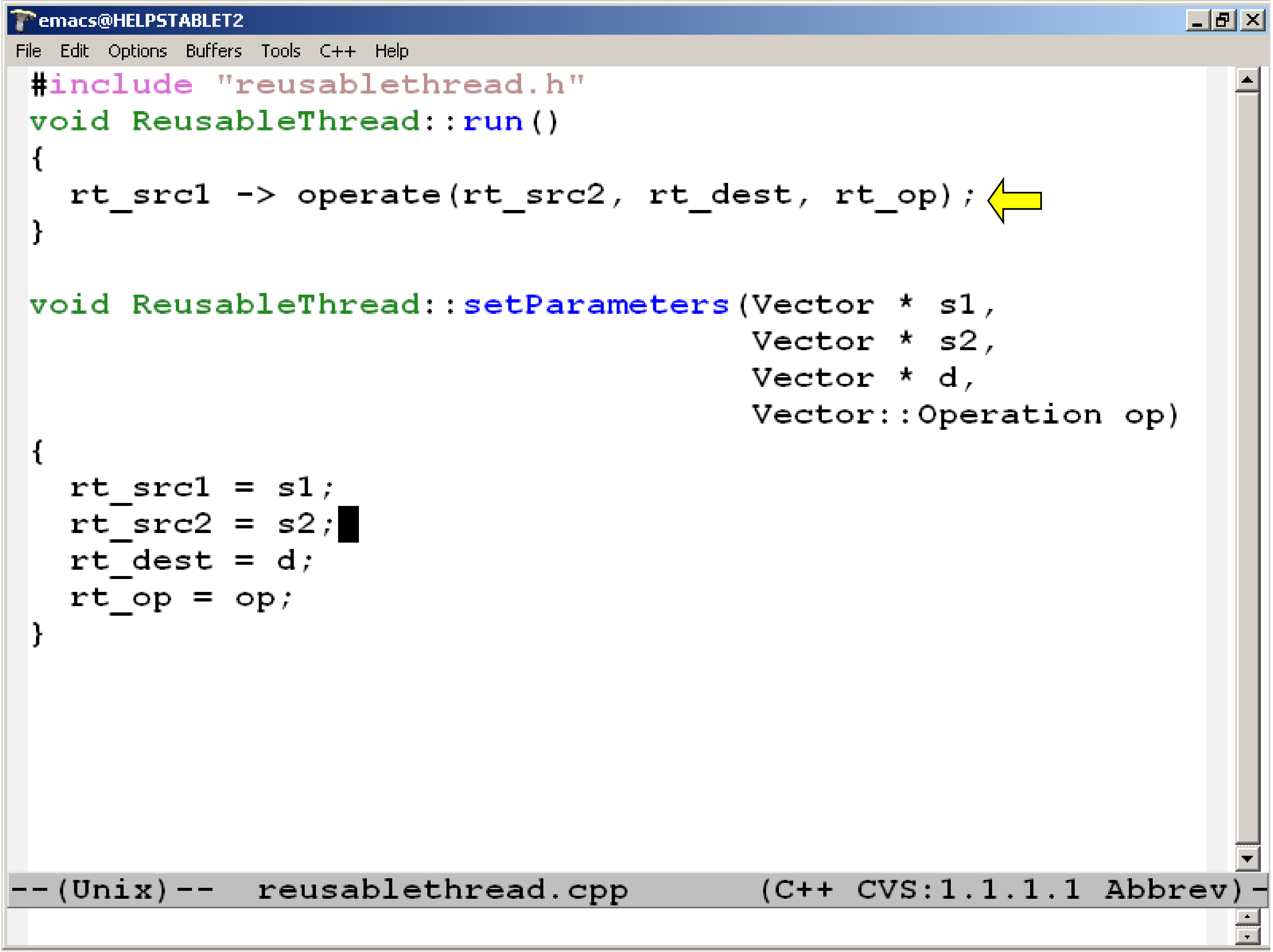
```
void Vector::subtract(Vector * v2, Vector * vdest)
{
  if (checkSize(v2, vdest) == false) { return; }
  for (int ecnt = 0; ecnt < v_size; ecnt ++)
    {
      vdest -> v_data[ecnt] = v_data[ecnt] -
        (v2 -> v_data)[ecnt];
    }
}

void Vector::print()
{
  for (int ecnt = 0; ecnt < v_size; ecnt ++)
    {
      cout << v_data[ecnt] << " ";
    }
  cout << endl << endl;
}
```

```
--(Unix)--  vector.cpp         (C++ Abbrev)--L70--Bot---------
```



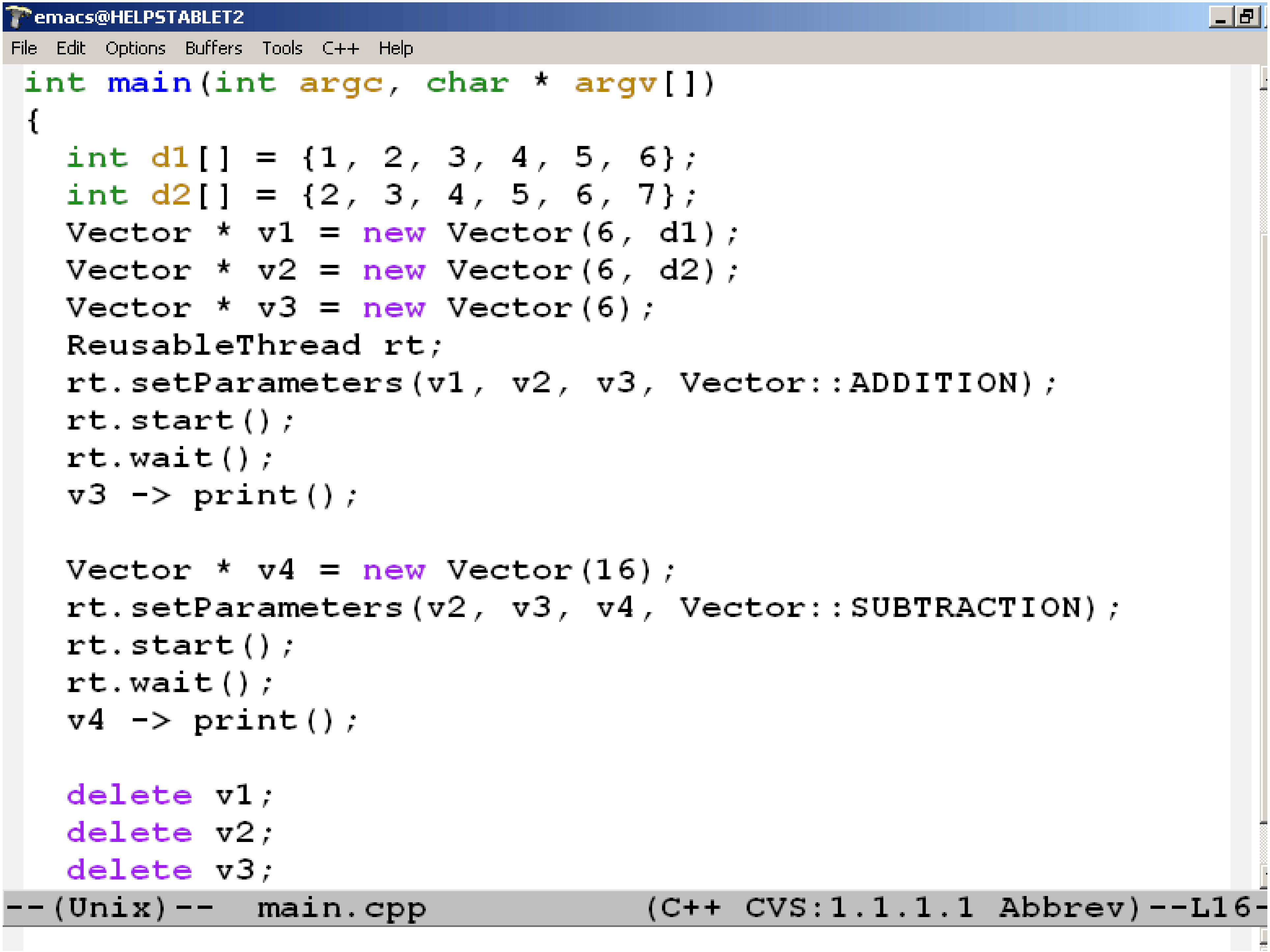
```
#ifndef REUSABLETHREAD_H
#define REUSABLETHREAD_H
#include <QtCore>
#include "vector.h"
class ReusableThread: public QThread
{
 private:
  Vector * rt_src1;
  Vector * rt_src2;
  Vector * rt_dest;
  Vector::Operation rt_op;
 public:
  void run();
  void setParameters(Vector * s1, Vector * s2,
                     Vector * d, Vector::Operation op);
};
#endif
```

```
--(Unix)--  reusablethread.h      (C CVS:1.1.1.1 Abbrev)--L18
```



```
#include "reusablethread.h"
void ReusableThread::run()
{
  rt_src1 -> operate(rt_src2, rt_dest, rt_op);
}

void ReusableThread::setParameters(Vector * s1,
                                   Vector * s2,
                                   Vector * d,
                                   Vector::Operation op)

{
  rt_src1 = s1;
  rt_src2 = s2;
  rt_dest = d;
  rt_op = op;
}
```

```
--(Unix)--  reusablethread.cpp      (C++ CVS:1.1.1.1 Abbrev)-
```


emacs@HELPSTABLET2
File
Edit
Options
Buffers
Tools
C++
Help
--(Unix)--   main.cpp   (C++ CVS:1.1.1.1 Abbrev)--L16-

```
int main(int argc, char * argv[])
{
  int d1[] = {1, 2, 3, 4, 5, 6};
  int d2[] = {2, 3, 4, 5, 6, 7};
  Vector * v1 = new Vector(6, d1);
  Vector * v2 = new Vector(6, d2);
  Vector * v3 = new Vector(6);
  ReusableThread rt;
  rt.setParameters(v1, v2, v3, Vector::ADDITION);
  rt.start();
  rt.wait();
  v3 -> print();

  Vector * v4 = new Vector(16);
  rt.setParameters(v2, v3, v4, Vector::SUBTRACTION);
  rt.start();
  rt.wait();
  v4 -> print();

  delete v1;
  delete v2;
  delete v3;
```

When to Reuse Threads

- Threads do identical or similar work in different parts of the program.
- The overhead of creating and destroying threads is too high.
- The overhead of assigning parameters is too high.

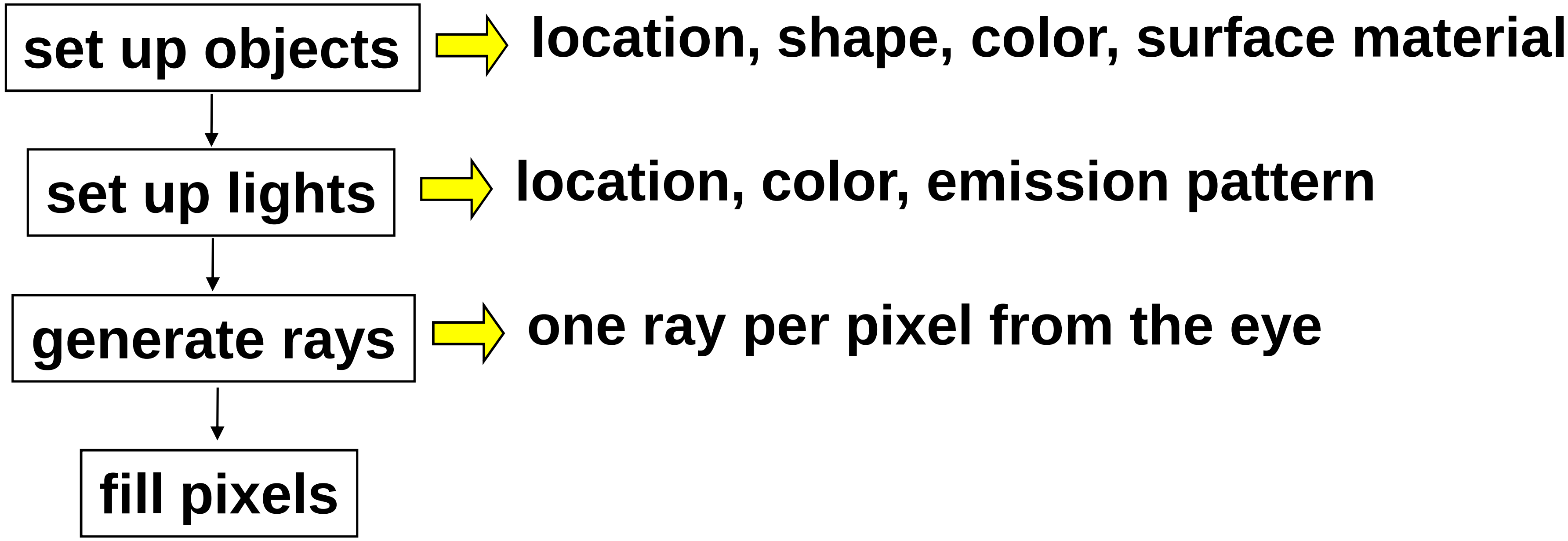
ECE 462
Object-Oriented Programming using C++ and Java

Ray Tracer

Yung-Hsiang Lu

yunglu@purdue.edu

Structure of a Ray Tracer

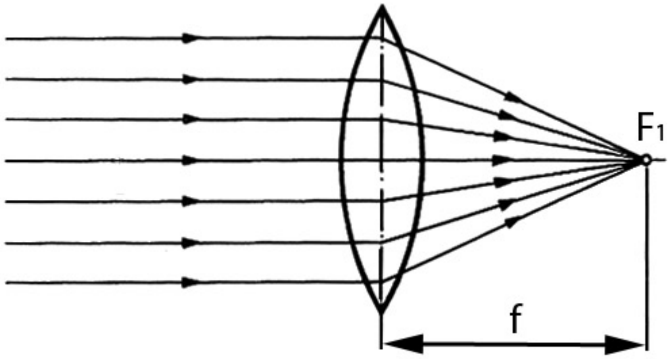


set up objects → location, shape, color, surface material

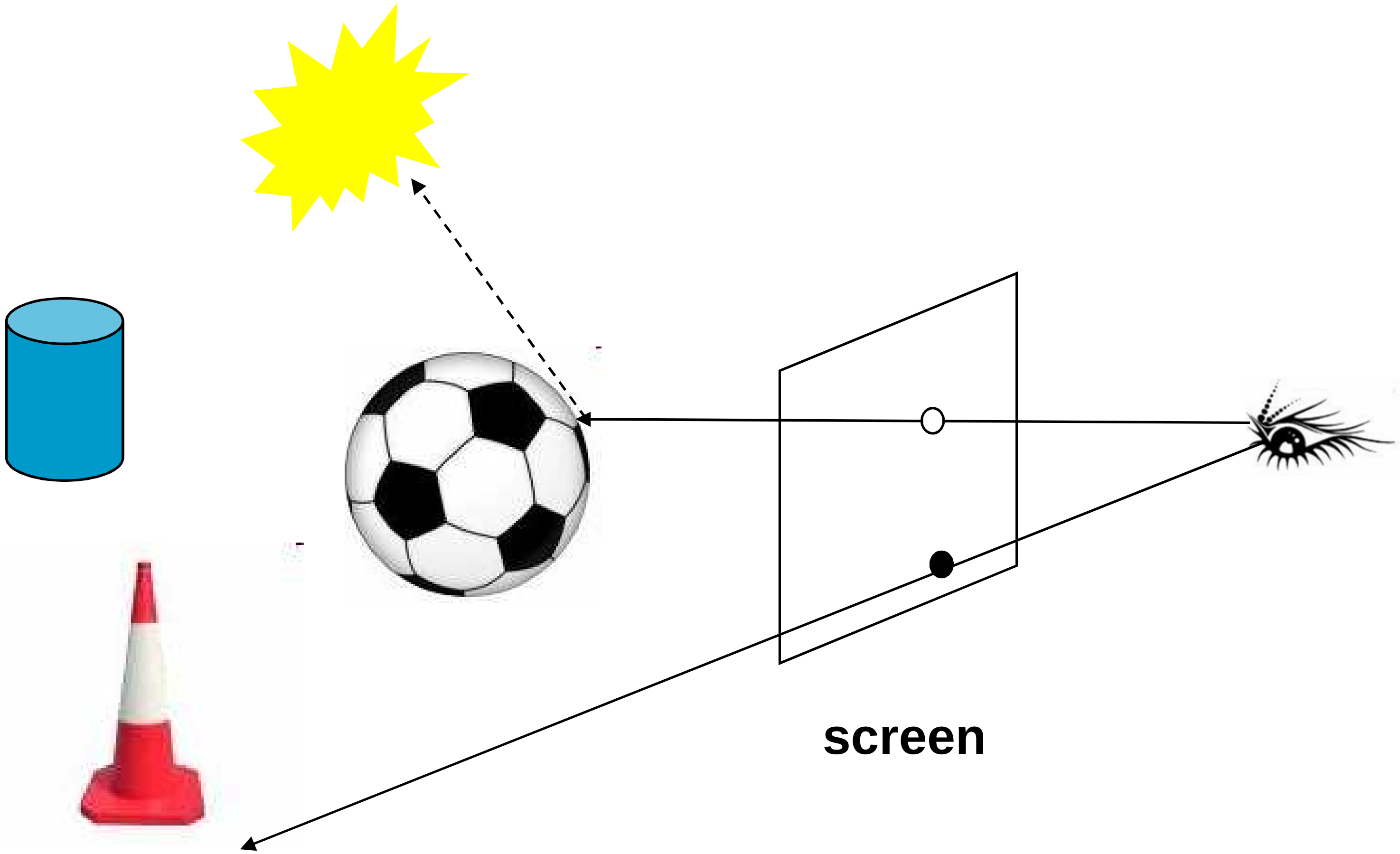
set up lights → location, color, emission pattern

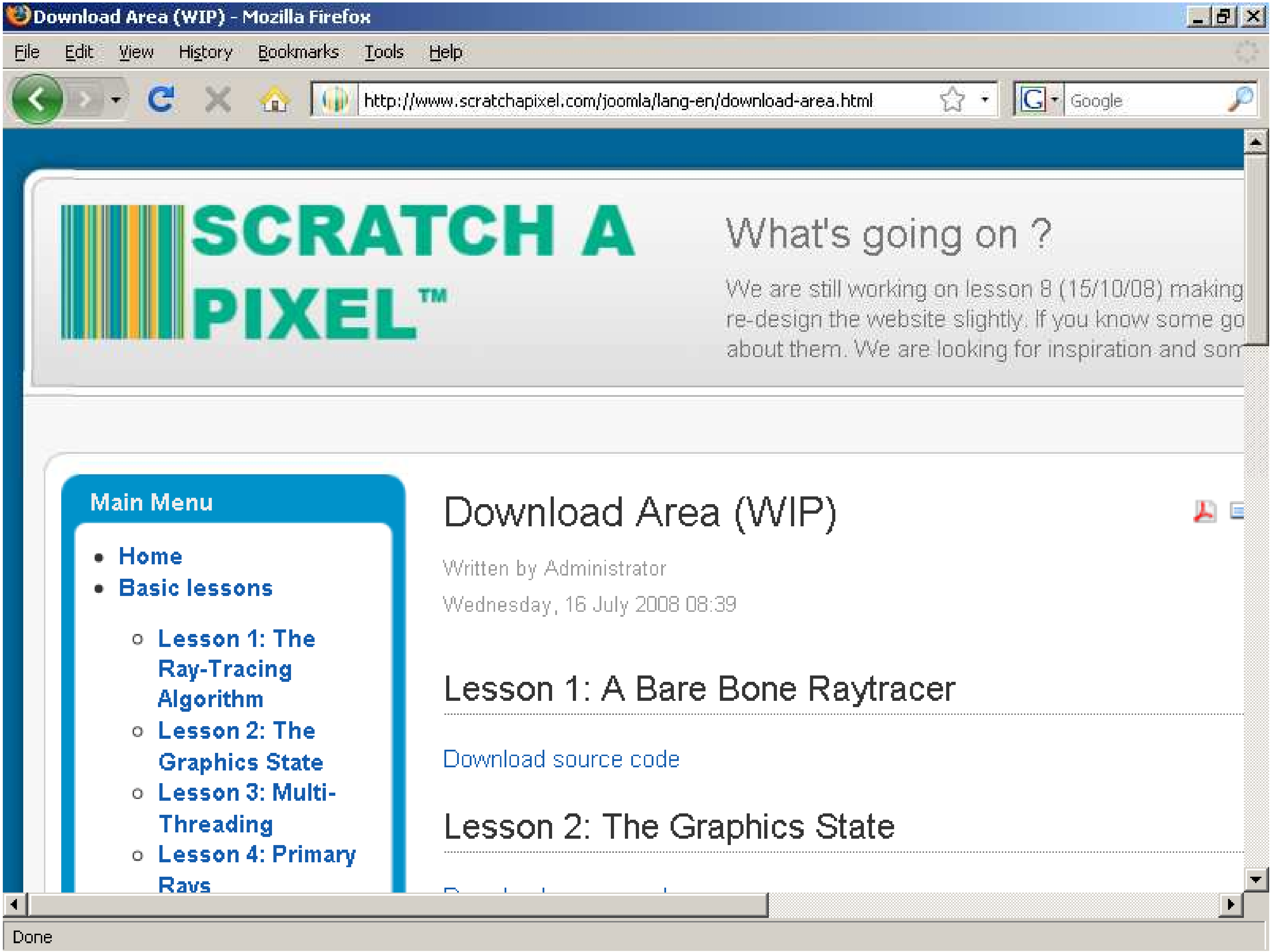
generate rays → one ray per pixel from the eye

fill pixels


F1
f

Reverse Ray Tracing





http://www.scratchapixel.com/joomla/lang-en/download-area.html

SCRATCH A PIXEL™

What's going on ?

We are still working on lesson 8 (15/10/08) making
re-design the website slightly. If you know some go
about them. We are looking for inspiration and son

Main Menu

- Home
- Basic lessons
  - Lesson 1: The Ray-Tracing Algorithm
  - Lesson 2: The Graphics State
  - Lesson 3: Multi-Threading
  - Lesson 4: Primary Rays

Download Area (WIP)

Written by Administrator

Wednesday, 16 July 2008 08:39

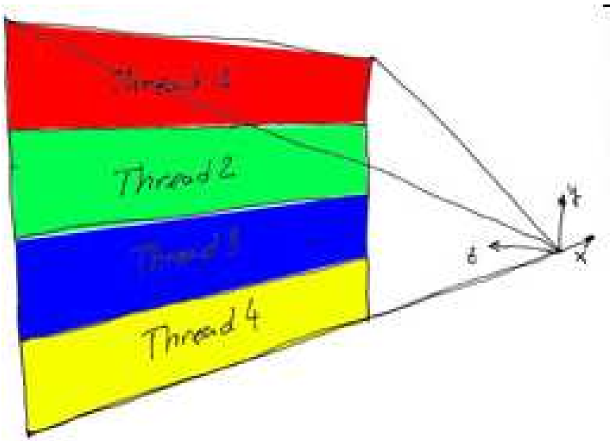
Lesson 1: A Bare Bone Raytracer

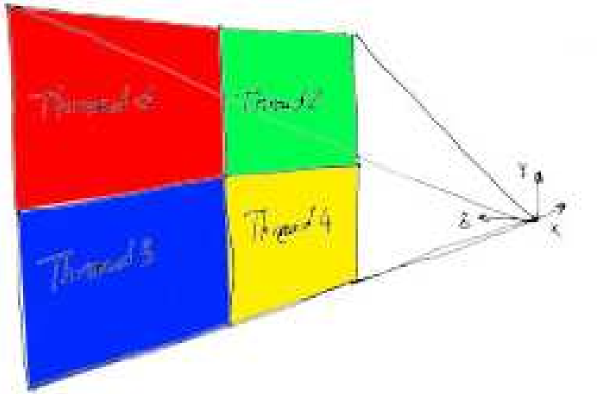
Download source code

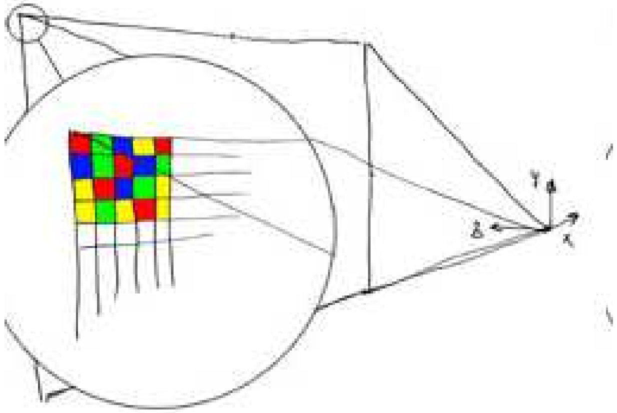
Lesson 2: The Graphics State

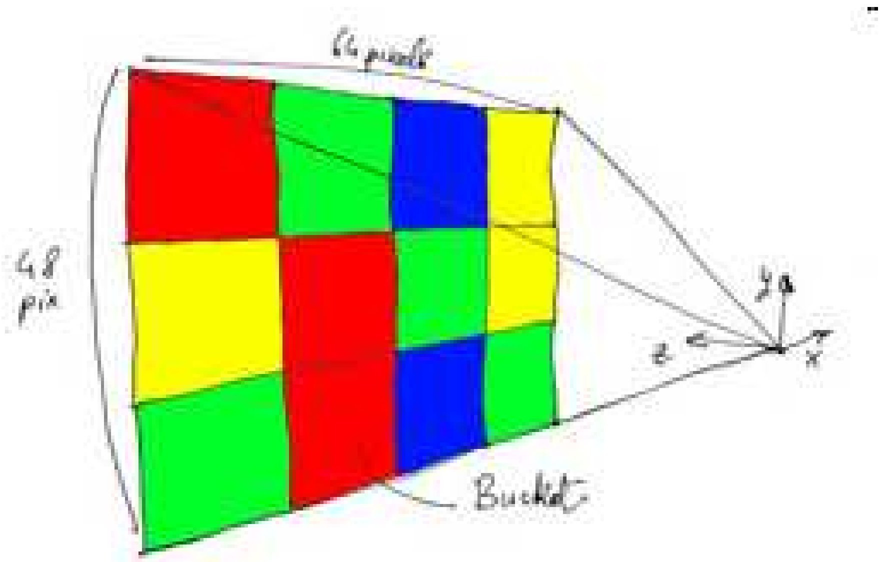
Parallelize Ray Tracer

- divide the pixels into different regions
- assign each region to one thread
- read shared data: the scene description is shared by all threads, read only
- write each pixel exclusively: each pixel is written by one and only one thread


Thread 1
Thread 2
Thread 3
Thread 4
y
z
x


Thread 1
Thread 2
Thread 3
Thread 4
y
z
x


y
z
x

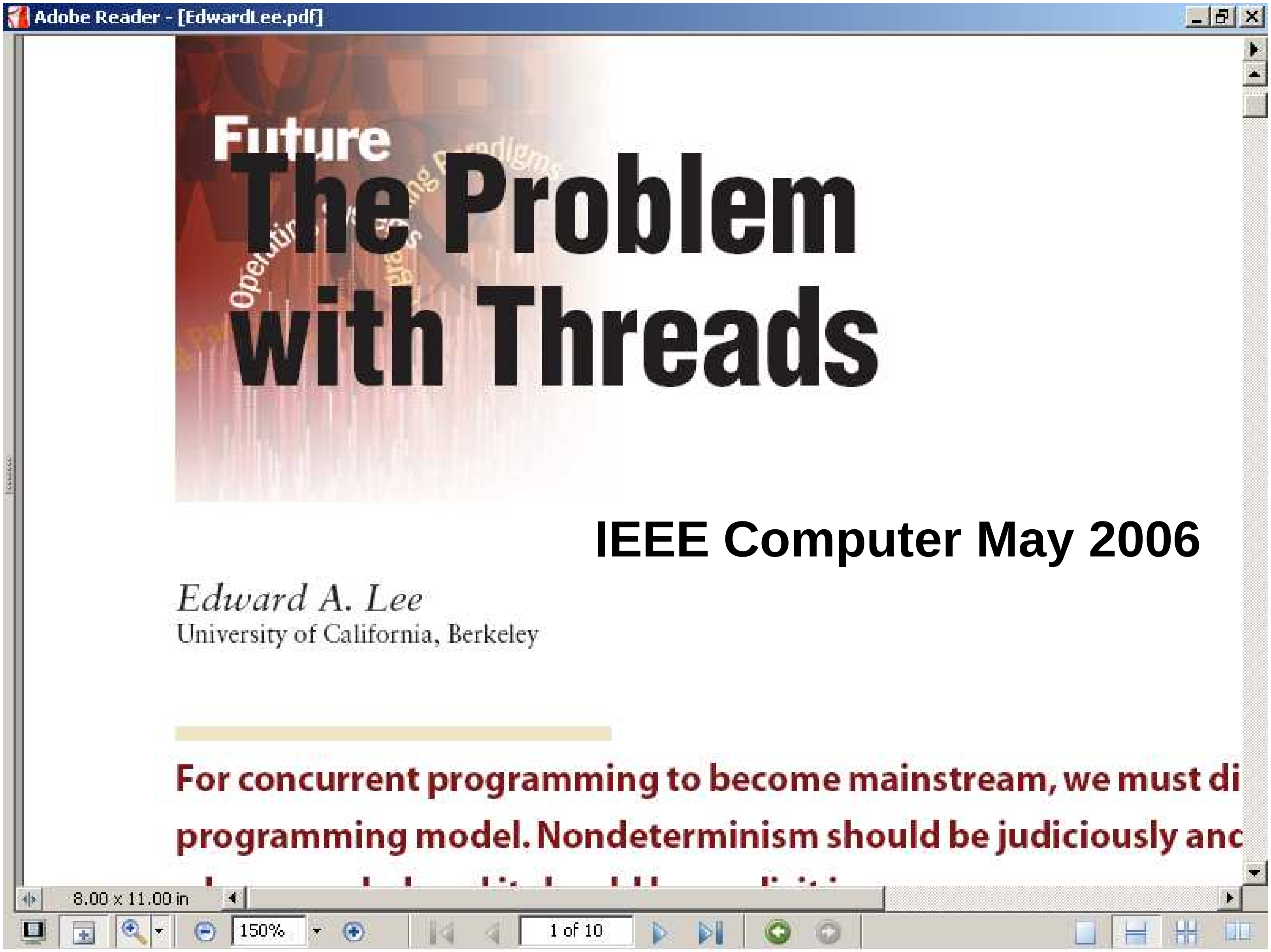

64 pixels
48 pix
Bucket
y
z
x

ECE 462
Object-Oriented Programming using C++ and Java

Problems with Threads

Yung-Hsiang Lu

yunglu@purdue.edu



Future

The Problem with Threads

IEEE Computer May 2006

Edward A. Lee
University of California, Berkeley

For concurrent programming to become mainstream, we must di
programming model. Nondeterminism should be judiciously anc

Parallel Programming

- Parallel programming on desktop, laptop, and palm computers are "real". Multi-core processors are now standard in most new computers.
- Parallel processing has been provided by hardware using pipeline, VLIW, superscalar (ECE 437).
- Automatically converting sequential programs (parallelizing compilers) is not mature.
- Programmers, in the foreseeable future, have to write parallel programs explicitly.
- Threads are **one** popular approach for parallel programming but **threads have serious problems**.

What is wrong with Threads?

- interleaving: there is **no guarantee about the orders** of threads' execution
- worse: **different results may occur** after executing the same program with the same inputs
- **Synchronization** (lock, conditional wait) is provided to prevent undesired results.
- This is a **wrong approach** (by Edward Lee). Threads assume no guarantee of ordering and some possible interleavings are removed by enforcing atomicity.

Synchronization

- Problems of programmer-inserted synchronization
  - too many: slow down the program
  - too few: incorrect
  - no easy way to analyze or detect deadlocks
- Bugs are probably common but they have not detected because most computers, so far, have only single processors. When multi-cores are widely used, more bugs may be discovered.
- It is not easy to create correct synchronization. Locks are too low-level for many programmers.

Future Parallel Programming

- Why does ECE 462 teach threads only? This is the **starting point** for you to learn other ways of parallel programming.
- Alternatives
  - different programming languages
  - different programming models (such as transactions)