An Introduction to the Kent C++CSP Library

Authors

Neil Brown (neil@twistedsquare.com)

Peter Welch (P.H.Welch@kent.ac.uk)

C++ Overview

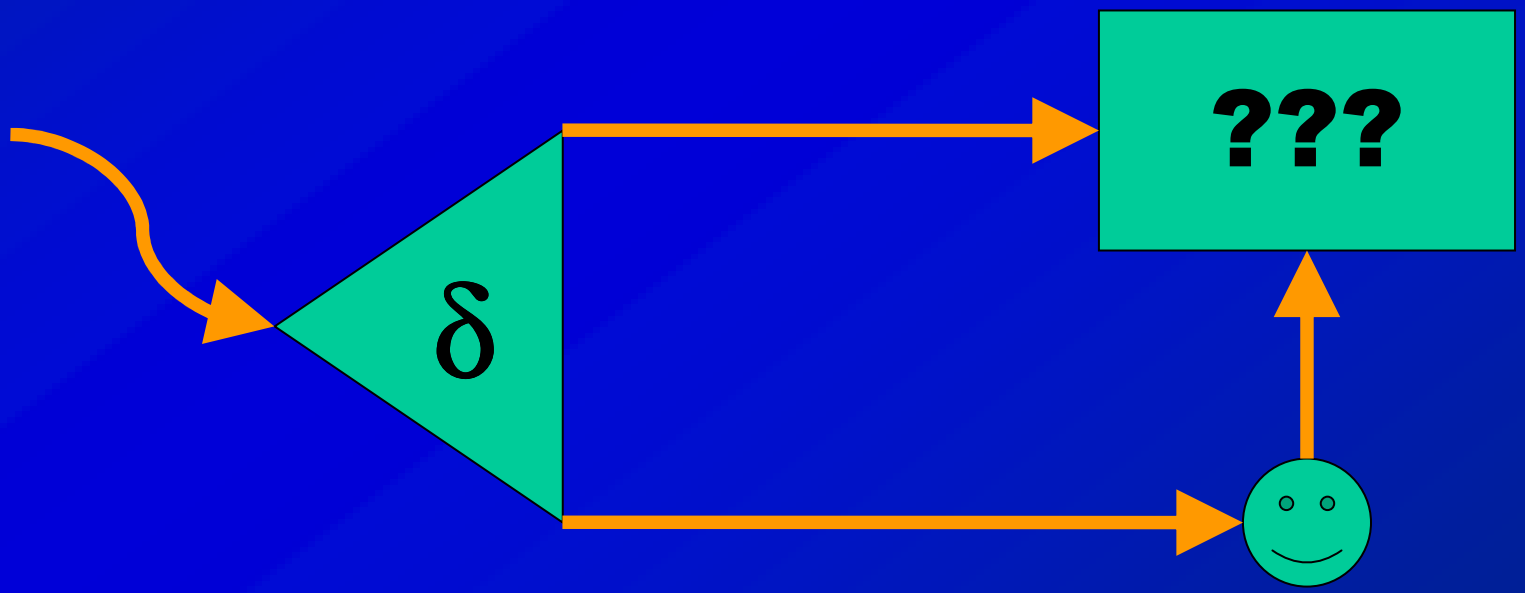
- Object-oriented but not “pure” object-oriented like Smalltalk
- “C++ provides elegant solutions to complex problems”
- “C with classes bolted on”
- Almost as fast as C, but with better structure

CSP (Quick!) Overview

- Processes, channels, concurrency etc.
- Very scaleable without becoming over-complex
- Easy to visualise and understand

CSP (Quick!) Overview

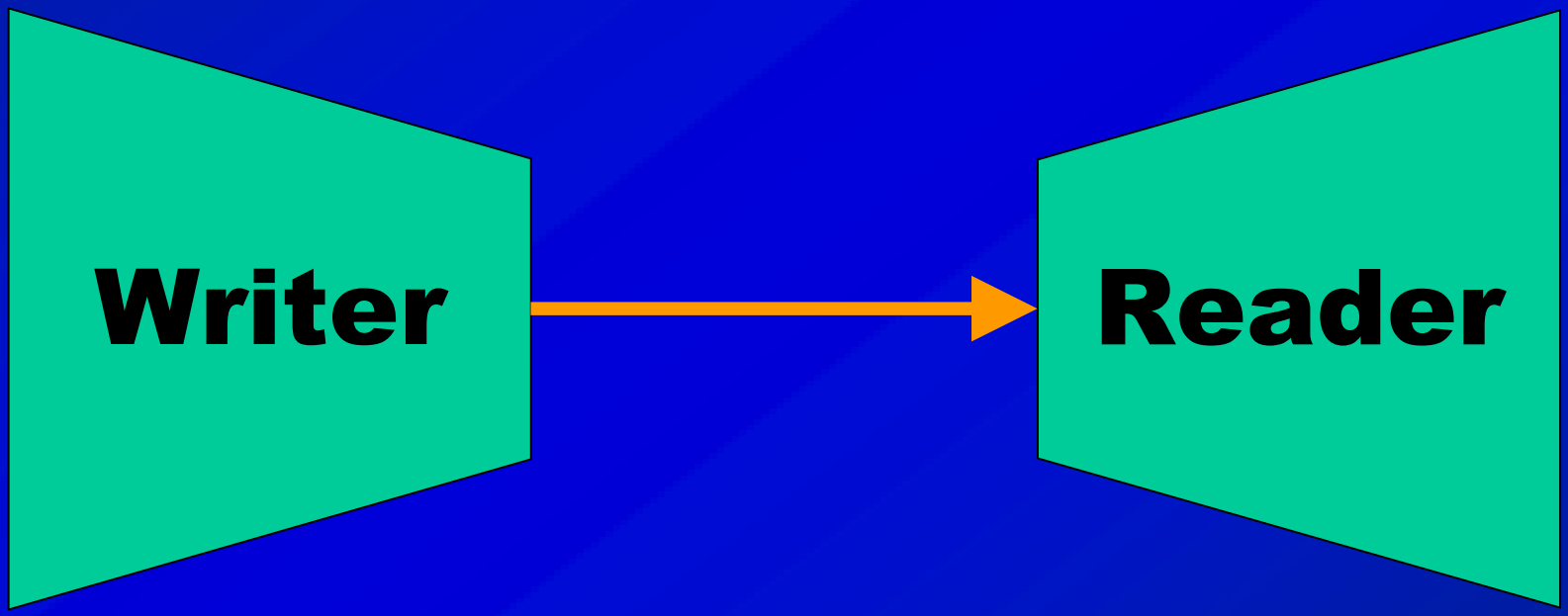
- Processes, channels, concurrency etc.
- Very scaleable without becoming over-complex
- Easy to visualise and understand



Library Overview

- A process is a subclass of `CSProcess`, implementing the `run` method
- Channels are templated, so they can communicate any type
- Communication is done using channel end objects, not the channel itself

Our First Processes



Our First Processes

```
class Writer
   :  public CSProcess
{
private:
   Chanout<int> out;
protected:
   void run();
public:
   Writer(
      const Chanout<int>& o
   );
};
```

Chanout<int> is the writing end of a channel of integers

Our First Processes

```
class Reader
   :  public CSProcess
{
private:
  Chanin<int> in;
protected:
  void run();
public:
  Reader(
    const Chanin<int>& i
  );
};
```

Chanin<int> is the reading end of a channel of integers

Our First Processes

```
void Writer::run() {
  int i;
  for (i = 0;i < 100;i++) {
    out << i;
  }
}

void Reader::run() {
  int i,n;
  for (i = 0;i < 100;i++) {
    in >> n;
  }
}
```

Input and output are done with simple operators, as in *occam*

Our First Processes

Our First Processes

```
void function()
{
  One2OneChannel<int> channel;

  Parallel(
    new Writer(channel.writer()),
    new Reader(channel.reader()),
  NULL);
}

void main() {
  Start_CSP();
  function();
  End_CSP();
}
```

Where appropriate, JCSP's API is copied

Use `writer/reader` calls to get channel ends

`Parallel` takes a `NULL`-terminated list of process pointers

The `Start_CSP/End_CSP` functions must be called before/after the library is used

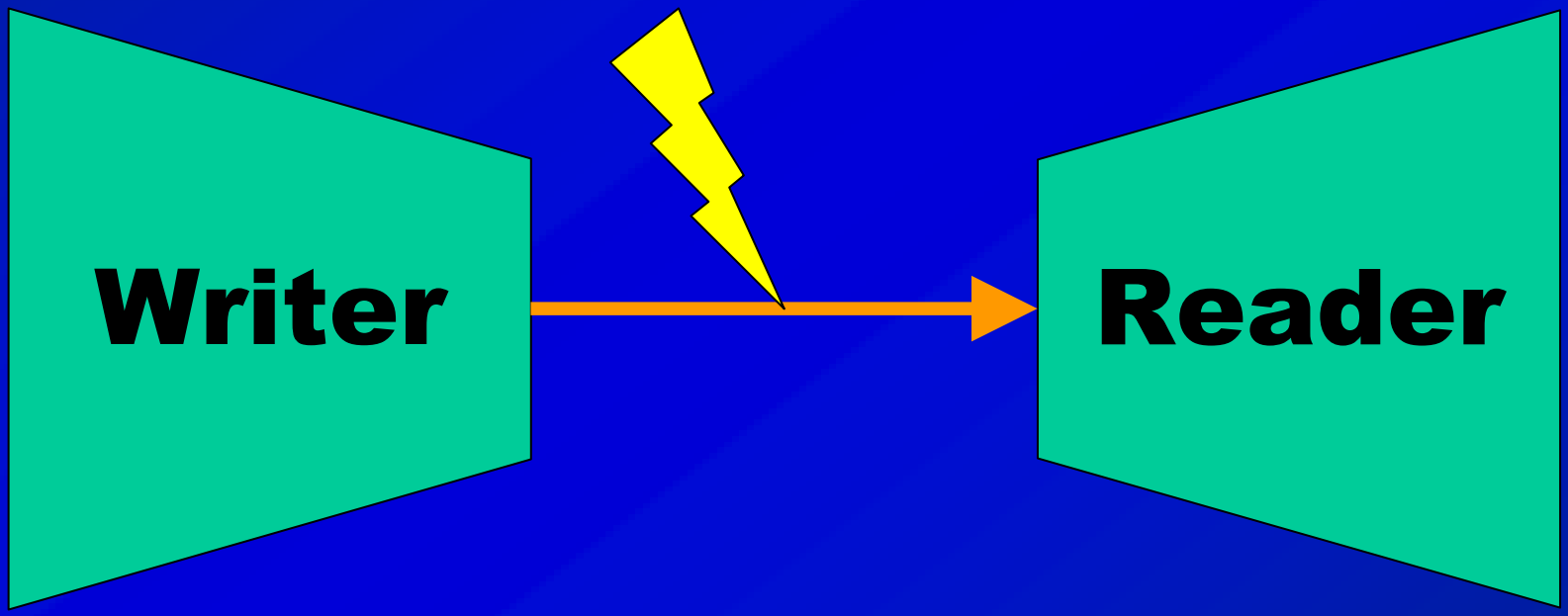
Our First Processes

```
void Writer::run() {
  int i;
  for (i = 0;i < 100;i++) {
    out << i;
  }
}

void Reader::run() {
  int i,n;
  for (i = 0;i < 100;i++) {
    in >> n;
  }
}
```

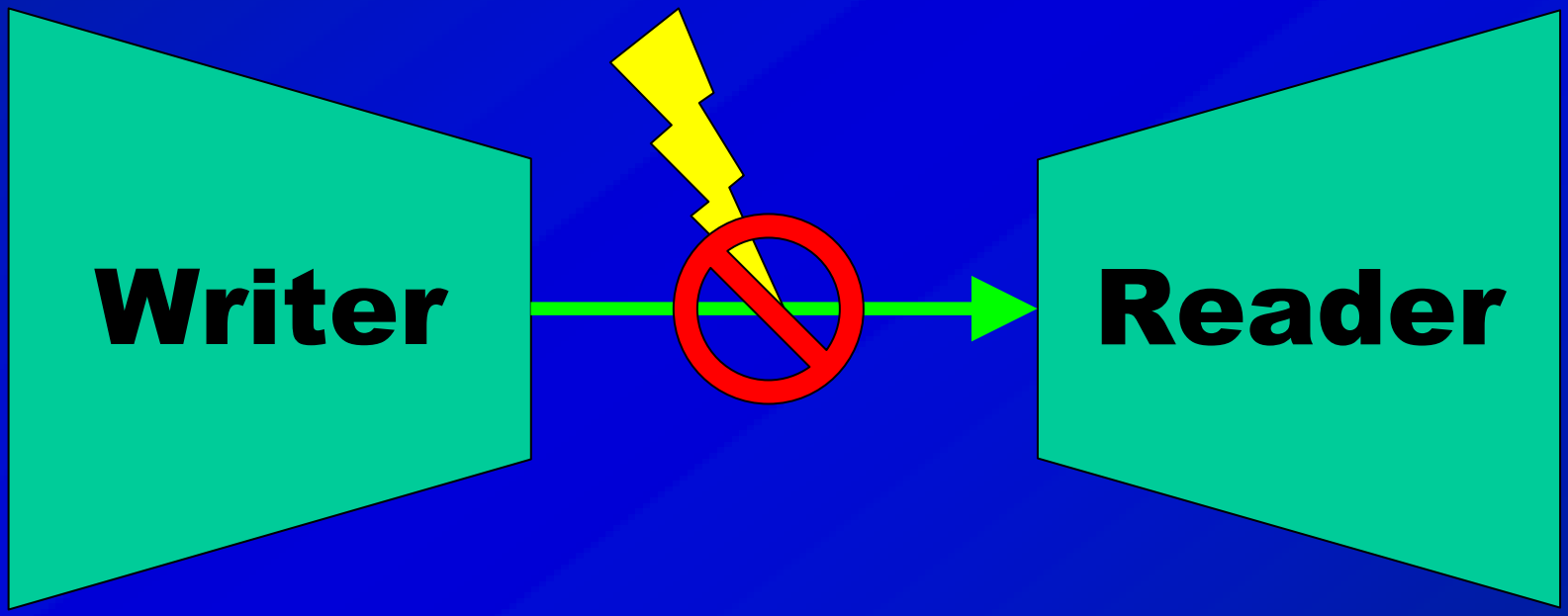
The for loop is inflexible - maybe there is another way? ....

Poisoning

Poisoning

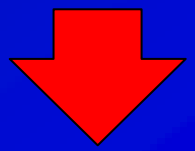


Poisoning



Poisoning

```
void Writer::run() {
  int i;
  for (i = 0;i < 100;i++) {
    out << i;
  }
}
```



```
void Writer::run() {
  int i;
  for (i = 0;i < 100;i++) {
    out << i;
  }
  out.poison();
}
```

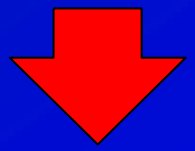
This channel will now be poisoned (forever - no antidote available!).

Any attempts to use the channel will cause a `PoisonException` to be thrown.

<u>Except:</u> poisoning an already-poisoned channel has no effect (no exception is thrown).

Poisoning

```
void Reader::run() {
  int i,n;
  for (i = 0;i < 100;i++) {
    in >> n;
  }
}
```



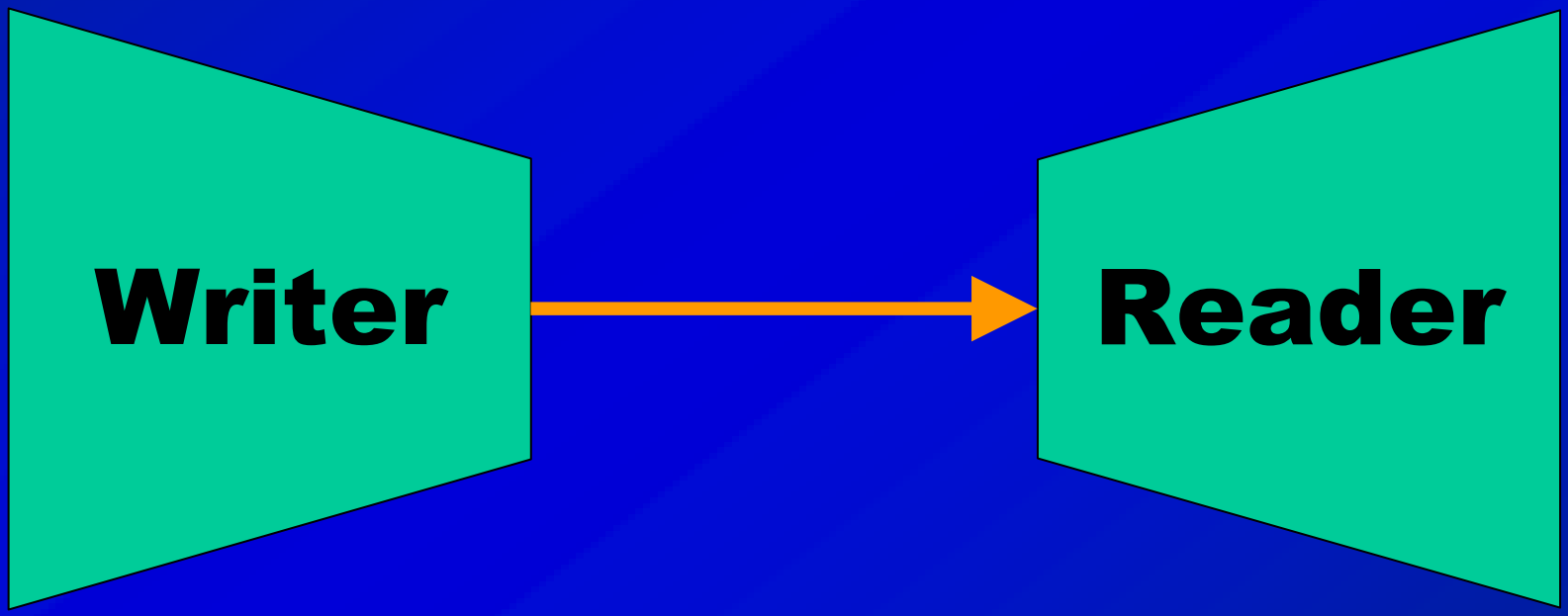
```
void Reader::run() {
  try {
    int n;
    while (true) {
      in >> n;
    }
  }
  catch (PoisonException e) {
  }
}
```

The f
now
while

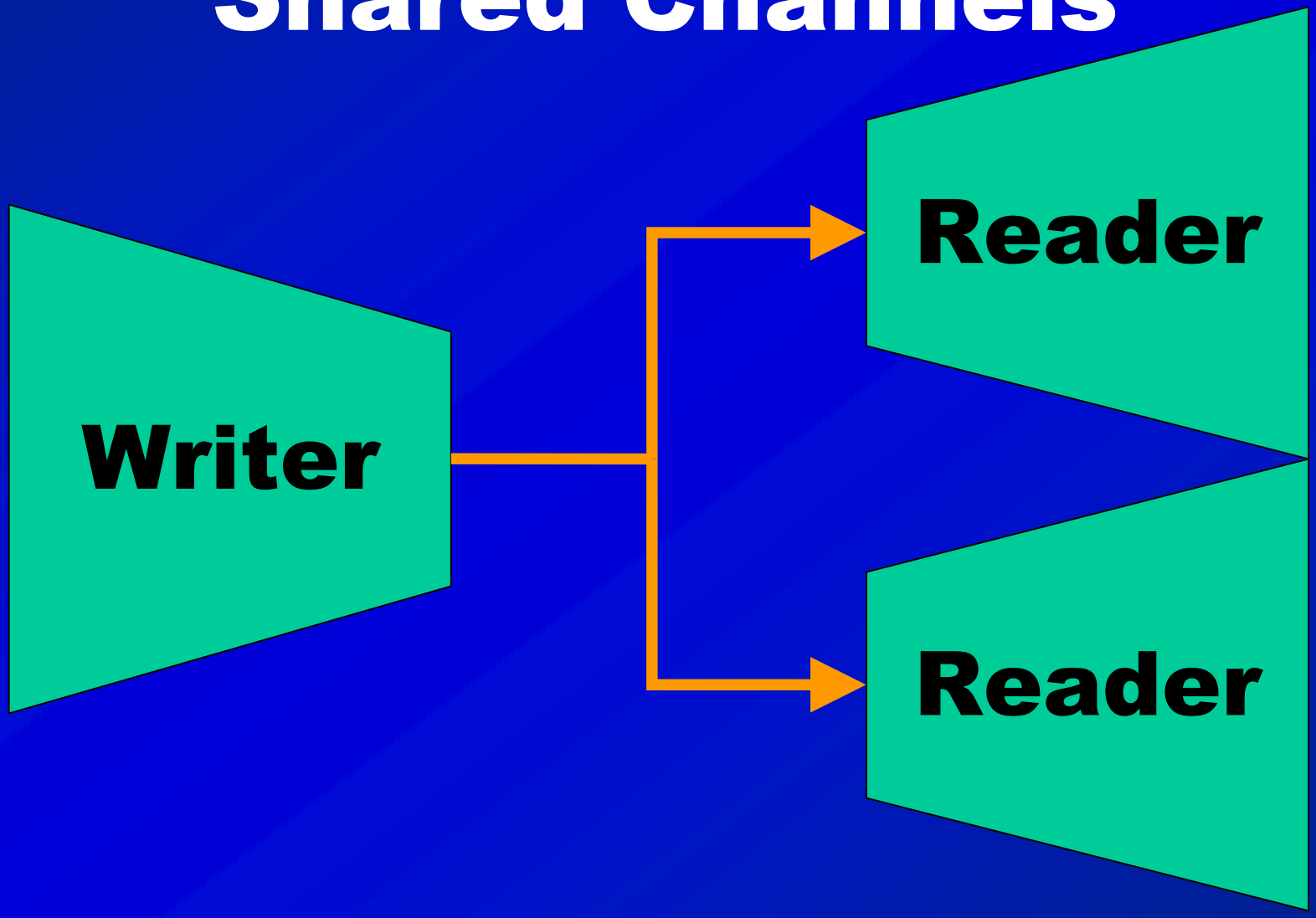
When the channel has been poisoned, the next input will cause a PoisonException to be thrown, and it will be caught here

Shared Channels

Shared Channels

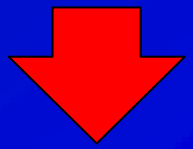


Shared Channels



Shared Channels

```
One2OneChannel<int> channel;

Parallel(
  new Writer(channel.writer()),
  new Reader(channel.reader()),
NULL);
```

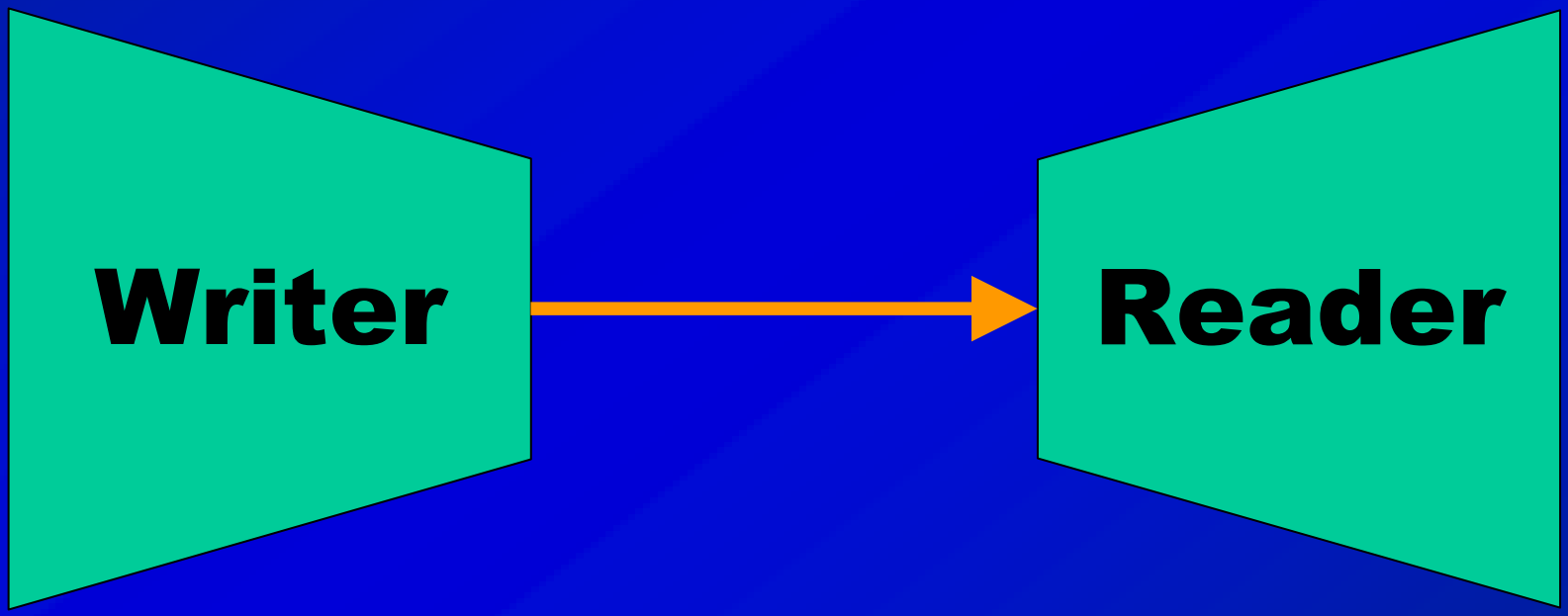


```
One2AnyChannel<int> channel;

Parallel(
  new Writer(channel.writer()),
  new Reader(channel.reader()),
  new Reader(channel.reader()),
NULL);
```

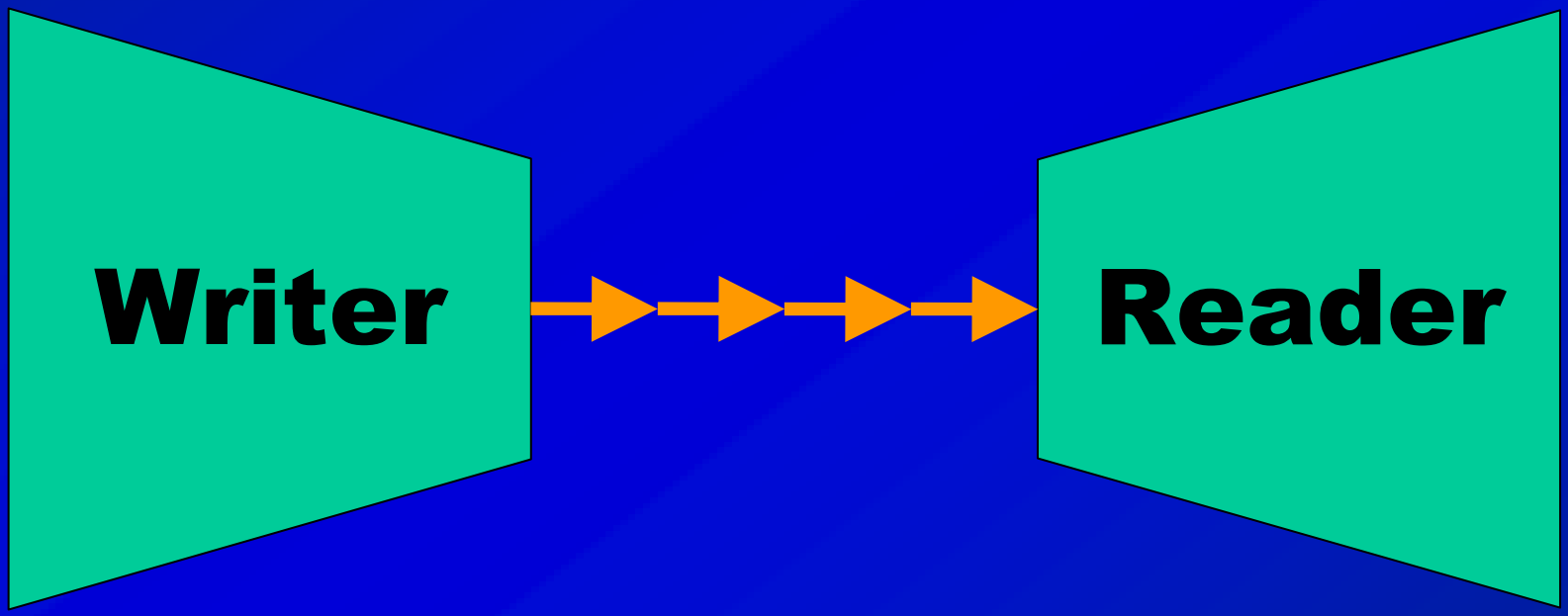
Change the channel type and add an extra reader - simple as that!

Buffered Channels

Buffered Channels

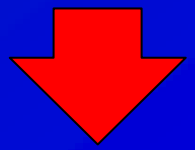


Buffered Channels



Buffered Channels

```
One2OneChannel<int> channel;

Parallel(
  new Writer(channel.writer()),
  new Reader(channel.reader()),
NULL);
```



```
One2OneChannelX<int>
  channel( Buffer<int>(4) );

Parallel(
  new Writer(channel.writer()),
  new Reader(channel.reader()),
NULL);
```

Pass in the buffer object you want to use for buffering - like JCSP, C++CSP takes a copy of it rather than use the original

Not
adde

Parallel Communications

```
void Delta::run() {
  int n;
  ParallelComm pc(
    outA.parOut(&n),
    outB.parOut(&n),
  NULL);

  while (true) {
    in >> n;
    pc.communicate();
  }
}
```

Like other library calls, the

This communicate call performs all the communications associated with the ParCommItems passed to the constructor of the ParallelComm

respectively as an argument

Alternatives (ALTing)

```
void Alter::run() {
  /*declarations*/
  int n;
  Time t = Seconds(1);
  Alternative alt(
    inA.inputGuard(&n),
    inB.inputGuard(&n),
    new RelTimeoutGuard(t),
  NULL);

  /*.. main body ..*/
}
```

The `inputGuard` method of `Chanin` returns a `Guard` object to be used with the `Alternative` class

`inputGuard` takes the destination of the input as an argument

`RelTimeoutGuard` is a library class to provide a timeout guard relative to the start of the `ALT`. An absolute timeout guard is also provided

Alternatives (ALTing)

```
void Alter::run() {
  /*declarations*/
  int n;
  Time t = Seconds(1);
  Alternative alt(
    inA.inputGuard(&n),
    inB.inputGuard(&n),
    new RelTimeoutGuard(t),
  NULL);

  /*.. main body ..*/
}
```

Alternatives (ALTing)

```
void Alter::run() {
  /*.. declarations ..*/

  /*main body*/
  while (true) {
    switch (alt.priSelect()){
    case 0: /*input on A*/
    case 1: /*input on B*/
    case 2: /*timeout*/
    }
  }
}
```

The actions associated with an ALT (i.e. inputs) take place during the priSelect method, and are not done separately afterwards like in JCSP

So by the time this line gets executed, the input will have been successfully completed

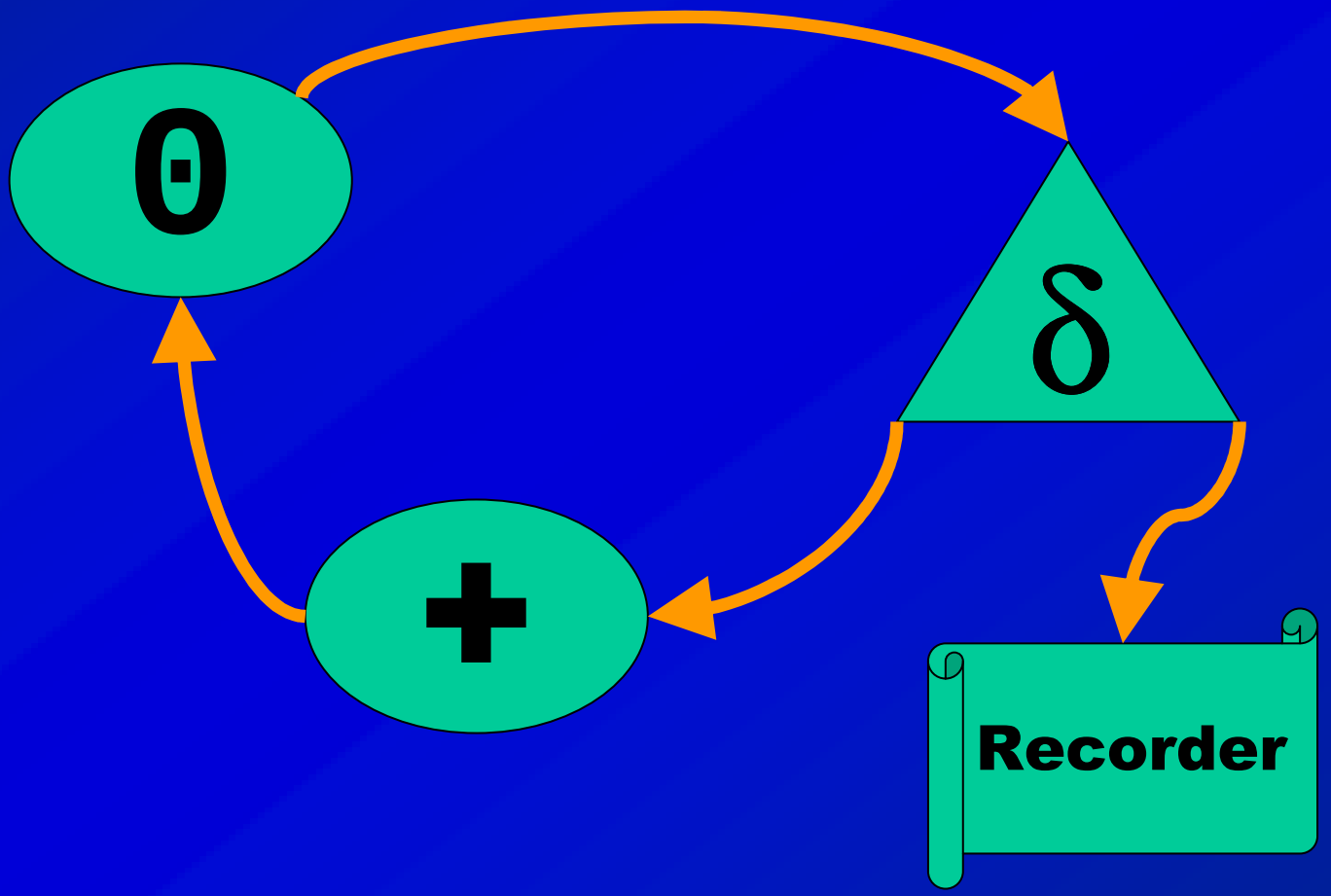
Extra Features

- Extended rendezvous available for both normal input operations and input guards in ALTs
- Templated “smart” pointer class `Mobile<T>` that has the same semantics as KRoC’s `MOBILEs`
- KRoC Interoperability using KRoC’s UDC mechanism -- both the KRoC and C++CSP program use their native channels

Technical Details

- The library uses cooperatively multitasked threading (in a single operating-system thread), similar to KRoC
- The scheduler kernel is highly portable - the library currently functions under Windows and most Unixes
- Unfortunately, processes must specify the size of the stack they want

CommsTime



CommsTime

CSP Framework

Time per iteration

CommsTime

CSP Framework	Time per iteration
occam (KRoC 1.3.3)	1.3 microseconds

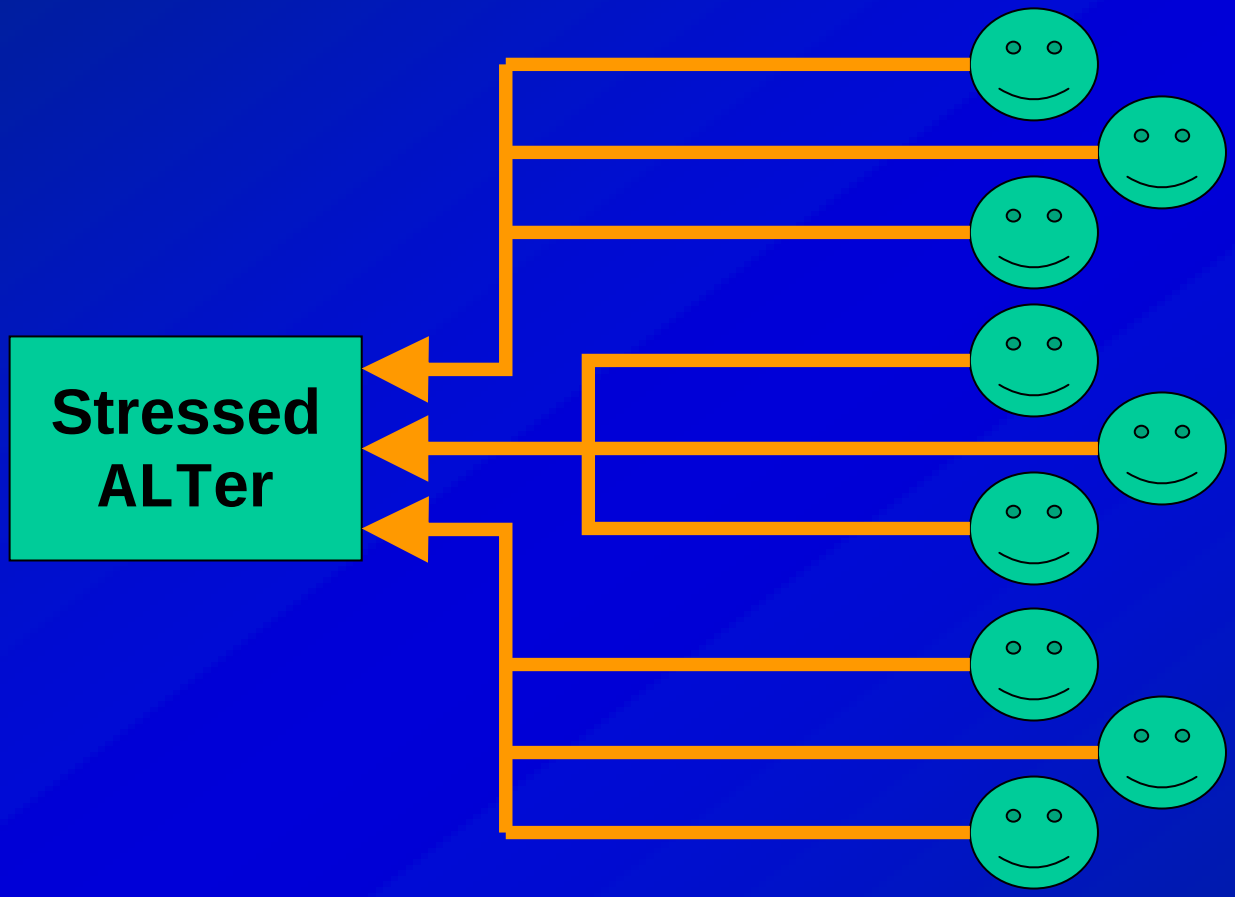
CommsTime

CSP Framework	Time per iteration
occam (KRoC 1.3.3)	1.3 microseconds
C++CSP	5 microseconds

CommsTime

CSP Framework	Time per iteration
occam (KRoC 1.3.3)	1.3 microseconds
C++CSP	5 microseconds
JCSP (JDK 1.4)	230 microseconds

Stressed ALT



Stressed ALT

CSP Framework	10 writers * 10 channels	100 writers * 20 channels	200 writers * 100 channels

Stressed ALT

CSP Framework	10 writers * 10 channels	100 writers * 20 channels	200 writers * 100 channels
occam (KRoC 1.3.3)	0.6	0.7	1

(All times are in microseconds)

Stressed ALT

CSP Framework	10 writers * 10 channels	100 writers * 20 channels	200 writers * 100 channels
occam (KRoC 1.3.3)	0.6	0.7	1
C++CSP	3	7	10

(All times are in microseconds)

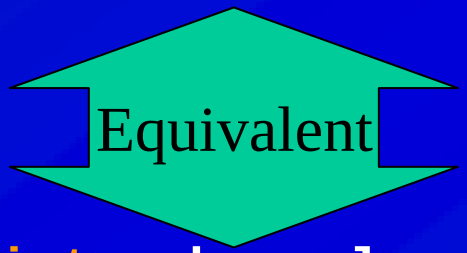
Stressed ALT

CSP Framework	10 writers * 10 channels	100 writers * 20 channels	200 writers * 100 channels
occam (KRoC 1.3.3)	0.6	0.7	1
C++CSP	3	7	10
JCSP (JDK 1.4)	130	200	-

(All times are in microseconds)

An Equivalence

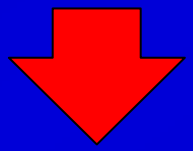
```
One2OneChannel<int> channel;
Parallel(
  new Writer(channel.writer()),
  new Reader(channel.reader()),
NULL);
```




```
One2OneChannel<int> channel;
Barrier barrier(1);

spawnProcess(new Writer(channel.writer()),&barrier);
spawnProcess(new Reader(channel.reader()),&barrier);

barrier.sync();
```

Going Multi-Threaded

```
One2OneChannel<int> channel;
Barrier barrier(1);

spawnProcess(new Writer(channel.writer()),&barrier);
spawnProcess(new Reader(channel.reader()),&barrier);

barrier.sync();
```



```
InterThreadChannel<int> channel;
InterThreadBarrier barrier(1);

spawnAsNewThread(new Writer(channel.writer()),&barrier);
spawnAsNewThread(new Reader(channel.reader()),&barrier);

barrier.sync();
```

Conclusion

- C++CSP is a new but stable library in the mould of JCSP
- It has many new useful features, such as stateful poisoning and templated channels
- It is very portable, and offers both single-threaded and multi-threaded concurrency