A <Basic> C++ Course

12 – lambda expressions and concurrency in C++11 and C++17, and C++20

Julien Deantoni

Lambda expressions

A unnamed function (which is a `std::function`)

- Usable in many place like algorithms, thread, …

`[capture] (parameters) ->return-type {body}`

- [ ] → putting objects in the scope of the lambda's body
  - [] Capture nothing
  - [&] Capture any referenced variable by reference
  - [=] Capture any referenced variable by making a copy
  - [=, &foo] Capture any referenced variable by making a copy, but capture variable foo by reference
  - [bar] Capture bar by making a copy; don't copy anything else
  - [this] Capture the this pointer of the enclosing class

Lambda expressions

A unnamed function (which is a `std::function`)

- Usable in many place like algorithms, thread, …

```
[capture] (parameters)->return-type {body}
```

- [ ] → putting objects in the scope of the lambda's body
  - [] Capture nothing
  - [&] Capture any referenced variable by reference
  - [=] Capture any referenced variable by making a copy
  - [=, &foo] Capture any referenced variable by making a copy, but capture variable foo by reference
  - [bar] Capture bar by making a copy; don't copy anything else
  - [this] Capture the this pointer of the enclosing class

Lambda expressions

A unnamed function (which is a `std::function`)

- Usable in many place like algorithms, thread, …

`[capture] (parameters)->return-type {body}`

- [ ] → putting objects in the scope of the lambda's body

```
vector<int> v;

//...
for_each( v.begin(), v.end(), [] (int val)
{
    cout << val;
});
```

KAIROS

Lambda expressions

A unnamed function (which is a `std::function`)

- Usable in many place like algorithms, thread, …

`[capture] (parameters)->return-type {body}`

- [ ] → putting objects in the scope of the lambda's body

```
vector<int> v;
ofstream f("toto.truc");
//...
for_each( v.begin(), v.end(), [&f] (int
val)
{
    f << val;
});
```

KAIROS

Lambda expressions

A unnamed function (which is a `std::function`)

- Usable in many place like algorithms, thread, …

```
[capture] (parameters)->return-type {body}
```

- [ ] → putting objects in the scope of the lambda's body

```
vector<int> v;
ofstream f("toto.truc");
//...
for_each( v.begin(), v.end(), [&f] (int
val)
{
    f << val;
});
```

Lambda expressions

A unnamed function (which is a `std::function`)

- Usable in many place like algorithms, thread, …

`[capture] (parameters) ->return-type {body}`

- [ ] → putting objects in the scope of the lambda's body

```
vector<int> v;
ofstream f("toto.truc");
//...
for_each( v.begin(), v.end(), [&f] (int
val) -> ostream&
{
    f << val;
    return f; //non-useful here...
});
```

KAIROS

Lambda expressions

A unnamed function (which is a `std::function`)

- Usable in many place like algorithms, thread, …

`[capture] (parameters)->return-type {body}`

- [ ] → putting objects in the scope of the lambda's body

```
vector<int> v;
ofstream f("toto.truc");
//...
for_each( v.begin(), v.end(), [&f] (int
val) -> ostream&
{
    f << val;
    return f; //non-useful here...
});
```

Lambda's type: `std::function<ostream& (int)>`

Function pointer and Lambda expressions

```
#include <functional>
#include <string>

class Delegator{
public:
    std::function<void (const std::string&)> handler_func;

    void process(const string& message){
        if (handler_func) {
            handler_func( message ); //call to a function pointer
        }
    }

    Delegator(std::function<void (const std::string&)> hf): handler_func(hf)
    {}
};

Delegator d1{[&](“blablabla”){dosomething usefull}};
Delegator d2{[](“/home/toto/toto.truc”){dosomething else}};
```

Concurrency in C++

- Only from C++11, still evolving
- Well adapted to the STL
    - Thread management
    - Shared data protection
    - Thread synchronization
    - …
- Different abstraction level
- Based on the boost library
- Platform independent

Concurrency in C++

C++11	C++14	C++17	C++20
2011	2014	2017	2020
• Memory model • Threads • Mutexes and Locks • Thread local data • Condition variables • Tasks	• Reader-writer locks	Parallel STL	• Atomic smart pointers • std::future extensions • Latches and barriers • Coroutines • Transactional memory • Task blocks

https://www.modernescpp.com/index.php/c-core-guidelines-rules-for-concurrency-and-parallelism

Concurrent hello world

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

void hello(){
        std::cout<<"Hello Concurrent World\n";
}

int main(){
        std::thread t(hello); //create and start a thread
        t.join(); //wait the end of the thread execution
return 0 ;
}
```

Every thread has to have an *initial function*, which is where the new thread of execution begins.

Concurrent hello world

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

class Myfunctor{
public:
 void operator();
};

int main(){
        MyFunctor m;
        std::thread t(m); //create and start a thread
        t.join(); //wait the end of the thread execution
return 0 ;
}
```

Every thread has to have an *initial function*, which is where the new thread of execution begins.

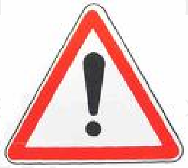
Concurrent hello world

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

class Myfunctor{
public:
 void operator();
};

int main(){
        std::thread t(MyFunctor()); //create and start a thread
        t.join(); //wait the end of the thread execution
return 0 ;
}
```

Every thread has to have an *initial function*, which is where the new thread of execution begins.

Concurrent hello world

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

class MyFunctor{
public:
  void operator();
};

int main(){
        std::thread t(MyFunctor()); //create and start a thread
        t.join(); //wait the end of the thread execution
return 0 ;
}
```



Is not understood as expected by the compiler
→ add parenthesis or **use the bracket initialization syntax**

```
std::thread t(MyFunctor());
```

✗

Every thread has to have an *initial function*, which is where the new thread of execution begins.

Concurrent hello world

Is not understood as expected by the compiler
→ add parenthesis or **use the bracket initialization syntax**

```
std::thread t{MyFunctor()};
```

Concurrent hello world

Is not understood as expected by the compiler
→ add parenthesis or **use the bracket initialization syntax**

```
std::thread t{MyFunctor()};
```

https://stackoverflow.com/questions/18222926/why-is-list-initialization-using-curly-braces-better-than-the-alternatives

Concurrent hello world

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

class Myfunctor{
 Xxxx someAttribute
public:
 MyFunctor(Xxxx someParam);
 void operator();
};

int main(){
        std::thread t{MyFunctor(xxx)}; //create and start a thread
        t.join(); //wait the end of the thread execution
return 0 ;
}
```

Every thread has to have an *initial function*, which is where the new thread of execution begins.

KAIROS

Concurrent hello world

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

int main(){
        std::thread t([](){cout << "Hello Concurrent World\n"});
                                 //create and start a thread
        t.join(); //wait the end of the thread execution
return 0 ;
}
```

Every thread has to have an *initial function*, which is where the new thread of execution begins.

Concurrent hello world

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

int main(){
        std::thread t([](){cout << "Hello Concurrent World\n"});
                                //create and start a thread
        t.join(); //wait the end of the thread execution
return 0 ;
}
```

For any versions, add `-pthread` to the linker flags

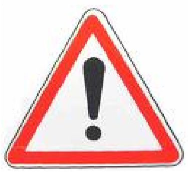
Concurrent hello world

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

int main(){
        std::thread t([](){cout << "Hello Concurrent World\n"});
                                //create and start a thread
        t.join(); //wait the end of the thread execution
        //t.detach();
return 0 ;
}
```

The destructor of std::thread calls `std::terminate()`
→ you may choose to detach or join

Joining a thread

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



`thread::join()` cleans up the storage associated with the thread.
After `join()` returned, `t` is not associated with a thread anymore
`t.joinable()` returns `false`
→ `thread::join()` can be called only once per thread object

Thread object life duration

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

int main(){
        std::thread t([](){cout << "Hello Concurrent World\n"});
                                //create and start a thread
        t.detach(); //autonomous life time duration for the thread
return 0 ;
}
```

The destructor of std::thread calls `std::terminate()`
→ you may choose to detach or join

The notion of Object life duration is to take into account carefully !

Detaching and object life duration

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

doSomethingLong(int& intref);  ✗

int main(){
        int anInt = 0;
        std::thread t{doSomethingLong, anInt};
                                //create and start a thread
        t.detach(); //autonomous life time duration for the thread
return 0 ;
}//anInt is destroyed and the thread can still access it !
```

The destructor of std::thread calls `std::terminate()`
→ you may choose to detach or join

The notion of Object life duration is to take into account carefully !

Detaching and object life duration

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

doSomethingLong(int intcopy);

int main(){
        int anInt = 0;
        std::thread t{doSomethingLong, anInt};
                                    //create and start a thread
        t.detach(); //autonomous life time duration for the thread
return 0 ;
}//anInt is destroyed and the thread can still access it !
```

The destructor of std::thread calls `std::terminate()`
→ you may choose to detach or join

The notion of Object life duration is to take into account carefully !

Passing parameters to thread functions

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

doSomething(int intcopy, string s);

int main(){
    int anInt = 0;
    std::thread t{doSomething, anInt, string("a string !")};
    t.join();
return 0 ;
}
```

Explicit call to string constructor to avoid conversion from const char * to int in doSomething...

Passing parameters to thread functions

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

doSomething(int intcopy, string s);

int main(){
     int anInt = 0;
     string s = “a string !”;
     std::thread t{doSomething, anInt, s};
     t.join();
return 0 ;
}
```

Explicit call to string constructor to avoid conversion from const char * to int in doSomething...

Passing parameters to thread functions

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

doSomething(string& s){ s+="!"}

int main(){
    int anInt = 0;
    std::thread t{doSomething, std::string("a string !")};
    t.join();
return 0 ;
}
```

Explicit call to string constructor to avoid conversion from const char * to string *inside* doSomething…

Thread constructor is unaware of reference passing and will copy first :-/
→ need to make the reference explicit

Passing parameters to thread functions

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

doSomething(string& s){ s+="!"}

int main(){
    int anInt = 0;
    std::thread t{doSomething, std::ref(std::string("a string !"))};
    t.join();
return 0 ;
}
```

Explicit call to string constructor to avoid conversion from const char * to string *inside* doSomething…

Thread constructor is unaware of reference passing and will copy first :-/
→ need to make the reference explicit

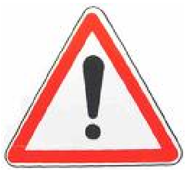
Passing parameters to thread functions

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

class X{
public: doSomething(int);
};

int main(){
    int anInt = 0;
    X myX;
    std::thread t{&X::doSomething, myX, 3}; //myX.doSomething(3);
    t.join();
return 0 ;
}
```

Calling a member function: remember the hidden parameter !

Passing a thread as parameter

```
void f(std::thread t);

void g()
{
  void some_function();
  f(std::thread(some_function));
  std::thread t(some_function);
  f(std::move(t));
//here 't' is empty...
}
```



Warning, a thread cannot be copied ! Only "moved"
(see move semantics for more details)

Shared data and mutex

```
#include <list>
#include <mutex>
#include <algorithm>

std::list<int> aList;
std::mutex aListMutex;
void add_to_list(int new_value){
  aListMutex.lock();
  aList.push_back(new_value);
  aListMutex.unlock();
}
bool list_contains(int value_to_find){
  aListMutex.lock();
  bool res = std::find(
          aList.begin(),
          aList.end(),
          value_to_find
          )
          != aList.end();
  aListMutex.unlock();
  return res;
}
```

Classical mutex...

Shared data and mutex

```
#include <list>
#include <mutex>
#include <algorithm>

std::list<int> aList;
std::mutex aListMutex;
void add_to_list(int new_value){
  std::lock_guard<std::mutex> guard(aListMutex);
  aList.push_back(new_value);
}

bool list_contains(int value_to_find){
  std::lock_guard<std::mutex> guard(aListMutex);
  return std::find(
        aList.begin(),
        aList.end(),
        value_to_find
        )
        != aList.end();
}
```

Mutex is taken until guard is destroyed !

Shared data and mutex

```
#include <list>
#include <mutex>
#include <algorithm>

std::list<int> aList;
std::mutex aListMutex;

void recursive_add_to_list(int new_value){
  if(new_value <= 0)return;
  aListMutex.lock();
  aList.push_back(new_value);
  aListMutex.unlock();
  recursive_add_to_list(newValue--)
}
```

Warning to correctly free the mutex before the recursion

→ **but non atomic**

KAIROS

Shared data and mutex

```
#include <list>
#include <mutex>
#include <algorithm>

std::list<int> aList;
std::recursive_mutex aListMutex;

void recursive_add_to_list(int new_value){
  if(new_value <= 0)return;
  aListMutex.lock();
  aList.push_back(new_value);
  recursive_add_to_list(newValue--);
  aListMutex.unlock();
  }
}
```

recursive_mutex can be locked several time by the same thread !
Need to be unlocked as many as locked !
→ **atomic**

KAIROS

Condition variable

```
std::mutex mut;
int anInt;                              //shared variables
std::condition_variable anInt_cond;

void f(){
 while(true){
  std::lock_guard<std::mutex> guard(mut);
  anInt = computeNewInt();
  anInt_cond.notify_one();
 }
}

void g(){
 while(true){
  std::unique_lock<std::mutex> guard(mut);
  anInt_cond.wait(guard,[]{return anInt == somethingSpecial;});
  //Do whatever to do
  guard.unlock();
  //continue doing something or not…
 }
}
```

Condition false → mutex unlocked and thread “blocked”
Condition true → control and thread given to thread

Asynchronous tasks and Futures

```
#include<future>
#include<iostream>

int solving_ltuae();

void do_other_stuff();
int main()
{
  std::future<int> the_answer=std::async(solving_ltuae);
  do_other_stuff();
  std::cout<<"The answer is "<<the_answer.get()<<std::endl;
return 0;
}
```

async runs a function and eventually returns a future
get() is blocking if the future is not ready, yet.

Asynchronous tasks and Futures

```
#include<future>
#include<iostream>

int solving_ltuae(int);

void do_other_stuff();
int main()
{
  std::future<int> the_answer=std::async(solving_ltuae, 42);
  do_other_stuff();
  std::cout<<"The answer is "<<the_answer.get()<<std::endl;
return 0;
}
```

async runs a function and eventually returns a future
get() is blocking if the future is not ready, yet.

promise and Futures

```
#include<future>
#include<iostream>

int solving_ltuae(std::promise<int>);
// at some point the function does: promiseObj.set_value(42);

void do_other_stuff();
int main()
{
  std::promise<int> promiseObj;
  std::future<int> futureObj = promiseObj.get_future();
  Thread t(solving_ltuae, promiseObj);
  do_other_stuff();
  std::cout<<"The answer is "<<futureObj.get()<<std::endl;
return 0;
}
```

Asynchronous tasks and Futures

```
#include<future>
#include<iostream>

int solving_ltuae(int);

void do_other_stuff();
int main()
{
  std::future<int> the_answer=std::async(solving_ltuae, 42);
  future<string> f2 = the_answer.then([](future<int> f) {
  return to_string(f.get());
  });

  do_other_stuff();
  std::cout<<"The answer is "<<f2.get()<<std::endl;
return 0;
}
```

async runs a function and eventually returns a future
get() is blocking if the future is not ready, yet.

C++20 adds a nice way to compose futures

Asynchronous tasks and Futures

```
#include<future>
#include<iostream>

int solving_ltuae(int);

void do_other_stuff();
int main()
{
  std::future<int> the_answer=std::async(solving_ltuae, 42);
  future<string> f2 = the_answer.then([](future<int> f) {
  return to_string(f.get());
  });

  do_other_stuff();
  std::cout<<"The answer is "<<f2.get()<<std::endl;
return 0;
}
```

async runs a function and eventually returns a future
get() is blocking if the future is not ready, yet.

C++20 adds a nice way to compose futures

Asynchronous tasks and Futures

```
#include<future>
#include<iostream>

int solving_ltuae(int);

void do_other_stuff();
int main()
{
  std::future<int> the_answer=std::async(solving_ltuae, 42);
  future<string> f2 = the_answer.then([](future<int> f) {
  return to_string(f.get());
  });

  do_other_stuff();
  std::cout<<"The answer is "<<f2.get()<<std::endl;
return 0;
}
```

when_any

→ still experimental

When all

async runs a function and eventually returns a future
get() is blocking if the future is not ready, yet.

C++20 adds a nice way to compose futures

Algorithm and concurrency in C++17

```
#include <algorithm>
#include <iterator>




std::reverse(
       this->ownedPixels.begin(),
       this->ownedPixels.end()
);
```

The algorithm is used in sequence...

Algorithm and concurrency in C++17

All algorithms

69 of the algorithms of the STL support a parallel or a parallel and vectorized execution. Here they are.

Standard library algorithms for which parallelized versions are provided [Collapse]

- std::adjacent_difference
- std::adjacent_find
- std::all_of
- std::any_of
- std::copy
- std::copy_if
- std::copy_n
- std::count
- std::count_if
- std::equal
- std::fill
- std::fill_n
- std::find
- std::find_end
- std::find_first_of
- std::find_if
- std::find_if_not
- std::generate
- std::generate_n
- std::includes
- std::inner_product
- std::inplace_merge
- std::is_heap
- std::is_heap_until
- std::is_partitioned
- std::is_sorted
- std::is_sorted_until
- std::lexicographical_compare
- std::max_element
- std::merge
- std::min_element
- std::minmax_element
- std::mismatch
- std::move
- std::none_of
- std::nth_element
- std::partial_sort
- std::partial_sort_copy
- std::partition
- std::partition_copy
- std::remove
- std::remove_copy
- std::remove_copy_if
- std::remove_if
- std::replace
- std::replace_copy
- std::replace_copy_if
- std::replace_if
- std::reverse
- std::reverse_copy
- std::rotate
- std::rotate_copy
- std::search
- std::search_n
- std::set_difference
- std::set_intersection
- std::set_symmetric_difference
- std::set_union
- std::sort
- std::stable_partition
- std::stable_sort
- std::swap_ranges
- std::transform
- std::uninitialized_copy
- std::uninitialized_copy_n
- std::uninitialized_fill
- std::uninitialized_fill_n
- std::unique
- std::unique_copy

https://www.modernescpp.com/index.php/parallel-algorithm-of-the-standard-template-library

In addition, we get 8 new algorithms.

Algorithm and concurrency in C++17

```
#include <algorithm>
#include <iterator>
#include <execution>



std::reverse(std::execution::par,
        this->ownedPixels.begin(),
        this->ownedPixels.end()
);
```

The algorithm is used in parallel in different threads…
but it’s your responsibility to avoid race conditions

Algorithm and concurrency in C++17

```
#include <algorithm>
#include <iterator>
#include <execution>




std::reverse(std::execution::par_unseq,
       this->ownedPixels.begin(),
       this->ownedPixels.end()
);
```

The algorithm is used in parallel in different threads
and potentially makes used of Vectorization…

but it’s your responsibility to avoid race conditions

Algorithm and concurrency in C++17

Vectorization means that the compiler detects that your independent instructions can be executed as one SIMD instruction. Usual example is that if you do something like

```
for(i=0; i<N; i++){
  a[i] = a[i] + b[i];
}
```

It will be vectorized as (using vector notation)

```
for (i=0; i<(N-N%VF); i+=VF){
  a[i:i+VF] = a[i:i+VF] + b[i:i+VF];
}
```

Basically the compiler picks one operation that can be done on VF elements of the array at the same time and does this N/VF times instead of doing the single operation N times.

It increases performance, but puts more requirement on the architecture.

https://stackoverflow.com/a/1516648

Algorithm and concurrency in C++17

```
#include <algorithm>
#include <iterator>
#include <execution>




std::reverse(std::execution::par_unseq,
        this->ownedPixels.begin(),
        this->ownedPixels.end()
);
```

The algorithm is used in parallel in different threads and potentially makes used of Vectorization…

but it’s your responsibility to avoid race conditions

Algorithm and concurrency in C++17

```
#include <algorithm>
#include <iterator>
#include <execution>
std::reverse(std::execution::par_unseq,
        this->ownedPixels.begin(),
        this->ownedPixels.end()
);
```

Historically, accelerating your C++ code with GPUs has not been possible in Standard C++ without using language extensions or additional libraries:

- CUDA C++ requires the use of `__host__` and `__device__` attributes on functions and the triple-chevron syntax for GPU kernel launches.
- OpenACC uses #pragmas to control GPU acceleration.
- Thrust lets you express parallelism portably but uses language extensions internally and only supports a limited number of CPU and GPU backends. The portability of the application is limited by the portability of the library.

Standard Parallelism

In many cases, the results of these ports are worth the effort. But what if you could get the same effect without that cost? What if you could take your Standard C++ code and accelerate on a GPU?

Now you can! NVIDIA recently announced NVC++, the NVIDIA HPC SDK C++ compiler. This is the first compiler to support GPU-accelerated Standard C++ with no language extensions, pragmas, directives, or non-standard libraries. You can write Standard C++, which is portable to other compilers and systems, and use NVC++ to automatically accelerate it with high-performance NVIDIA GPUs. We built it so that you can spend less time porting and more time on what really matters—solving the world's problems with computational science.

https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar/

Many more but not addressed here…