C++ 11 - Overview

Berenger Bramas - HiePACS - Inria

Summary

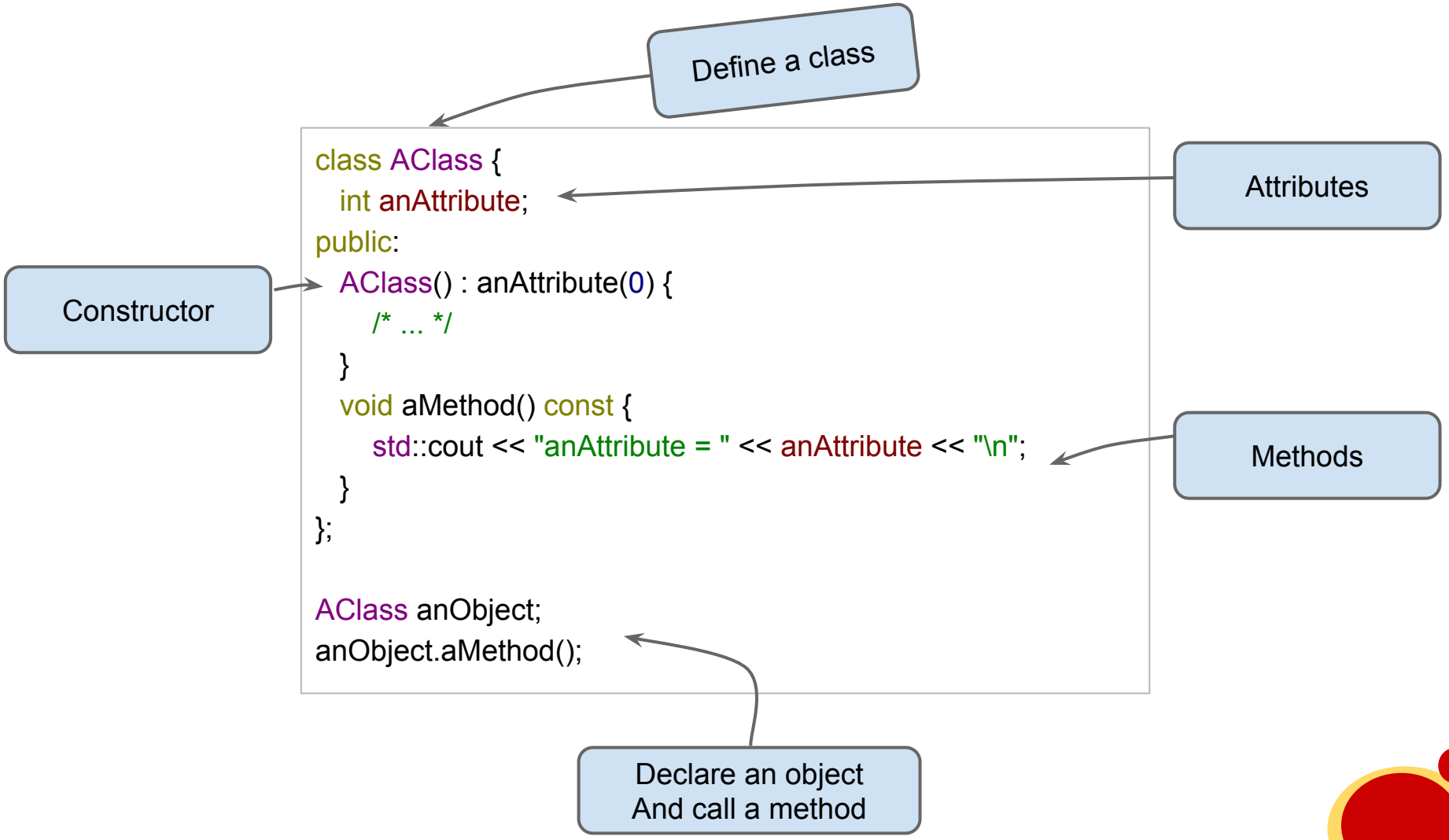
- Remind, What is C++ about?
- C++11 (everything except concurrency)
- C++11 (everything about concurrency)
- Some advice and the conclusion

C++ ?

- Object oriented language
- Compiled language
- Looks like C (and is mainly C compatible)
- Previous standard: C++03

- Can achieve the same performance as C
  - But, can be slower than C because of the developer...

C++ - Example - 1


Define a class
class AClass {
int anAttribute;
public:
AClass() : anAttribute(0) {
/* ... */
}
void aMethod() const {
std::cout << "anAttribute = " << anAttribute << "\n";
}
};
AClass anObject;
anObject.aMethod();
Attributes
Constructor
Methods
Declare an object
And call a method

C++ - Example - 2

```
std::vector<int> giveMeAVector(){
   return std::vector<int>();
}
```

Return an empty vector by value

```
std::vector<int> vec = giveMeAVector();

for(std::vector<int>::iterator iter = vec.begin() ; iter != vec.end() ; iter++){
   /* .... */
}
```

Iterate on the vector obtained from “giveMeAVector”

C++ - Example - 2

```
std::vector<int> giveMeAVector(){
   return std::vector<int>();
}

std::vector<int> vec = giveMeAVector();

for(std::vector<int>::iterator iter = vec.begin() ; iter != vec.end() ; iter++){
   /* .... */
}
```

Copy

Temp objects

```
iterator end(){
    iterator myEnd();
    /** init iterator */
    return current_value;
}
```

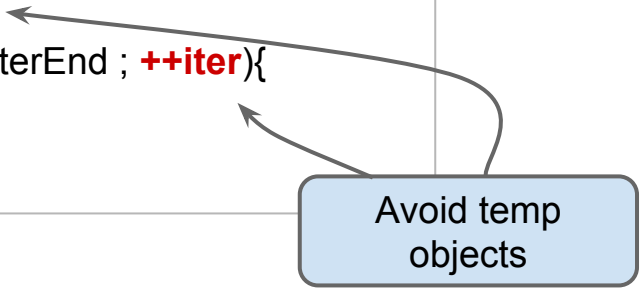
```
operator post ++ {
    current_value = iter;
    ++iter;
    return current_value;
}
```

C++ - Example - 2

```
std::vector<int> vec = giveMeAVector();
```

```
for(std::vector<int>::iterator iter = vec.begin() ; iter != vec.end() ; iter++){
   /* .... */
}
```

```
const std::vector<int>::iterator iterEnd = vec.end();
for(std::vector<int>::iterator iter = vec.begin() ; iter != iterEnd ; ++iter){
   /* .... */
}
```



C++11 Introduction

- I used Gcc 4.9 (should work with Intel > 15)
  - Flag: -std=c++0x or -std=gnu++0x
  - -pthread to enable threading support on POSIX OS
- Proposed new possibilities in many areas:
  - Syntax
  - Core language
  - Stl
  - Concurrency
  - …
- In most cases it helps to improve the design and the portability but also the performance

C++11 - Everything except Concurrency

C++11 - nullptr

```
void print(char *){
   std::cout << "print(char*) \n";
}

void print(int){
   std::cout << "print(int) \n";
}

print(0);
print(NULL);
```

Which function to call?

C++11 - nullptr

```
void print(char *){
   std::cout << "print(char*) \n";
}

void print(int){
   std::cout << "print(int) \n";
}

print(0);
print(NULL);
```

Which function to call?

error: call of overloaded ‘print(NULL)’ is ambiguous

#define NULL 0 ?

C++11 - nullptr

```
void print(char *);
void print(int);

print(0); // => print(int)
// print(NULL); // error: call of overloaded ‘print(NULL)’ is ambiguous
print(reinterpret_cast<char *>(0)); // => print(char*)
print(nullptr); // => print(char*)

char *pointerToChar = nullptr;     // OK
int  *pointerToInt  = nullptr;     // OK
bool  aBool         = nullptr;     // OK. b is false.
if( nullptr );                     // OK. it is like false.

// int aInt = nullptr;         // error, no conversion known to int
// ((void*)0) and 0 still work but should be avoid
```

Nullptr is the real NULL!

C++11 - auto & decltype

Error! The compiler knows that “int*” is not the correct type

```
int* ptrInt = new char[10];
```

The compiler could answer that

```
int func(){
}

?? a = 0;
?? b = func();
?? c = a;
```

C++11 - auto & decltype

```
int func(){
}

auto a = 0;
auto b = func();
decltype(a) c = a;
```

auto : let the compiler choose
decltype(X) : same type as X

There is no extra-cost! Everthing is done at compile time.

C++11 - auto & decltype

```
// Create a vector of size 1
const std::vector<int> vectorOfInt(1);

// Test some declarations
auto a = vectorOfInt[0];           // a has type int
decltype(vectorOfInt[1]) b = 1; // b has type const int&, the return type of
                                   // std::vector<int>::operator[](size_type) const
auto c = 0;            // c has type int
auto d = c;            // d has type int
decltype(c) e;        // e has type int, the type of the entity named by c
decltype((c)) f = c;  // f has type int&, because (c) is an lvalue (you can forget that)
decltype(0) g;         // g has type int, because 0 is an rvalue
std::vector< decltype(0) > anotherVectorOfInt;

// Old style: for (std::vector<int>::const_iterator itr = vectorOfInt.cbegin(); itr != vectorOfInt.cend(); ++itr)
for (auto itr : vectorOfInt){
   // Work with itr
}
```

C++11 - Stl - Hashmap

```
std::unordered_map<std::string,std::string> stringToStringMap;

stringToStringMap["a key"] = "a value";             // Create key and set the value

std::string name = stringToStringMap["a key"];   // existing element accessed ("a value")

stringToStringMap["another key"] = name;         // new element

stringToStringMap["copy of key with no value"] = stringToStringMap["key with no value"];

for (auto& keyValIter: stringToStringMap) {
   std::cout << keyValIter.first << ": " << keyValIter.second << std::endl;
}
```

User can customize the “hash” function
Collision are managed with linear chaining

Output:
copy of key with no value:
key with no value:
another key: a value
a key: a value

C++11 - Stl - HashMultimap

```
// With many values per key
std::unordered_multimap<std::string, std::string> stringToStringMultiMap;

stringToStringMultiMap.insert(std::make_pair<std::string,std::string>("a key","value 1"));
stringToStringMultiMap.insert(std::make_pair<std::string,std::string>("a key","value 2"));
stringToStringMultiMap.insert(std::make_pair<std::string,std::string>("another key","value 3"));

for (auto& keyValIter: stringToStringMultiMap) {
   std::cout << keyValIter.first << ": " << keyValIter.second << std::endl;
}
```

Output:
another key: value 3
a key: value 2
a key: value 1

std::unordered_set
std::unordered_multiset
std::unordered_map
std::unordered_multimap
Associated values × Equivalent keys

C++11 - Keywords

```
class Base{
public:
   explicit Base(int){}
   virtual void work(){}
   virtual void workVirtual() = 0;
};
```

```
// In C++03
class Derivate03 : public Base{
private:
   Derivate03(const Derivate03&){}
public:
   explicit Derivate03(int inVal) : Base(inVal){}
   void work(){}
   void workVirtual(){}
};
```

C++11 - Keywords

```
class Base{
public:
   explicit Base(int){}
   virtual void work(){}
   virtual void workVirtual() = 0;
};
```

```
// In C++03
class Derivate03 : public Base{
private:
   Derivate03(const Derivate03&){}
public:
   explicit Derivate03(int inVal) : Base(inVal){}
   void work(){}
   void workVirtual(){}
};
```

```
// In C++11
class Derivate11 : public Base{
public:
   using Base::Base;
   Derivate11(const Derivate11&) = delete;
   void work() override {}
   void workVirtual() override final {}
   // void workNotInherited() override {} Ill-formed
};
```

C++11 - Lambda Function

```
bool compareInt(const int i1, const int i2){
   return i1 < i2;
}

struct CompareIntClass{
   bool operator()(const int i1, const int i2){
      return i1 < i2;
   }
};

// C++03
std::vector<int> vecOfInts = { 1, 2, 3};
std::sort(vecOfInts.begin(), vecOfInts.end()); // Default compare operator <
std::sort(vecOfInts.begin(), vecOfInts.end(), compareInt); // Using external function
std::sort(vecOfInts.begin(), vecOfInts.end(), CompareInt()); // Using external class
```

Heavy! And even more heavy with external dependencies

C++11 - Lambda Function

[capture](parameters) -> return_type { function_body }

```
std::vector<int> vecOfInts = { 1, 2, 3};

std::sort(vecOfInts.begin(), vecOfInts.end(),[](const int& v1, const int& v2) -> bool {
   return v1 < v2;
});
```

C++11 - Lambda Function

```
std::vector<int> vecOfInts = { 1, 2, 3};

std::sort(vecOfInts.begin(), vecOfInts.end(),[](const int& v1, const int& v2) -> bool {
   return v1 < v2;
});
```

```
std::vector<int> vecOfInts = { 1, 2, 3};

bool anonymousFunction(const int i1, const int i2){
   return i1 < i2;
}

std::sort(vecOfInts.begin(), vecOfInts.end(),anonymousFunction);
```

C++11 - Lambda Function

```
auto aLambdaFunc = new auto([=](int x) { /*...*/ });
auto aLambdaFuncWithReturn = new auto([=](int x) -> int { /*...*/ ; return 0; });

std::function<void(int)> aFuncPtr = [=](int x) { /*...*/ };
std::function<int(int)> aFuncPtrWithReturn = [=](int x) -> int { /*...*/ ; return 0;};

int a = 0, b = 0;
std::function<int(int)> aFuncPtrWithExternalDep = [&a, b](int x) -> int { /*...*/ ; return (++a) + b;};
```

We can still have pointers to functions.

Capture:
[=] all by values
[&] all by references
[var] var by value
[&var] var by reference

Watch the objects lifetime!

C++11 - Lambda Function

```
template <class ContainerClass, class ElementClass>
void CallForEach(const ContainerClass& container, std::function<void(ElementClass)> func){
  for(auto val : container){
     func(val);
  }
}
```

Apply func to all elements in container

```
std::vector<int> vecOfInts = { 1, 2, 3};
int total = 0;

int passedByValue = 10;
CallForEach<decltype(vecOfInts), int>(vecOfInts, [passedByValue, &total](const int & val){
  total += val * passedByValue;
});
std::cout << "Total = " << total << "\n";
```

Output: Total = 60

C++11 - Variadic Template

```
template <typename... ManyTypes>
void variadicFunction(ManyTypes... manyVariables){
}

variadicFunction(0, "string", 10.4, true);
```

```
template <typename... ManyTypes>
void prePrintf(ManyTypes... manyVariables){
   printf("Someone calls printf!\n");
   printf(manyVariables...);
}

prePrintf("%d %s %e\n", 0, "string", 10.4);
```

Unlimited number of templates
(and unlimited number of parameters)

C++11 - Rvalue/move/constructors

```
MySmartVec<int> buildAVec(){
  MySmartVec<int> aVec;
  aVec.resize(4000);
  return aVec;
}
```

Return by value a big vector

```
MySmartVec<int> aCopy1;
MySmartVec<int> aCopy2 = buildAVec(); // copy a temp object
aCopy1 = buildAVec(); // copy a temp object
```

```
MySmartVec< MySmartVec<int> > vecOfVec;
{
  MySmartVec<int> aBigVec;
  aBigVec.resize(10000);
  vecOfVec.push_back(aBigVec); // aBigVec is copied
  // aBigVec is not used any more after this line!
}
```

C++11 - Rvalue/move/constructors

```
MySmartVec<int> buildAVec(){
  MySmartVec<int> aVec;
  aVec.resize(4000);
  return aVec;
}
```

Do not call copy constructor, but rvalue copy constructor

```
MySmartVec<int> aCopy1;
MySmartVec<int> aCopy2 = buildAVec();
aCopy1 = buildAVec();
```

```
MySmartVec< MySmartVec<int> > vecOfVec;
{
  MySmartVec<int> aBigVec;
  aBigVec.resize(10000);
  vecOfVec.push_back(std::move(aBigVec));
}
```

Do not call copy operator, but rvalue copy operator

C++11 - Rvalue/move/constructors

```
MySmartVec<int> aCopy1 = existingVec;

MySmartVec(const MySmartVec& other) : ptrData(nullptr), nbElements(0) {
  ptrData    = new ObjectType[other.nbElements];
  nbElements = other.nbElements;
  std::copy_n(other.ptrData, other.nbElements, ptrData);
}
```

Copy constructor : duplicate the data

```
MySmartVec<int> aCopy1 = std::move(existingVec);

MySmartVec(MySmartVec&& other) : ptrData(nullptr), nbElements(0) {
  ptrData    = other.ptrData;
  nbElements = other.nbElements;
  other.ptrData    = nullptr;
  other.nbElements = 0;
}
```

Copy rvalue constructor: become the owner of the data but keep the source consistent

Concurency with C++11

C++11 - Concurrency

Now C++ supports:

- Threads
- Mutexes
- Atomics
- Conditions
- Asynchronous tasks
- Lock-Free functions
- Futures/Promises

Before, external libs were needed

C++11 - Create threads 1

```
void background_task(){
  std::cout << "I am " << std::this_thread::get_id() << "\n";
}
```

```
std::thread my_thread1(background_task);

std::thread my_thread2{ background_task };

std::thread my_thread3([](){
  background_task();
});
```

```
my_thread1.join(); my_thread2.join(); my_thread3.join(); // We must join
```

Callback function

Create threads

Wait threads

C++11 - Create threads 2

```
class background_class{
public:
   void operator()() const{
      background_task();
   }
};
```

```
std::thread my_thread1( (background_class()) );

std::thread my_thread2{ background_class() };

background_class cl;
std::thread my_thread3([&](){
   cl(); // Be sure cl is alive here!
});
```

```
my_thread1.join(); my_thread2.join(); my_thread3.join(); // We must join
```

Callback class

Create threads

Wait threads

C++11 - Create threads 3

```
void background_task_with_param(int input){
   std::cout << "I am " << std::this_thread::get_id() << " with input " << input << "\n";
}
class background_class{
public:
   void afunc(int input) const{
      background_task_with_param(input);
   }
};
```

Callback function and class with parameter

```
std::thread my_thread1(background_task_with_param, 40);

background_class cl;
std::thread my_thread2(&background_class::afunc, &cl, 40);

std::thread my_thread3([&](){
   cl.afunc(40); // Be sure cl is alive here!
});
```

Create threads (use callback with parameter)

```
my_thread1.join(); my_thread2.join(); my_thread3.join(); // We must join
```

C++11 - Threads detached

```
const int nbThreads = std::thread::hardware_concurrency();

std::unique_ptr<std::thread[]> threads(new std::thread[nbThreads]);

for(int idxThread = 0 ; idxThread < nbThreads ; idxThread){
   assert(!threads[idxThread].joinable());
   threads[idxThread] = std::thread(background_task);
}
```

Create threads based on the number of cores

```
if(waitToComplete){
   for(int idxThread = 0 ; idxThread < nbThreads ; idxThread){
      threads[idxThread].join();
   }
}
```

Wait threads to complete

```
else{
   for(int idxThread = 0 ; idxThread < nbThreads ; idxThread){
      threads[idxThread].detach();
      assert(!threads[idxThread].joinable());
   }
}
```

Or detach the threads (they are no longer joinable)

C++11 - Future

Future is a way to get a result that will be available

```
int thread_work(){
   /* ... */
   return 40;
}
std::future<int> result_accessor = std::async(thread_work);
// Master thread can do other thing
std::cout << "The result is "<< result_accessor.get() << std::endl;
```

Launch a task

get: return the result maybe now or when it will be available

Waiting from many threads :
std::future is movable only
std::shared_future is copyable

C++11 - Future

```
std::future<int> result_accessor = std::async([&]() -> int {
   return 40;
 });
// Master thread can do other thing
std::cout << "The result is "<< result_accessor.get() << std::endl;
```

async with lambda function

C++11 - Future

```
std::future<int> result_accessor = std::async([&]() -> int {
   return 40;
 });
// Master thread can do other thing
std::cout << "The result is "<< result_accessor.get() << std::endl;
```

async with lambda function

```
std::future<int> result_accessor = std::async(std::launch::deferred | std::launch::async,
                                              [&]() -> int {
                                                  return 40;
                                              });
// Master thread can do other thing
std::cout << "The result is "<< result_accessor.get() << std::endl;
```

async with lambda function + launch type

std::launch::deferred: task run in wait or get
std::launch::deferred | std::launch::async (like default) implementation chooses

C++11 - Future + Promise

A promise should be set by the producer,
And the consumer retrieve the value using a future

```
std::promise<int> promisingValue;
std::future<int> result_accessor = promisingValue.get_future();
```

Connecte future and promise

```
std::thread worker = std::thread([&]() {
   // Can sleep
   promisingValue.set_value( 40 );
});
```

Run a thread that set the promise

```
// Master thread can do other thing
std::cout << "The result for promising is "<< result_accessor.get() << std::endl;
worker.join();
```

Ask for the result and join the thread

C++11 - Mutex

```
std::mutex listMutex;
std::list<int> listOfInt;

std::thread t1 = std::thread([&](){
   listMutex.lock();
   listOfInt.push_back(1);
   listMutex.unlock();
});

t1.join();
```

Mutex is not copyable:
Not possible std::mutex listMutexCopy = listMutex;
Not possible: std::mutex listMutexCopy = std::move(listMutex);

Recursvie locking of mutex has undefined behavior
Instead use std::recursive_mutex

C++11 - Mutex

```
std::mutex listMutex;
std::list<int> listOfInt;

std::thread t1 = std::thread([&](){
   listMutex.lock();
   listOfInt.push_back(1);
   listMutex.unlock();
});

std::thread t2 = std::thread([&](){
   std::lock_guard<std::mutex> locker(listMutex);
   listOfInt.push_back(2);
});

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

The mutex is locked when the lock_guard is created (and relaxed when the object is destroyed)

Lock_guard is not copyable (or movable)

C++11 - Atomic/Indivisible operation

```
std::atomic<bool> atomBool;
std::atomic<int> atomInt;

std::thread t1 = std::thread([&](){
   atomBool = true;
   atomInt++;
});
atomBool = false;
atomInt += 4;
t1.join();
```

There exist alias:
atomic_bool std::atomic<bool>
atomic_int : std::atomic<int>
even atomic_ptrdiff_t

C++11 - Atomic/Indivisible operation

```
std::atomic<bool> atomBool;
std::atomic<int> atomInt;

std::thread t1 = std::thread([&](){
   atomBool = true;
   atomInt++;
});
atomBool = false;
atomInt += 4;
t1.join();
```

```
std::cout << "atomBool.is_lock_free() " << atomBool.is_lock_free() << "\n";
std::cout << "atomInt.is_lock_free() " << atomInt.is_lock_free() << "\n";
```

There exist alias:
atomic_bool std::atomic<bool>
atomic_int : std::atomic<int>
even atomic_ptrdiff_t

Usually implemented as lock free (but can be emulate)
We can test using is_lock_free

C++11 - Atomic/Indivisible operation

```
std::atomic<bool> atomBool;
std::atomic<int> atomInt;

std::thread t1 = std::thread([&](){
   atomBool = true;
   atomInt++;
});
atomBool = false;
atomInt += 4;
t1.join();
```

There exist alias:
atomic_bool std::atomic<bool>
atomic_int : std::atomic<int>
even atomic_ptrdiff_t

```
std::cout << "atomBool.is_lock_free() " << atomBool.is_lock_free() << "\n";
std::cout << "atomInt.is_lock_free() " << atomInt.is_lock_free() << "\n";
```

Usually implemented as lock free (but can be emulate)
We can test using is_lock_free

```
std::atomic<bool> atomBool2;
// NO! atomBool2 = atomBool;
atomBool2.store( atomBool.load() );
```

Not copyable or assignable

```
const int oldValue = atomInt.fetch_add(4);
int currentValue = std::atomic_load(&atomInt); // same as atomInt.load()
```

C++11 - Condition

```
std::mutex queueMutex;
std::queue<int> dataQueue;
std::condition_variable dataConditions;
```

```
std::thread t1 = std::thread([&](){
  for(int idx = 0 ; idx <= 10 ; ++idx){
    std::lock_guard<std::mutex> lk(queueMutex);
    std::cout << "Produce " << idx << "\n";
    dataQueue.push(idx);
    dataConditions.notify_one();
  }
});
```

```
t1.join();
```

C++11 - Condition

```
std::mutex queueMutex;
std::queue<int> dataQueue;
std::condition_variable dataConditions;
```

```
std::thread t1 = std::thread([&](){
  for(int idx = 0 ; idx <= 10 ; ++idx){
    std::lock_guard<std::mutex> lk(queueMutex);
    std::cout << "Produce " << idx << "\n";
    dataQueue.push(idx);
    dataConditions.notify_one();
  }
});
```

```
std::thread t2 = std::thread([&](){
  while(true){
    std::unique_lock<std::mutex> lk(queueMutex);
    dataConditions.wait( lk, [&]{return !dataQueue.empty();});

    const int data = dataQueue.front();
    dataQueue.pop();
    lk.unlock();

    std::cout << "Consumme " << data << "\n";
    if(data == 10){
      break;
    }
  }
});
```

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

C++11 - Condition

```
std::mutex queueMutex;
std::condition_variable dataConditions;

std::lock_guard<std::mutex> lk(queueMutex);
dataConditions.notify_one();
dataConditions.notify_all();
```

Notify one or all

```
std::mutex queueMutex;
std::condition_variable dataConditions;

std::unique_lock<std::mutex> lk(queueMutex);
dataConditions.wait( lk, [&]{return test function;});
dataConditions.wait_for( time, lk, [&]{return test function;});
```

Wait until true or with a timeout

C++11 - Thread storage

```
thread_local int xThreadVariable;

class ClassWithStaticThread {
   static thread_local std::string threadStringVariable;
};
thread_local std::string ClassWithStaticThread::threadStringVariable;

void funtionWithThreadVariable(){
   thread_local std::vector<int> vectorThreadVariable;
   std::cout << "vectorThreadVariable @"  << (&vectorThreadVariable) << "\n";
}

for(int idx = 0 ; idx < 4 ; ++idx){
    std::thread th = std::thread([&](){
        std::cout << "xThreadVariable @"  << (&xThreadVariable) << "\n";
        funtionWithThreadVariable();
    });
    th.detach();
}
```

All threads work on different variables here

C++11 - I did not talk about...

- Type support (basic types, RTTI, type traits)
- typeinfo
- Shared/Smart/Unique pointers
- Extern template
- String literals
- constexpr
- Time/duration
- (*) static_assert, memory alignement, list initializer, emplace_back, std::array
- (*) More about concurrency

(*) Some extra slides at the end present these novelties

C++11 - Conclusion

- Nothing heavy
- Help to design and to get performance
- It takes some time to really master
- Be carreful about the compiler!
- … And what about C++14?

C++11 - Should I use it?

- Yes, but not too much!
- New applications should use it
  - Every design keyword (override, …)
  - But do not put auto everywhere!
- Existing applications should switch to it slowly
  - When you refactor some code
  - It might be very benefit! (even without doing anything)
- If you agree to constraint your users to have a compiler that supports C++11

References

- Wikipedia! C++11 page and some others

http://en.wikipedia.org/wiki/C%2B%2B11

- CppReference.com

http://cppreference.com

- C++ Concurrency in Action

Thanks!

Questions?

C++ - Example - 3

For any type

For vec of list type

For two parameters

```
template <class Type1>
void printAll(Type1 v1){
  std::cout << v1 << "\n";
}
template <class ElementType>
void printAll(std::vector<std::list<ElementType> > v1){
  std::cout << &v1 << "\n";
}
template <class Type1, class Type2>
void printAll(Type1 v1, Type2 v2){
  std::cout << v1 << "\n";
  std::cout << v2 << "\n";
}

std::vector<std::list<int> > vecOfLists;
float floatVal = 0.3f;
int intVal = 4;
printAll(vecOfLists);
printAll(floatVal, intVal);
```

C++11 - Memory alignement

```
// alignof & alignas
// alignas(type), is exactly equivalent to alignas(alignof(type))

int alignas(double) b;

alignas(float) unsigned char charArrayHoldAFloat[sizeof(float)];

// every object of type sse_t will be aligned to 16-byte boundary
struct alignas(16) sse_t
{
   float sse_data[4];
};

// the array "cacheline" will be aligned to 128-byte boundary
alignas(128) char cacheline[128];

//std::cout << alignof(std::max_align_t) << '\n';
std::cout << std::alignment_of<int>::value << '\n';
std::cout << std::alignment_of<double>::value << '\n';
```

C++11 - static_assert

```
static_assert(true , "This will compile.");
// static_assert(false , "This will NOT compile.");
```

```
static const int checkMe = 4;
static_assert(checkMe < 10 , "This will compile.");
```

```
template <class ObjectType, ObjectType constValue>
void testAssertFunc(){
  static_assert(10 < constValue , "This will compile.");
  static_assert(std::is_fundamental<decltype(constValue)>::value , "This will compile.");
  // static_assert(std::is_unsigned<decltype(constValue)>::value , "This will NOT compile.");
  // static_assert(std::has_virtual_destructor<decltype(constValue)>::value , "This will NOT compile.");
}

testAssertFunc<int, 40>();
```

To check at compile time and throw a message in case of problem

C++11 - List initializer

```
// C++03
struct Object {
   float first;
   int second;
};

Object scalar = {0.43f, 10}; //One Object, with first=0.43f and second=10
Object anArray[] = {{13.4f, 3}, {43.28f, 29}, {5.934f, 17}}; //An array of three Objects

// C++11 (not POD - Plain Old Data)
std::vector<int> vecOfInts = { 1, 2, 3};
std::vector<std::string> vecOfStrings = { "xyzzy", "plugh", "abracadabra" };
```

C++11 - List initializer

```
struct AltStruct {
   AltStruct(int inX, double inY) : x{inX}, y{inY} {}
private:
   int x;
   double y;
};

AltStruct var1{2, 4.3};
//  warning: narrowing conversion of ‘2.0e+0’ from ‘double’ to ‘int’ inside { }
AltStruct var2{2.0, 4};
```

```
struct IdString {
   std::string name;
   int identifier;
};

IdString get_string() {
   return {"foo", 42}; //Note the lack of explicit type.
}
```

C++11 - Stl - initializer_list

```
void function_name(std::initializer_list<float> listOfFloat){
   // Some conversion exists
   std::vector<float> vecOfFloats = listOfFloat;
}

function_name({1.0f, -3.45f, -0.4f});

float fl = 1.0;
// Not : function_name(fl);
function_name({fl});
```

initializer_list has size/begin/end

C++11 - Stl - emplace_back, array

```
// Emplace back
std::list< std::unordered_map<std::string,std::string> > lstOfMap;
lstOfMap.push_back(std::unordered_map<std::string,std::string>()); // Push a new object in back
// The previous line used temp object!
lstOfMap.emplace_back(); // Create a new object in back
lstOfMap.back();         // To work with the previous insertion
```

Avoid temp object

```
// std::array to enable working with usual array in stl algorithm
std::array<int , 6> anArrayOf6Integers; // like int [6]
```

Has copy operators, etc.

C++11 - Variadic Template

```
void printAll(){
}
```

Called if no parameter

```
template <class FirstParameterClass, typename... ManyTypes>
void printAll(FirstParameterClass firstVal, ManyTypes... manyVariables){
  std::cout << "Print : " << typeid(firstVal).name() << " = " << firstVal << "\n";
  printAll(manyVariables...);
}

printAll(0, "string", 10.4, false);
```

Output:
Print : i = 0
Print : PKc = string
Print : d = 10.4
Print : b = 0

```
template<typename ... ManyTypes>
unsigned countArgs(ManyTypes ... manyVariables){
  // Count the number of args
  return sizeof... (ManyTypes);
}

std::cout << "Result of countArgs = " << countArgs(0, "string", 10.4, false) << "\n";
```

Output:
Result of countArgs = 4

C++11 - Threads ownership

```
std::thread t1(background_task);     // Starts a thread
std::thread t2 = std::move(t1);      // The thread is own by variable t2

t1 = std::thread(background_task);   // Starts another thread

std::thread t3;                      // No execution thread created
assert(!t3.joinable());              // No thread is attached to t3 yet
t3 = std::move(t2);                  // Thread from t2 is owned by t3

//t1 = std::move(t3);                // Kill thread from t1, and move thread from t3 to t1

t3.join(); t1.join();
```

We cannot “copy” a thread object!

C++11 - Mutex

```
std::mutex listMutex;
std::list<int> listOfInt;
std::mutex listMutex2;
std::list<int> listOfInt2;

std::thread t4 = std::thread([&](){
   std::unique_lock<std::mutex> lock(listMutex,std::defer_lock);
   std::unique_lock<std::mutex> lock2(listMutex2,std::defer_lock);
   std::lock(lock,lock2); // Lock here
   if( listOfInt.size() ){
      listOfInt2.push_back( listOfInt.back() );
   }
   // Automatically unlock
});

t4.join();
```

Unique_lock is not copyable but movable

C++11 - Atomic properties

```
std::atomic<bool> x,y;
std::atomic<int> z;
x=false;
y=false;
z=0;
std::thread a([&](){
   x.store(true,std::memory_order_seq_cst);
});
std::thread b([&](){
   y.store(true,std::memory_order_seq_cst);
});
std::thread c([&](){
   while(!x.load(std::memory_order_seq_cst));
   if(y.load(std::memory_order_seq_cst)){
      ++z;
   }
});
std::thread d([&](){
   while(!y.load(std::memory_order_seq_cst));
   if(x.load(std::memory_order_seq_cst)){
      ++z;
   }
});

a.join(); b.join(); c.join(); d.join();
assert(z.load()!=0); // Cannot never fire
```

x = true

y = true

x & y
++z

y & x
++z

It exists:
memory_order_relaxed, memory_order_consume, memory_order_acquire,
memory_order_release, memory_order_acq_rel, and memory_order_seq_cst

Default is:
memory_order_seq_cst (sequentially consistent)

C++11 - Atomic properties

```
std::atomic<bool> x,y;
std::atomic<int> z;
x=false;
y=false;
z=0;

std::thread a([&](){
   y.store(true,std::memory_order_relaxed);
   x.store(true,std::memory_order_relaxed);
});
std::thread b([&](){
   while(!y.load(std::memory_order_relaxed));
   if(x.load(std::memory_order_relaxed)){
       ++z;
   }
});

a.join(); b.join();

assert(z.load()!=0); // CAN fire
```

C++11 - Lock free operations

```
template<typename T>
class lock_free_stack{
private:
  struct node{
    T data;
    node* next;
    node(T const& data_): data(data_){}
  };
  std::atomic<node*> head;
public:
  void push(T const& data){
    node* const new_node=new node(data);
    new_node->next=head.load();
    while(!head.compare_exchange_weak(new_node->next,new_node));
    // if head == new_node->next, head = new_node and return true
    // else new_node->next = head and return false
  }
};
```

Do not lock, but loop to insert a node